OS-9 Commands Reference

Contents

Chapter 1 Introduction 1-1
The Kernel 1-1
The Input/Output Manager 1-2
Device Drivers 1-2
Device Descriptors 1-2
The Shell 1-3
Going On 1-3

Chapter 2 The OS-9 File System 2-1
Input/Output Paths 2-1
Disk Directories 2-2
Subdirectories 2-3
Disk Files 2-3
Sectors 2-4
Text Files 2-5
Random-Access Data Files 2-6
Procedure Files 2-6
Executable Program Module Files 2-7
Miscellaneous File Use 2-8
The File Security System 2-8
Examining and Changing File Attributes 2-9
Record Lockout 2-11
Device Names 2-12

Chapter 3 Advanced Features of the Shell 3-1
More About Command Line Processing 3-1
Command Modifiers 3-3
Execution Modifiers 3-3
Alternate Memory Size Modifier 3-3
I/O Redirection Modifiers 3-4
Command Separators 3-5
Sequential Execution Using the Semicolon 3-6
Concurrent Execution Using the Ampersand 3-6
Combining Sequential and Concurrent Executions 3-7
Using Pipes: the Exclamation Mark 3-7
Raw Disk Input/Output 3-8
Command Grouping 3-9

Shell Procedure Files 3-10
Built-in Shell Commands and Options 3-11
Running Compiled Intermediate Code Programs 3-12

Chapter 4 Multiprogramming and Memory Management 4-1
Processor Time Allocation and Timeslicing 4-1
Process States 4-2
Creation of Processes 4-3
Basic Memory Management Functions 4-5
Loading Program Modules Into Memory 4-6
Deleting Modules From Memory 4-7
Loading Multiple Programs 4-8

Chapter 5 Useful System Information and Functions 5-1
File Managers, Device Drivers, and Descriptors 5-1
The Sys Directory 5-2
The Startup File 5-3
The CMDS Directory 5-3
Making New System Diskettes 5-3
Technical Information for the RS-232 Port 5-4

Chapter 6 System Command Descriptions 6-1
Organization of Entries 6-1
Command Syntax Notations 6-1
Command Summary 6-3

Chapter 7 Macro Text Editor 7-1
Overview 7-1
Text Buffers 7-1
Edit Pointers 7-1
Entering Commands 7-2
Control Keys 7-2
Command Parameters 7-3
Numeric Parameters 7-3
String Parameters 7-4
Syntax Notation 7-4
Getting Started 7-4
Edit Commands 7-6
Displaying Text 7-6
Manipulating the Edit Pointer 7-7
Inserting and Deleting Lines 7-10

Searching and Substituting 7-13
Miscellaneous Commands 7-14
Manipulating Multiple Buffers 7-17
Text File Operations 7-18
Conditionals and Command Series Repetition 7-21
Edit Macros 7-25
Macro Headers 7-25
Using Macros 7-26
Macro Commands 7-28
Sample Session 1 7-32
Sample Session 2 7-38
Sample Session 3 7-40
Sample Session 4 7-45
Sample Session 5 7-49
Edit Quick Reference Summary 7-55
Edit Commands 7-55
Pseudo Macros 7-57
Editor Error Messages 7-59

Appendices

A OS-9 Error Codes A-1
Device Driver Errors A-5

B Color Computer 2 Compatibility B-1
Alpha Mode Display B-3
Using Alpha Mode Controls with Windows B-3
Alpha Mode Command Codes B-4
Graphics Mode Display B-6
Graphics Mode Selection Codes B-6
Graphics Mode Control Commands B-7
Display Control Codes Summary B-9

C OS-9 Keyboard Codes C-1

D OS-9 Keyboard Control Functions D-1

Index

Introduction

Getting Started With OS-9 contains the information you must know to use the system. However, the handbook reveals only a small part of OS-9's capabilities. To learn about all of its features, you need to know more about how OS-9 works. This introduction provides such basic background information.

The Kernel

At the center of the OS-9 system is a module (program) called a *kernel*. (See the following illustration.) The kernel provides basic system services, such as multitasking and memory management. It links other system modules and serves as the system administrator, supervisor, and resource manager.

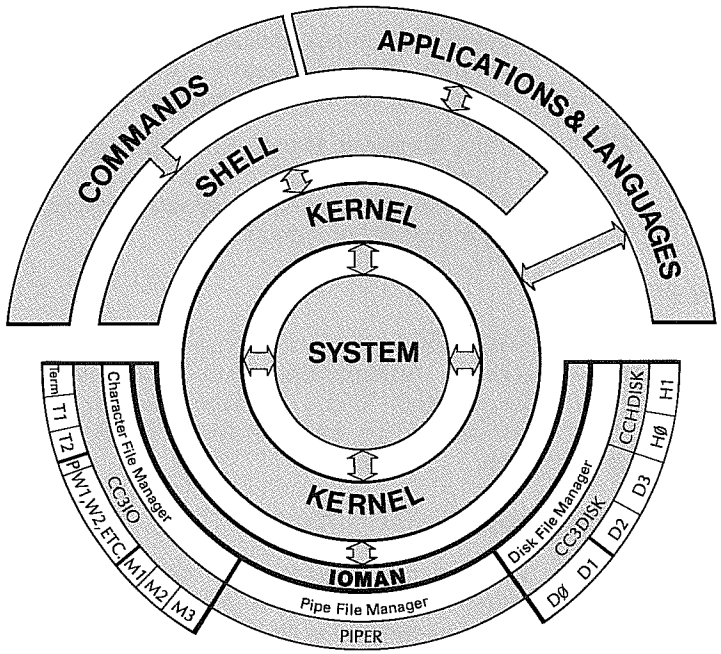


Figure 1
Term is your keyboard and video.
T1 and T2 are additional terminals.
P is a printer.
M1, M2, and M3 are modems.

The Input/Output Manager

Although the kernel manages OS-9, it does not directly process the input and output of data among the other modules and your computer *hardware* (printers, disk drives, terminals, and so on). Instead the kernel passes this responsibility to the input/output manager, IOMAN.

IOMAN has three *submanagers*: a character file manager, a pipe file manager, and a disk file manager. The responsibilities of these managers are as follows:

The Character File Manager — Handles the transfer of data between OS-9 and *character devices* (devices that operate on a character-by-character basis, such as terminals, printers, or modems). The sequential character file manager (SCF) can handle any number or type of such devices.

The Pipe File Manager — Handles communication between processes or tasks. Pipes let you use the output of one process as the input of another process.

The Disk File Manager — This Random Block File Manager (RBF) handles the transfer of data to and from *block-oriented, random access* devices, such as a disk drive system.

Device Drivers

CC3IO, PIPER, and CC3DISK are *device drivers*. These files contain code that transforms standard data into a form acceptable to a particular device, whether it is a terminal, printer, modem, disk drive, any other device, or another file. PIPES transfers data between processes.

Device Descriptors

Term, T1, P, M1, D0, and so on, are *device descriptors*. These files describe the devices connected to the system. They contain device initialization data as well as code that directs OS-9 to the physical addresses of the ports to which devices are connected.

The Shell

The kernel, in conjunction with IOMAN and its associated managers and modules, make up the OS-9 operating system. These modules handle all of the system's functions. However, OS-9 needs directions before it can accomplish useful tasks.

Directions to the system have two sources: commands and applications or computer language programs.

Before commands are useful to the kernel, the *shell* must interpret them. It analyzes commands and converts them into code that the kernel can understand.

Some application programs and computer languages also use the shell's functions. Others can access the kernel directly and do not need to go through the shell.

Going On

Chapters 2 through 5 contain detailed information on the operation of the OS-9 system illustrated in Figure 1. These chapters more fully describe the composition of files and directories. They tell about advanced features of commands and of the shell and contain information on multiprogramming and memory management.

Chapter 6 contains descriptions of the OS-9 commands. Chapter 7 tells you how to use OS-9's Macro Text Editor.

Chapter 2

The OS-9 File System

Input and output refer to the data your computer system receives and the data that it sends. OS-9 can receive (input) data from a keyboard, disk files, modems, and other terminals. It can send (output) data to all of these devices—except the keyboard—and to a video display.

OS-9 receives and sends data in the same format, regardless of whether the destination is a file or a device. It overcomes the differences in the devices by defining a standard for them and using *device drivers* to make each device conform to the standard. The result: a much simpler and more versatile input/output system.

Input/Output Paths

The base of OS-9's unified I/O system is an organization of *paths*. Input/output paths are, in effect, software links between files. (Remember, OS-9 thinks of all devices as files.)

Individual device drivers process data so that it conforms to the hardware requirements of the device involved. Data transfer is in streams of 8-*bit bytes* that can be either bidirectional (read/write) or unidirectional (read only or write only), depending on the device, how you establish the path, or both. A byte is a unit of computer data. (A byte may contain the code for one alphabet character.) A bit is a binary digit and has a value of either 0 or 1.

OS-9 does not require the data it manages to have any special format or meaning. The meaning of the data is determined by the programs that use it.

Some of the advantages of such a unified I/O system are:

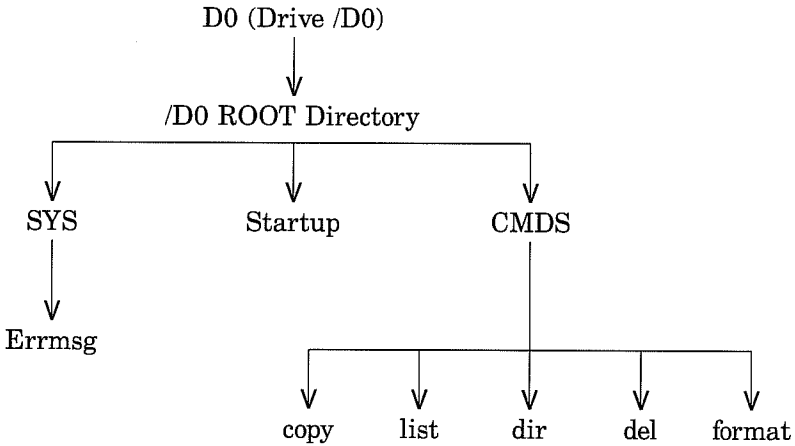
- Programs operate correctly regardless of the I/O devices selected.
- Programs are highly portable from one computer to another, even when the computers have different types of I/O devices.
- You can redirect I/O to alternate files or devices when you run a program, without having to alter the program.

- You can easily create and install new or special device driver routines.

Disk Directories

A directory is a storage place for other directories and files. It contains the information about the directories and files assigned to it so that OS-9 can easily find and access the data they contain.

Each disk has its own directory system. For example, a typical system diskette, diagrammed partially and simply, might look like this:



The *ROOT directory* of /D0—the *ROOT* from which the rest of the disk's file system *grows*—contains a file called Startup and two directories, SYS and CMDS.

SYS and CMDS, in turn, contain files: SYS contains Errmsg, and CMDS contains Copy, List, Dir, Del, and Format. All these files and directories, and many more, come built into the OS-9 system.

OS-9 organizes each directory area into 32-byte *records*. The first 29 bytes contain filename characters. The first byte of the name has its *sign bit* (the leftmost or most significant bit) set. When you delete a file, it is not immediately destroyed. Rather, the deletion process sets the first character position of the record to zero, and OS-9 no longer recognizes the record. Although the file contents still exist, they are no longer accessible to you or OS-9. Subsequent file creations overwrite deleted records.

The last three bytes of a record make up a 24-bit binary number that is the *logical sector number* pointing to the file descriptor record. Logical sectors are numbered with reference to the sequence of their use, rather than their physical location on a disk. See "Disk Files" for more information on disk organization.

You create directories using the MAKDIR command and can identify them by the D (directory) attribute. (See "Examining and Changing File Attributes".) MAKDIR initializes each directory with two entries having the names "." and "..". These entries contain the logical sector numbers of the directory and its parent directory, respectively.

You cannot use the COPY and LIST commands (as described in *Getting Started With OS-9*) with directories. Instead, use DSAVE and DIR.

You cannot delete directories directly. You must first empty a directory of files, convert it into a standard file, and then delete it. However, the DELDIR command performs all these functions automatically.

Subdirectories

A *subdirectory* is a directory residing in another directory. Actually, all directories you create are subdirectories, since all directories branch from the ROOT directory. However, because the system automatically creates the ROOT directory when you format a disk, this manual does not consider directories residing in the ROOT directory to be subdirectories.

Subdirectories can contain files and other subdirectories. In effect, OS-9 catalogues files and directories in much the same way that you might put files pertaining to a particular subject in a file cabinet drawer. With OS-9, you can have as many directory levels as your disk storage space permits.

Disk Files

A disk file is a *logical* block of data. (Logical means that although the data might not actually exist in a contiguous *block*, OS-9 treats it as though it does.) A file can contain a program, text, a command, a computer language, or any other form of code. Every time you ask OS-9 to store data on a disk, you must specify a *filename* for that block of data. Both you and the system must then use the filename to access the data.

The system stores all files as an ordered sequence of 8-bit bytes. The file can be any size from 0 bytes to the maximum capacity of the storage device and can be expanded or shortened as desired.

When OS-9 creates or opens a file, it establishes a *file pointer* for it. OS-9 addresses bytes within the file in the same manner it addresses memory, and the file pointer holds the *address* of the next byte to write or read. OS-9's *read* and *write* functions always update the pointer as the system transfers data.

This pointer function lets assembly-language programmers and high-level language programmers reposition the file pointer. To expand a file, write past the previous end of the file. Reading up to the last byte of a file causes the next read request to return an end-of-file status.

OS-9's file system also uses a universal organization for all I/O devices. This feature means that application programs can treat each hardware device similarly. The following section gives basic information about the physical file structure used by OS-9. (For more information, see the *OS-9 Level Two Technical Reference* manual.)

Sectors

The data contained in a file is stored in one or more *sectors* (disk storage units). These file sectors have both a *logical* and a *physical* arrangement. The logical arrangement numbers the sectors in sequence. The physical arrangement can be in any order based on the actual location of a sector on a disk's surface. For instance, Logical Sector 1 might be located at Physical Sector 10, and Logical Sector 2 might be located at Physical Sector 19.

Each sector contains 256 data bytes. The first sector of every file (Logical Sector Number 0 or LSN 0) is called the *file descriptor*. It contains the logical and physical description of the file. The disk driver module links sector numbers to physical track/sector numbers on a disk.

A sector is the smallest physical unit of a file that OS-9 can allocate for storage. On the Color Computer, a sector is also the smallest file unit. (To increase efficiency on some larger-capacity disk systems, OS-9 uses uniform-sized groups of sectors, called *clusters*, as the smallest allocatable unit. A cluster is always an integral power of two—2, 4, 8, and so on.)

OS-9 uses one or more sectors of each disk as a bitmap (usually starting at LSN 1) in which each data bit corresponds to one cluster on the disk. The system sets and clears bits to indicate which clusters it is using, which clusters are defective, and which clusters are free for allocation. The Color Computer default floppy disk system uses this format:

- Double-density recording on one side
- 35 tracks per diskette
- 18 sectors per track
- One sector per cluster

Each OS-9 file has a directory entry that includes the filename and the logical sector number of the file's *file descriptor sector*. The file descriptor sector contains a complete description of its file, including:

- Attributes
- Owner
- Date and time created
- Size
- Segment list (description of data sector blocks)

Unless the file size is 0, the file uses one or more sectors/clusters to store data. The system groups data sectors into one or more adjacent blocks called *segments*.

Text Files

Text files contain variable-length lines of ASCII characters. A carriage return (ASCII code 0D hexadecimal or 13 decimal) terminates each line. Text files contain such data as program source code, procedure files, messages, and documentation.

Programs usually read text files sequentially. Almost all high-level languages (such as BASIC09) support text files.

Use LIST to examine the content of text files.

Random-Access Data Files

Random-access files consist of sequences of records, with each record the same length. A program can find any record's beginning address by multiplying the record number by the number of bytes used for each record. This feature allows direct access of any record.

Usually, high-level languages let you subdivide records into fields. Each field can have a fixed length and use. For example, the first field of a record can be 25 text characters in length, the next field can be two bytes in length and used to hold 16-bit binary numbers, and so on.

OS-9 does not directly process records. It only provides the basic file functions used by high-level languages to create and handle random-access files.

Programmers use high-level languages like BASIC09, Pascal, and C to create random-access data files. For instance, in BASIC09 and Pascal, GET, PUT, and SEEK functions operate on random-access files.

Procedure Files

Procedure files are disk files that contain commands. You can use them to execute a series of commands by typing and entering a single command name.

Your System Master diskette contains one procedure file named Startup. You can create your own procedure files using the BUILD command, copying input from the keyboard to a file, or by using a text editor program. For instance, suppose you have three disk drives, /D0, /D1, and /H0. You could create three very simple procedures to allow you to look at the directories of these disks by typing and entering a simple two-character command.

To create a procedure file to look at the directory of /D1, type:

```
build p1 [ENTER]
display 0C [ENTER]
dir /d1 [ENTER]
display 0A [ENTER]
[ENTER]
```

The first line creates a file named P1 (print directory for Drive /D1). When you press [ENTER], a question mark appears on the screen telling you that OS-9 is waiting for input. Type the rest of the lines. Finally, press [ENTER] at the beginning of a line to tell OS-9 that the input is complete. OS-9 closes the file.

Now, to see the contents of Drive /D1, type `p1` [ENTER]. The command `display 0C` clears the video screen. The command `display 0A` causes the cursor to drop down one line on the screen.

Use your imagination. Almost anything you can do from the keyboard, you can do with a procedure file. However, remember that OS-9 looks only in the current data directory for a procedure file, unless you provide a full pathlist to the procedure. Also, OS-9 must be able to find any command in the current execution directory that is part of a procedure file. If the current execution directory does not contain the commands you need, change it, either from the keyboard or as part of your procedure file.

Executable Program Module Files

OS-9 *program modules* are executable program code, generated by an assembler or compiled by a high-level language. A file can contain one or more program modules.

Each module has a standard format that includes the *object code* (the executable portion of the module), a *module header* that describes the type and size of the module, and a CRC (Cyclic Redundancy Checksum) value. The system stores program modules in files in the same structure in which they load into memory. Because OS-9 programs are position-independent, they do not require specific memory addresses for loading.

For OS-9 to load program module(s) from a file, the file execute attribute must be set, and each module must have a valid module header and CRC value. If you or the system alters a program module in any way (either as a file or in memory), its CRC check value becomes incorrect, and OS-9 cannot load the module.

If a file contains two or more modules, OS-9 treats them as a *group* and assigns contiguous memory locations for them.

Using LIST on program files or any other files that contain binary data, causes a jumbled display of random characters and an error message.

Miscellaneous File Use

OS-9's basic file functions are so versatile that you can devise almost unlimited numbers of special-purpose file formats for particular applications that require formats not discussed here (text, random-access, and so on).

The File Security System

Each file and directory has properties called *ownership* and *attributes* that determine who can access the file and how they can use it.

OS-9 automatically stores the user number associated with the creation of a file. The system considers the *owner* of the number to be the owner of the file.

Security functions are based on access attributes. There are eight attributes, each of which can be turned *off* or *on* independently. When the D (directory) attribute is on for a file, that file is a directory. (Only MAKDIR can set the D attribute for a file.) When the S (single-user) attribute is on, only one program or user can access the file at a time.

The other six attributes control whether the file can be read from, written to, or executed by either the owner or the *public* (all other users.) When on, these six attributes function as follows:

Owner read permission — The owner can read from the file. Use this permission to prevent *binary* files from being used as *text* files.

Owner write permission — The owner can write to the file or delete it. Use this permission to protect important files from accidental deletion or modification.

Owner execute permission — The owner can load the file into memory and execute it. To be loaded, the file must contain one or more valid OS-9 memory modules.

Public read permission — Anyone can read and copy the file.

Public write permission — Anyone can write to or delete the file.

Public execute permission — Anyone can execute the file.

For example, if a file has all permissions on except *write permit to public* and *read permit to public*, the owner has unrestricted access to the file. Other users can execute it but cannot read, copy, delete, or alter it.

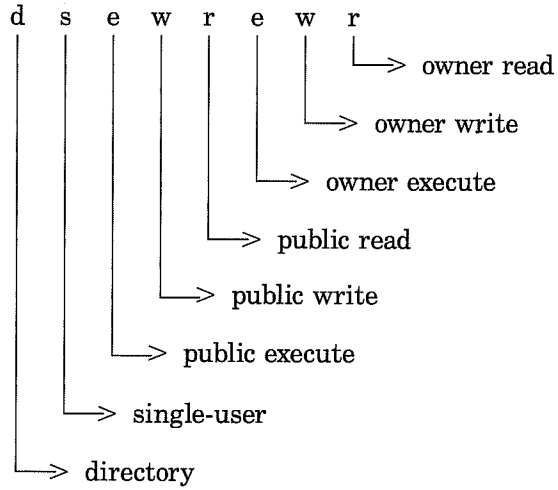
Examining and Changing File Attributes

You can use the DIR command, with the E (entire) option, to examine the security permissions of all files in a particular directory. An example of output using DIR E on the current data directory is:

```
Directory of . 10:20:44

Owner Last modified Attributes Sector Bytecount Name
----- ------------- ---------- ------ --------- ---------
  0   86/07/31 1436  ----r-wr       A      6567 OS9Boot
  0   86/07/31 1437  d-ewrewr      71       560 CMDS
  0   86/07/31 1442  d-ewrewr     1B8        80 SYS
  0   86/07/31 1409  ------wr     1Bd        55 startup
```

The *Attributes* column shows which attributes are on by listing one or more of the following codes.



For example, the first file shows:

```
----r-wr
```

This means that (1) The file is not a directory. (2) It is shareable. (3) The public cannot execute it or (4) write to it, but can (5) read it. (6) The owner cannot execute the file, but can (7) write to it, and (8) can read it.

To examine the attributes of a particular file, use ATTR. Typing ATTR followed by a filename shows you the file's current attributes, for example:

```
attr file2 ENTER
```

A possible screen display is:

```
---wr-wr
```

To change a file's attributes use ATTR and a filename, followed by a list of one or more attribute abbreviations. However, you must own a file before you can change its attributes.

The following command enables public write and public read permissions and removes the execute permission for both the owner and the public:

```
attr file2 pw pr -e -pe ENTER
```

Note: In order to protect data stored in directories, the D attribute behaves somewhat differently from the other attributes. You cannot use ATTR to turn on the D attribute—only MAKDIR can do that—and you can use ATTR to turn off D only if the directory is empty.

Record Lockout

When two or more processes use the same file simultaneously, they might attempt to update the file at the same time, causing unpredictable results. When you open a file in the *update* mode, OS-9 eliminates the problem of simultaneous use by *locking* the sections of the file. The lock covers any disk sectors containing the bytes last read by each process accessing the file. If one process attempts to access a locked portion of a file, OS-9 puts the process to sleep until the locked area is free.

OS-9 moves the lock and frees the area when it reads from or writes to another area. The system removes a lock on a file when the process associated with the lock closes its path to the file. A process can have only one lock on a file, but it can have locks on more than one file.

You can lock an entire file by activating its single user bit. (See the earlier section "Examining and Changing File Attributes.") When the single user bit is on, only one process can open a path to the file at a time. Attempts by other processes to access the file result in an error.

Device Names

Each physical I/O device supported by OS-9 has a unique name. The following list describes some of the device names supported by the system. Your system diskette already contains several of these devices. You can define others to use with CONFIG.

D0_35S	Floppy Disk Drive /D0, single sided, 35 cylinders.
D1_35S	Floppy Disk Drive /D1, single sided, 35 cylinders.
D2_35S	Floppy Disk Drive /D2, single sided, 35 cylinders.
D3_35S	Floppy Disk Drive /D3, single sided, 35 cylinders.
DDD0_35S	Default Disk Drive /D0, single sided, 35 cylinders.
D0_40D	Floppy Disk Drive /D0, double sided, 40 cylinders.
D1_40D	Floppy Disk Drive /D1, double sided, 40 cylinders.
D2_40D	Floppy Disk Drive /D2, double sided, 40 cylinders.
DDD0_40D	Default Disk Drive /D0, double sided, 40 cylinders.
D1_80D	Floppy Disk Drive /D1, double sided, 80 cylinders.
D2_80D	Floppy Disk Drive /D2, double sided, 80 cylinders.
P	a printer using the RS-232 serial port
TERM	your computer keyboard and video display
T1	a terminal port using the standard RS-232 port
T2	a terminal using the optional RS-232 communications pak
T3	a terminal using the optional RS-232 communications pak
M1	a modem using an optional 300 baud modem pak
M2	a modem using an optional 300 baud modem pak
W	a generic window descriptor
W1	window device Number 1

W2	window device Number 2
W3	window device Number 3
W4	window device Number 4
W5	window device Number 5
W6	window device Number 6
W7	window device Number 7

Although OS-9 and your computer can access all these devices, your original diskette does not configure it to do so. For information on configuring your system, see Chapter 7 of *Getting Started With OS-9*.

Because device names are at the root of the file system, you can use them only as the first part of a pathlist. Always precede the name of a device with a slash.

When you refer to a non-disk device, for example a terminal or printer, use only the device name. /P, for instance, is the full allowable pathlist for a printer.

Note: An I/O device name is actually the name of an OS-9 device descriptor that you precede with a slash (/). OS-9 automatically loads device descriptors during the OS-9 boot sequence. You can add or delete other device descriptors while the system is running or add device descriptors to the bootfile using CONFIG.

Chapter 3

Advanced Features of the Shell

This chapter discusses the advanced capabilities of the shell. In addition to basic command line processing, the shell facilitates:

- Input/output redirection, including filters
- Memory allocation
- Multitasking (concurrent execution)
- Procedure file execution
- Built-in commands

You can use these advanced capabilities in many combinations. Following are several examples. Study the basic rules, use your imagination, and explore.

More About Command Line Processing

The shell is a program that reads and processes command lines, one at a time, from the computer's input device (usually your keyboard). It *parses* (scans) each line to identify and process any of the following parts that might be present:

- A program, procedure file, or built-in command
- Parameters to be passed to the program
- Execution modifiers to be processed by the shell

For some commands, only the *keyword* (the program, procedure file, or command name) must be present. Other commands have required or optional parameters. As well, a command line can include *modifiers* that influence the operation of the command. OS-9 features that affect command execution are:

Execution Modifiers Let you increase the amount of random access memory (RAM) available for a command. Also lets you redirect input to a process, output from a process, or both.

Command Separators Let you enter more than one command on a line, perform concurrent execution of commands, or connect the input or output of one command to another command.

Command Grouping Lets you group all the commands that you want affected by command modifiers or separators.

Note: The next section, "Command Modifiers," provides details on these features.

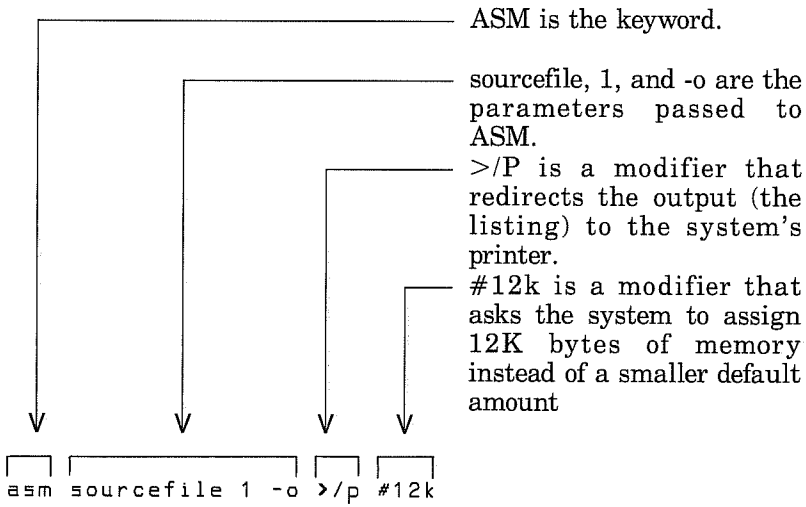
Once the shell identifies the keyword, it processes any modifiers. It then assumes the remaining text consists of parameters, which it passes to the program being called.

When the shell receives a built-in command, it immediately executes it. If it receives a command that is not built in, it searches for the appropriate program and then runs it as a new process. The keyword must be the first entry in any line.

While the program is running, the shell deactivates itself. At the termination of the program, the shell reactivates and accepts the next input. This cycle continues until the shell detects an end-of-file in the input path. It then terminates its own execution. You can input an end of file from the keyboard by pressing SHIFT BREAK.

Following is a sample shell command line that calls the assembler.

In this example:



Command Modifiers

Add command modifiers to a command line to change the way in which the command functions. Modifiers let you tailor OS-9 commands to your specifications. Type them in a command line after the keyword and either before or after other parameters you might be using.

The shell processes command modifiers before it executes a program. If it detects an error in any of the modifiers, it stops execution and reports the error.

The shell strips command modifiers from the part(s) of the command line passed to the program as parameters. Therefore, you cannot use the characters reserved as modifiers (# ; ! < > &) inside other parameters.

Execution Modifiers

Execution modifiers alter the amount of memory commands have available, or they redirect command input or output.

Alternate Memory Size Modifier. When the shell invokes a command program, it allocates the minimum amount of working RAM (random access memory) specified in the program's module header.

> **Note:** All executable programs include a module header which holds the program's name, size, memory requirements, and other information. For information on viewing the contents of a module header, see the IDENT command.

You might want to increase a command's default memory size. You can assign memory either in 256-byte pages or in 1024-byte increments. To add memory in pages, use the modifier *#n*, where *n* is the number of pages. To add memory in 1024-byte increments, use the modifier *#n*K, where *n* is the number of 1024-byte increments.

The following two examples have identical results:

```
copy #8 file1 file2 [ENTER]     (8 x 256  = 2048 bytes)
copy #2K file1 file2 [ENTER]    (2 x 1024 = 2048 bytes)
```

I/O Redirection Modifiers. Input/output redirection modifiers reroute a program's I/O from the standard path to alternate files or devices.

One of OS-9's advantages is that its programs use standard I/O paths rather than individual, specific file, or device names. You can easily redirect the I/O to any file or device without altering the program itself.

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

- **Standard input path**—Routes data from the terminal's keyboard to programs. The standard input path is Path Number 0.

  Use the less-than symbol (<) to redirect data to this path.

- **Standard output path**—Routes data from programs to the terminal's display. The standard output path is Path Number 1.

  Use the greater-than symbol (>) to redirect data from this path.

- **Standard error output path**—Routes routine status messages (prompts and errors) to the terminal's display. (The name *error output path* is somewhat misleading, since many kinds of messages in addition to error messages travel the path.) The standard error path is Path Number 2.

  Use two greater-than symbols (>>) to redirect data from this path.

When you use a redirection modifier in a command line, follow it immediately with a pathlist for the substitute device. For example, you can use LIST to redirect the contents of a file called Correspondence from the terminal to the printer, by typing:

```
list correspondence >/p ENTER
```

The shell automatically creates, opens, and closes files referenced by redirection modifiers as needed. **The system immediately restores normal I/O paths at the completion of any command using redirection modifiers.**

In the next example, the shell redirects DIR's output—a list of files in the MEMOS directory—to the file /D1/Savelisting:

```
dir /d0/memos >/d1/savelisting ENTER
```

You can now view the contents of Savelisting by typing:

```
list /d1/savelisting ENTER
```

OS-9 displays the file contents in a format similar to a directory listing.

```
 Directory of /d0/memos
CMDS        SYS         startup
OS9Boot
```

You can use one or more redirection modifiers before the program's parameters, after the program's parameters, or both. However, use each modifier only once in a command.

The following example shows how you can use all of the redirection modifiers together to start BASIC09 on a device window and redirect all input and output to it:

```
basic09 <>>>/w1 ENTER
```

When you redirect multiple paths, you must use the redirection symbols in the proper order as shown here:

Legal	Illegal
<> /w1	>< /w1
<>> /w1	>>< /w1
>>> /w1	>>< /w1

Command Separators. You can include more than one command on a command line by using command separators. Command separators cause multiple commands to execute either sequentially or concurrently, depending on the separator you use.

Sequential execution means that one program must complete its function and terminate before the shell lets the next program execute. *Concurrent execution* means that two or more programs begin execution and run simultaneously.

Sequential Execution Using the Semicolon. Using a semicolon between commands on one line causes them to execute sequentially. For instance:

```
copy myfile /d1/newfile; dir >/p ENTER
```

This command causes the shell to: (1) execute the COPY command, (2) enter a *waiting* state until COPY terminates, then awake, and (3) execute DIR.

If an error occurs in any program, the shell does not execute subsequent commands, regardless of the state of the SHELL command's X (stop on error) option.

Here are two more examples of commands using the semicolon:

```
copy oldfile newfile; del oldfile; list newfile
ENTER
```

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

Commands separated by semicolons are in fact separate and equal *child* processes of the shell.

> **Note:** When one process creates another process, OS-9 calls the creator the *parent process* and the created process the *child process*. The child can become a parent by creating yet another process.

Concurrent Execution Using the Ampersand. You can use the ampersand (&) to cause multiple commands to run at the same time. Each command you specify runs as a separate *child* process of the shell. That is, for each process, the shell creates a separate shell to handle the operation. When the process is complete, the child shell terminates, or *dies*.

While more than one process is running, OS-9 divides execution time equally among the processes.

The number of programs that can run at the same time varies. It depends on the amount of free memory in the system and the memory requirements of the programs being executed.

An example of a simple command line using the & separator is:

```
dir >/p& ENTER
```

The shell begins to run DIR, sending output to the printer. At the same time it displays both the number of the *forked* process (DIR) and a new prompt, like this:

```
&007
OS9:
```

To fork a process means to create a process as a branch of another process—a subroutine.

After using the ampersand to fork a background process, you can then enter another command, which the shell executes while output from your original command continues to go to the printer. This means you don't waste time waiting for OS-9 to finish a task. At times, the keyboard might not seem to respond to your typing, because characters do not appear on the screen. However, OS-9 stores the characters in the keyboard buffer and displays them as soon as the shell can accept input again.

If you have several processes running simultaneously and want information about them, use the PROCS command.

Combining Sequential and Concurrent Executions. You can, if you want, use both the concurrent and sequential command separators in one command line. For example:

```
dir >/p& list file1& copy file1 file2; del temp
ENTER
```

Because the & modifier joins the DIR, LIST, and COPY commands, these commands run concurrently. But, because a semicolon precedes the DEL command, DEL does not run until the other commands terminate.

Using Pipes: The Exclamation Mark. You can use the exclamation mark (!) to construct *pipelines* for OS-9 commands. Pipelines consist of two or more concurrently executing programs with standard input paths, and standard output paths or both, connected to each other with *pipes*.

Pipes are the primary means of transferring data from process to process. They are vital to interprocess communications. Pipes are first-in, first-out buffers, or *holding areas* for data.

The shell automatically buffers and synchronizes I/O transfers using pipes. A single pipe can direct data to several destinations or *readers*, and can receive data from several sources, or *writers* on a first-come, first-serve basis. An end-of-file occurs if a program attempts to read from a pipe when writers are not available to send data. Conversely, a write error occurs if a program attempts to write to a pipe when readers are not available.

Pipelines are created by the shell when it processes an input line with one or more pipe separators (!). For each pipe separator, the shell directs the standard output of the program on the left of the pipe separator to the standard input of the program on the right of the separator. The shell creates an individual pipe for each pipe separator in the command line. For example:

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

This command redirects input from a path called Master_file to a file named Update. The output of Update becomes the input for the program Sort. The output of Sort, in turn, becomes the input for the program Write_report. Finally, the command redirects output from Write_report to the printer. The shell executes all programs in a pipeline concurrently. The pipes synchronize the programs so the output of one never *gets ahead* of the input request of the next program. This synchronization means that data cannot flow through a pipeline any faster than the slowest program can process it.

Raw Disk Input/Output. OS-9 has a special pathlist function to perform *raw* physical input/output operations on a disk. The pathlist consists of the device name, immediately followed by the "@" character.

This command causes OS-9 to treat the entire diskette in Drive /D0 as one logical file. The operation reads each byte of each sector, without regard to actual file structure. Commands such as DIR, ATTR, and MFREE use this feature to access sectors of disks that are not part of file data areas, such as header sectors.

> **Warning:** When using this raw access, use extreme caution. Because you can write on any sector, you can easily damage file or directory structures and lose data. Using @ defeats any file security and record locking systems.

To convert a byte address to a logical sector number when using @, multiply the sector number by 256. Conversely, the logical sector number of a byte address is the byte address, modulo 256.

Command Grouping

You can enclose sections of command lines in parentheses to permit modifiers and separators to affect an entire set of programs. The shell processes the material in the parentheses by recursively calling itself to execute the enclosed command list.

For example, if you want to send directory listings of the ROOT directory of Drive /D0 and then the ROOT directory of Drive /D1 to the printer, you can type either:

```
dir /d0 >/p; dir /d1 >/p [ENTER]
```

or:

```
(dir /d0; dir /d1) >/p [ENTER]
```

The results are identical except that the system *keeps* the printer continuously in the second example. In the first example, another user could *steal* the printer between DIR commands.

You can group commands to cause programs to execute both sequentially and concurrently with respect to the shell that initiated them. For instance:

```
(del file1; del file2; del file3)& [ENTER]
```

Here, the shell does the overall deleting process concurrently with whatever else you tell it to do, because you're using &. However, the shell deletes the three specified files sequentially because you're using semicolons within the parentheses.

Suppose you have a program named Makeuppercase that converts lowercase characters to uppercase and a program named Transmit that sends data to another computer, you can use a command line like this:

```
(dir cmds; dir sys) ! makeuppercase ! transmit
[ENTER]
```

The shell processes the output of the first DIR command and then the second. It sends all the DIR output to Makeuppercase, and Transmit sends all the output to another computer.

Shell Procedure Files

The shell is a *re-entrant program*. This means it can be simultaneously executed by more than one process. Like most other OS-9 programs, the shell uses standard I/O paths for routine input and output.

OS-9's shell offers you a special feature, a time and effort saver called a *procedure file*. A procedure file is a related group of commands, and when you run the file, you execute all the commands.

Other names for procedure file processing are *batch* and *background* processing. A procedure file becomes new input for the shell. By running a procedure file, you're using the shell to create a new shell, a *subshell* that accepts and carries out the commands in the procedure file.

> **Note:** If you plan to use the same command sequences repeatedly, create a procedure file to do the job by using BUILD.

When you enter any command line, the shell looks for the specified program in memory or in the execution directory. If it cannot find the program there, it searches the data directory for a file with the specified name. If it finds the file, the shell automatically interprets it as a procedure file, and creates the subshell, which executes the commands listed in the procedure file.

Procedure files can also let the computer execute a lengthy series of programs while it is unattended, or even while it is running other programs.

There are two ways to run a procedure file. For instance, to execute a procedure file called Mailsequence, type either:

```
shell mailsequence [ENTER]
```

or

```
mailsequence [ENTER]
```

Both commands do the same thing: create a subshell to run the commands you've built into your Mailsequence procedure file.

To run a procedure file in a concurrent mode, use the ampersand (&) modifier. As long as memory is available, you can run any number of files concurrently.

You can even build procedure files to sequentially or concurrently execute other procedure files.

> **Note:** If you use procedure files to run programs you don't intend to monitor closely, you can redirect standard output and standard error output to another file. Later you can review the file's contents. Output redirection eliminates the annoying output of shell messages on your terminal at random times.

Built-in Shell Commands and Options

The shell has a number of built-in commands and options. Whenever you use one of these functions, the shell executes it without loading it or creating a new process to execute it.

You can execute built-in functions alone, use them at the beginning of a command line, or use them following any program separator. You can separate adjacent built-in commands by spaces or commas.

The built-in commands and their functions are:

CHD *pathlist* Changes the data directory to the directory specified by the pathlist.

CHX *pathlist* Changes the execution directory to the directory specified by the pathlist.

EX *modname* Directly executes the module named. This function deletes the shell process so that it ceases to exist and executes the new module in its place. Use EX to replace the executing shell with the program specified by *modname*. You can also use EX without a module name to eliminate the current shell, for example, a shell you initialized in a window (see below).

i=*devname* Makes a shell an immortal shell. This means that when the shell ends, due to an EOF, OS-9 restarts it. Each time the shell restarts, it has the same data and execution directories. To kill an immortal shell, use EX to "chain" to a null process, such as:

```
ex ENTER
```

w	Waits for any process to terminate.
* ***text***	Allows you to make a *comment*. The shell does not process text following the asterisk. Use this function to label operations in a procedure file.
kill *procID*	Stops the specified process.
setpr *procID number*	Changes the specified process's priority number.
x	Causes the shell to cease operation whenever an error occurs (a system default).
-x	Causes the shell to continue operation when an error occurs. Use this function in procedure files to enable the shell to continue to other commands if one command process fails because of a system error.
p	Turns the shell prompt and messages on (a system default).
-p	Inhibits the shell prompt and messages. Use this option in procedure files to disable screen display. Be sure to turn the prompt and message function back on afterward.
t	Makes the shell copy all input lines to output. Use this function with a procedure file to cause command lines to display as they execute.
-t	Sets the system so that it does not copy input lines to output (a system default).

Running Compiled Intermediate Code Programs

Before the shell executes a program, it checks the program module's language type. If it is not 6809 machine language, the shell calls the appropriate run-time system for that module.

For instance, if you have BASIC09 on your OS-9 system and want to run a BASIC09 I-code module called Adventure, you can type:

```
basic09 adventure ENTER
```

or:

```
adventure ENTER
```

or:

```
runb adventure ENTER
```

In the last example, the shell uses the RUNB module to interpret the Adventure I-code module.

Chapter 4

Multiprogramming and Memory Management

One of OS-9's most valuable capabilities is *multiprogramming*—sometimes called timesharing or multitasking. This feature lets your computer run more than one process at the same time. Multiprogramming can be a time saving advantage in many situations. For example, you can edit one program while the system prints another. Or you can use your Color Computer to control a household alarm system, lighting, and heating and at the same time use it for routine work or entertainment.

OS-9 uses multiprogramming regularly for internal functions. You can use it by putting an ampersand at the end of a command line. Doing this causes the shell to run your command as a *background*, or concurrent, task.

To run several processes simultaneously, OS-9 must coordinate its input/output system and CPU time and allocate its memory as needed. This chapter gives you some basic information about how OS-9 manages its resources to optimize system efficiency and make efficient multiprogramming a reality.

Processor Time Allocation and Timeslicing

CPU time is the most precious resource of a computer. If the CPU is busy with one task it cannot perform another. For example, many processes must wait for you to enter information from the terminal. While the process is waiting, your computer's CPU must also wait. Your computer is limited by your typing speed.

On many systems there is no way around such a bottle neck. However, OS-9 is more efficient. It assigns CPU time to processes only as they need it.

To do this, OS-9 uses *timeslicing*. Timeslicing, as described in the following paragraphs, lets all active processes share CPU time.

A real-time clock interrupts the Color Computer's CPU 60 times each second. The interruption points are called *ticks*, and the spaces between ticks are called timeslices.

OS-9 allocates timeslices to each process. At any tick it can suspend execution of one process and begin execution of another. This starting and stopping of processes does not affect their execution.

How often OS-9 gives a process timeslices depends on the process's priority relative to the priority of other active processes. You can access priority using a decimal number from 0 through 255, where 255 is the highest priority.

OS-9 automatically gives the shell a priority of 128. Because child processes inherit their parents' priorities, the shell's child processes also have priorities of 128. You can find a process's priority with the PROCS command, and can change it using the SETPR command.

You cannot compute the exact percentage of CPU time assigned to any particular process, because there are some dynamic variables involved, such as the time the process spends waiting for I/O devices. But you can approximate the percentage by dividing the process's priority by the sum of the priority of all active processes:

$$\text{process's CPU share} = \frac{\text{priority of the process}}{\text{sum of the priorities of all active processes.}}$$

Note: Timeslicing happens so quickly that it looks as if all processes execute simultaneously and continuously. If, however, the computer becomes overloaded with processing, you might notice a delay in response to input from the terminal. Or, you might notice that a procedure program takes longer than usual to run.

Process States

The CPU time allocation system automatically assigns each process one of three *states* that describes its current status. Process states are important for coordinating process execution. A process can have only one state at any instant, although state changes can be frequent. The states are:

- **Active**—Applies to processes currently able to work—that is, those not waiting for input or for anything else. These are the only processes assigned CPU time.

- **Waiting**—Applies to processes that the system suspends until another process terminates. This state allows coordination of sequential process execution. The shell, for example, is in the waiting state during the execution of a command it has initiated.
- **Sleeping**—Applies to a process suspending itself for a specified time, or until receipt of a *signal.* (Signals are internal messages that coordinate concurrent processes.) This is the typical state of processes waiting for input/output operations.

  The shell does not assign CPU time to sleeping or waiting processes. It waits until they become active. The PROCS command gives information about process states.

Creation of Processes

If a parent process creates more than one child process, the children are called *siblings* with respect to each other. If you examine the parent/child relationship of all processes in the system, a hierarchical lineage becomes evident. In fact, this hierarchy resembles a family tree. (The *family* concept makes it easy to describe relationships between processes.) OS-9 literature uses the family concept extensively in describing OS-9's multiprogramming functions.)

OS-9's *fork* function automatically performs the sequence of operations required to create a new process and initially allocate resources to it.

If for any reason, fork cannot perform any part of the sequence, the system stops and fork sends its parent an error code. The most frequent reason for failure is the unavailability of required resources (especially memory), or the inability of the system to find the specified program.

A process can create many processes, subject only to the availability of unassigned memory. When the parent issues a fork request to OS-9, it must specify certain information:

- **A primary module**—The name of the program to be executed by the new process. The program can already be present in memory, or OS-9 can load it from a disk file with the same name.

- **Parameters**—Data to be passed to and used by the new process. OS-9 copies this data to part of the child process's memory area. (Parameters frequently pass filenames, initialization values, and other information.)

The new process *inherits* some of its parent's properties, including:

- **A user number**—For use by the file security system to identify all processes belonging to a specific user. (This is not the same as the *process ID*, which identifies a process.) OS-9 obtains this number from the system password file when a user logs on. The system manager is always User 0.
- **Standard input and output paths**—The three paths: input, output, and error, used for routine input and output. Most paths can be shared simultaneously by two or more processes.
- **Current directories**—The data directory and the execution directory.
- **Process priority.**

As part of the fork operation, OS-9 automatically assigns:

- A process ID, a number in the range 1 to 255 that identifies the process. Each process has a unique number.
- Enough memory to support the new process. In OS-9, all processes share a memory address. OS-9 allocates a data area for the process's parameters and variables and a stack for each process's use. It needs a second memory area in which to load the process if it does not reside in memory.

In summary, each new process has:

- A primary module
- Parameters
- A user number
- Standard I/O paths
- Current directories
- A priority
- An ID number
- Memory

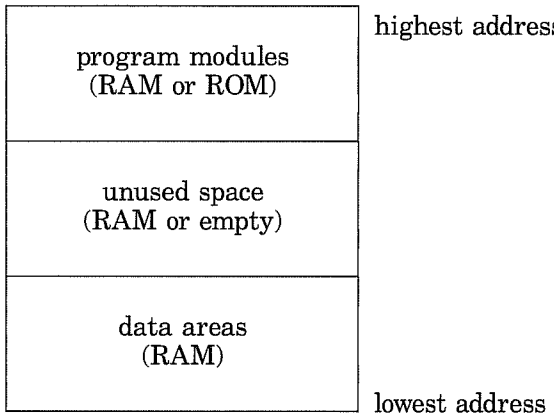
Basic Memory Management Functions

Memory management is an important OS-9 function. OS-9 automatically allocates all system memory to itself and to processes, and also keeps track of the logical contents of memory (the program modules that are resident in memory at any given time). The result is that you seldom need to bother with the actual memory addresses of programs or data.

The operating system and each process have individual address spaces. Each address space has the potential to contain up to 64 kilobytes of RAM memory. Using memory management unit (MMU) hardware, OS-9 moves memory into and out of each address space as required for system operations.

Although each unit is subject to the 64K maximum program size, you can run several processes simultaneously and utilize more than 64K overall. The system logically assigns RAM memory in 256-byte pages, but the MMU's hardware block size is 8K. Each of these physical blocks has an extended address that is called a *block number*. For example, the 8K physical block residing at address $3C000 to $3DFFF is Block Number $3C.

Within an address space, OS-9 assigns memory from higher addresses downward for program modules and from lower addresses upward for data areas. The following chart shows this organization:



Loading Program Modules into Memory

When performing a fork operation, OS-9 first attempts to locate the requested program module by searching the *module directory*, which has the address of every module present in memory. The 6809 instruction set supports a type of program called *re-entrant code*, which means that processes can share the code of a program simultaneously.

Since almost all OS-9 family software is re-entrant, the system can make the most efficient use of memory. For example, suppose that OS-9 receives a request (from a process) to run BASIC09 (which requires 22 kilobytes of memory), but has already loaded it into memory for another process. Because the software is re-entrant, OS-9 does not have to load it again and use another 22K of memory. Instead the new process shares the original BASIC09 by including the location of the BASIC09 module in its memory map.

OS-9 automatically keeps track of how many processes are using each program module, and deletes the module when all processes using it terminate.

If the requested program does not yet reside in memory, OS-9 uses its name as a pathlist (filename) and attempts to load the program from disk.

Every program module has a module header describing the program and its memory requirements. OS-9 uses the header to determine how much memory the process needs for variable storage. The module header includes other information about its program, and is an essential part of the OS-9 machine language operation.

You can also place commands or programs into memory using the LOAD command. Doing so makes the program available to OS-9 at any time, without having to be loaded from disk. A program is physically loaded into memory only if it does not already reside there.

LOAD causes OS-9 to copy the requested module from a file into memory, verifying the CRC (Cyclic Redundancy Check). If the module is not already in the module directory, OS-9 adds it.

If the program module is already in memory, the load process still begins in the same way. But, when OS-9 attempts to add the module to the module directory and notices that the module is already there, it merely increments the known module's *link count* (the number of processes using the module).

When OS-9 loads multiple modules in a single file, it associates them logically in the memory management system. You cannot deallocate any of the group modules until all modules have zero link counts. Similarly, linking to one module within a group causes all other modules in the group to be mapped into the process's address space.

Deleting Modules From Memory

UNLINK is the opposite of LOAD. It decreases a program module's link count by one. When the count becomes zero (presuming that the module is no longer used by any process), OS-9 deletes the module, deallocates its memory, and removes its name from the module directory.

> **Warning:** Never use the UNLINK command on a program or a module not previously installed using LOAD. Unlinking a module you did not LOAD (or LINK) might permanently delete it when the program terminates. The shell automatically unlinks programs loaded by fork.

Suppose you plan to use the COPY command ten times in a row. Normally, the shell must load COPY each time you enter the command. But if you load the COPY module into memory and then enter your string of commands, you don't have to wait for the system to load and unload COPY repeatedly. When you finish using COPY, use UNLINK to unlock the module from memory. The sequence looks like this:

```
load copy [ENTER]
copy file1 file1a [ENTER]
copy file2 file2a [ENTER]
copy file3 file3a [ENTER]
copy file4 file4a [ENTER]
copy file5 file5a [ENTER]
copy file6 file6a [ENTER]
copy file7 file7a [ENTER]
copy file8 file8a [ENTER]
copy file9 file9a [ENTER]
copy file10 file10a [ENTER]
unlink copy [ENTER]
```

It is important to use UNLINK when you no longer need the program. Otherwise, the program continues to occupy memory that might be used for other purposes.

> **Warning:** Be careful not to unlink modules that are in use, because OS-9 deallocates the memory used by the module and destroys its contents. All programs using the unlinked module crash.

Loading Multiple Programs

Because all OS-9 program modules are *position-independent*, you can have more than one program in memory at the same time. Since position-independent code (PIC) programs don't have to be loaded into specific, predetermined memory addresses to work correctly, you can load them at different memory addresses at different times.

PIC programs require special types of machine language instructions that few computers have. The ability of the 6809 microprocessor to use PIC programs is a powerful feature and one of the greatest aids toward multiprogramming. You can load any number of program modules until available system memory is full.

OS-9 automatically loads each program module at non-overlapping addresses. (Most operating systems write over the previous program's memory when loading a new program.) OS-9's technique means that you do not need to be concerned with absolute memory addresses.

Chapter 5

Useful System Information and Functions

The OS-9 system must load many parts of the operating system during startup and system operation. Therefore, on a floppy disk system, you must keep the system diskette in Drive /D0.

Two files used during the system startup operation, OS9Boot and Startup, must remain in the system diskette's ROOT directory. Other files on the system diskette are organized into two directories: CMDS (commands) and SYS (other system files). You can also create other files and directories on the system diskette. OS-9 always creates the initial data directory, or ROOT directory, when you format a diskette.

File Managers, Device Drivers, and Descriptors

The *bootstrap* (instructions that initialize OS-9) loads a file called OS9Boot into RAM memory at startup. This file contains file managers, device drivers and descriptors, and any other modules that permanently reside in memory. For instance, the OS9Boot file might contain these modules:

OS9p2	OS-9 Kernel
INIT	System Initialization Table
IOMan	OS-9 input/output manager
RBF	Random block (disk) file manager
SCF	Sequential character (terminal) file manager
PipeMan	Pipeline file manager
Piper	Pipeline driver
Pipe	Pipeline device descriptor
CC3IO	Keyboard/video graphics device driver
VDGINT	32x16 screen subroutines
GRFINT	Windowing subroutines
PRINTER	Printer device driver
SIO	RS-232 serial port device driver
CC3Disk	Disk driver
D0, D1	Disk device descriptor
TERM	Terminal device descriptor
T1	RS-232 serial port device descriptor
P	Printer (serial) device descriptor
P1	Printer (serial) device descriptor

Clock	Real-time clock module
CC3GO	System startup process
W - W7	Window device descriptors W, W1, W2, W3, W4, W5, W6, W7

OS-9 stores the modules loaded during the system startup with a minimum of fragmentation. To include additional modules, create new bootstrap files using the OS9GEN command or the CONFIG program supplied with OS-9. You cannot unlink a module loaded as part of the bootstrap.

After booting, when the system switches the boot block into its own address space, any non-system files included in the bootstrap decrease the memory available in the system mode. It is best to place optional modules in a separate file and load them as part of the system startup procedure. One example is the shell. Never include the shell as part of a system boot file in OS-9 Level Two systems.

The Sys Directory

The OS-9 SYS directory contains a number of important files:

- Errmsg is the error message file.
- Helpmsg contains syntax and usage information.
- Stdfonts contains the standard software fonts for use on graphic windows.
- Stdpats_2, Stdpats_4, and Stdpats_16 contain screen background and fill patterns for 2, 4, and 16 color graphics screens, respectively.
- Stdptrs contains graphic pointer images for use with a mouse.

These files, and the SYS directory itself, are not required to boot OS-9, but you do need them if you plan to use the ERROR or HELP commands, or if you intend to use text, or mouse pointers on graphic windows. You can also add other system-wide files of a similar nature.

The Startup File

The Startup file (/D0/startup) is a shell procedure file that OS-9 automatically processes as part of the system boot. You can include any legal shell command line in the Startup file. Many people include SETIME to start the system clock. If this file is not present, the system starts correctly, but the system time is not accurate.

The CMDS Directory

The directory /D0/CMDS is the system-wide command directory normally shared by all users as their working execution directory. The shell resides in the CMDS directory. The system startup process CC3go makes CMDS the initial execution directory. You can add your own programs to the CMDS directory and have them execute in the same manner as the original system commands.

Making New System Diskettes

Getting Started With OS-9 told you how to create new system diskettes using the CONFIG utility. There are other ways to create system diskettes and either add or subtract capabilities. The following information provides guidelines on how to do this. For more detailed instructions see the descriptions of the CONFIG, OS9GEN, and COBBLER commands in this manual.

Before starting any of the following procedures, you need a blank, formatted diskette on which to place your system files. Then, choose one of the following methods to update your system:

- Use the OS9GEN command to add modules to the existing OS9Boot file.
- Use CONFIG to select the modules you want to include in the OS9Boot file.

If you choose to use CONFIG, the utility creates a complete system during the process. If you use OS9GEN, follow these steps:

1. Create the OS9Boot file using OS9GEN.
2. Create or copy the Startup file.
3. Copy the CMDS and SYS directories and the files they contain.

You can perform these steps manually or do them automatically by using one of these methods:

- Creating and using a shell procedure file
- Using a shell procedure file generated by DSAVE

Technical Information for the RS-232 Port

You can operate the RS-232 port or the printer at all standard baud rates from 110 baud to 19200 baud. (The default rate is 9600 baud for /t2, and 600 baud for /p.) The default format used is 8 data bits, no parity, and 1 stop bit.

Use the XMODE command to set the port's baud rate, parity, word length, stop bits, end-of-line delay, auto line feed, and so forth. To examine the printer's current settings, type:

```
xmode /p ENTER
```

Then, if you want to make changes, use XMODE with information from the following chart. Select the parameter you want from the left column of each chart, and then select the corresponding number from the "Value to Use" column and write it down. After you select the proper value from each chart, add them together to obtain a final value for XMODE. All values must be hexadecimal.

Stop Bits

Number of Stop Bits	Value to Use
1 Stop Bit	0
2 Stop Bits	80

Word Length

Word Length	Value to Use
7 Bits	20
8 Bits	0

Baud Rate

Bits Per Second	Value to Use
110 BPS	0
300 BPS	1
600 BPS	2
1200 BPS	3
2400 BPS	4
4800 BPS	5
9600 BPS	6
19200 BPS	7

For instance, to set the printer parameters to one stop bit, a word length of seven bits, and a baud rate of 600, select 0 from the Stop Bits chart, 20 from the Word Length chart, and 2 from the Baud Rate chart. Add the values together:

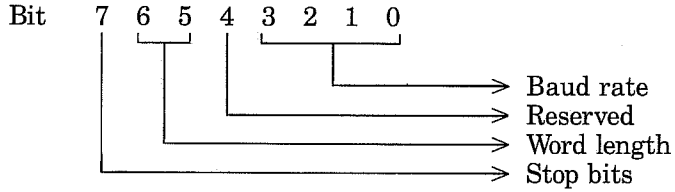
$$0 + 20 + 2 = 22$$

The command to set the printer port for this configuration is:

```
xmode /p baud=22 ENTER
```

When you use XMODE to set baud, parity, and stop bit values, you are actually setting the bits of a special byte to certain values. OS-9 uses these values to determine how to handle subsequent input/output operations. A bit is a binary digit and can be either 1 or 0. A byte consists of eight bits and can represent a value between 0 and 255.

The following chart shows the bits that control baud rate, word length, and stop bits for input/output operations on a specified device.



If the stop bit value = 0, stop bits = 1
If the stop bit value = 1, stop bits = 2

If the word length value = 00, word length = 8 bits
If the word length value = 01, word length = 7 bits

If the baud rate value = 0, baud rate = 110
If the baud rate value = 1, baud rate = 300
If the baud rate value = 2, baud rate = 600
If the baud rate value = 3, baud rate = 1200
If the baud rate value = 4, baud rate = 2400
If the baud rate value = 5, baud rate = 4800
If the baud rate value = 6, baud rate = 9600
If the baud rate value = 7, baud rate = 19200 (/t2 ACIAPAK only)
If the baud rate value = 7, baud rate = 32000 (/t1 SIO only)

Use `XMODE TYPE=value` to set parity, MDM (modem) kill, and the not ready delay. *Value* is a hexadecimal value you calculate from the following chart:

Parity

Type of Parity	Value to Use
None	0
Mark	A0
Space	E0
Even	60
Odd	20

MDM Kill

Kill Switch	Value to Use
On	10
Off	0

Not Ready Delay

Not Ready Delay	Value to Use
0 seconds	0
1 second	1
2 seconds	2
3 seconds	3
↓	↓
↓	↓
↓	↓
	↓
15 seconds	F

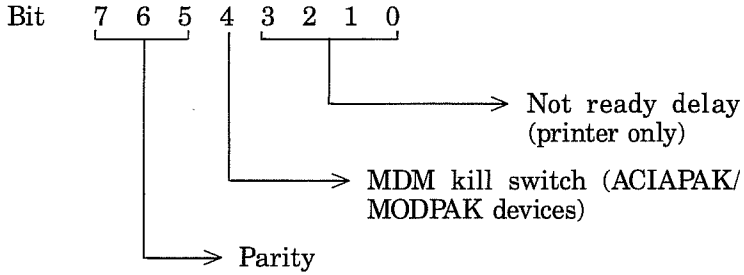
Select a value from each chart, and add them together to get a final TYPE value. For instance, to select even parity, MDM kill off, and a not ready delay of 10 seconds, select these values and add them:

60 + 0 + A = 6A

To set the new values, type:

```
xmode /p type=6a [ENTER]
```

The following chart shows the bits that control parity, the modem kill switch, and the not ready delay.



If the parity value is 000, then parity = none
If the parity value is 101, then parity = MARK, no check
If the parity value is 111, then parity = SPACE, no check

If the MDM kill switch value is 0, then DCD loss = no kill
If the MDM kill switch value is 1, then DCD loss = kill

The value of the not ready delay bits equals the number of seconds delay.

For more information on setting other parameters, such as the end-of-line delay (null count), see the XMODE command reference in Chapter 6.

Chapter 6

System Command Descriptions

This chapter contains alphabetical descriptions of the commands supplied with OS-9. Ordinarily, you call the commands from the shell, but you can also call them from most other programs in the OS-9 family—including BASIC09 and the Macro Text Editor.

Warning: Do not attempt to use OS-9 Level One commands with the OS-9 Level Two system or to use Level Two commands with the Level One system.

Organization of Entries

Each command entry includes:

- The name of the command
- A *syntax* line, which shows you the format and spelling to use when you type the command
- A brief definition of what the command does
- Information about any options available with the command
- Notes about the command and how to use it
- One or more examples of command use

Command Syntax Notations

OS-9 requires that you enter the various parts of a command in the correct order and in the correct format. An example of the proper *syntax* follows the command name.

The syntax line always begins with the name of the command. Occasionally, that's all you need (except for pressing ENTER). But other commands either require, or can accept, parameters (variables that give instructions to OS-9).

Some syntaxes include *variables* (shown in italics) that you replace with specific parameters. For instance, the BUILD command syntax is:

```
build filename ENTER
```

BUILD is the command name. You type it exactly as shown. However, *filename* is a variable. Replace it with the actual name you want to give to the file you are creating. If you want to create a file named Myfile, type:

```
build myfile ENTER
```

Pressing ENTER executes the command.

Common variables are:

arglist	arglist (argument list) is similar to *paramlist*, but it includes command names as well as command parameters.
devname	device name (/P, /TERM, /M1)
commandname	command name
dirname	directory name
filename	file name
hex	a hexadecimal number
hh/mm/ss	hour/minutes/seconds
modname	name of a memory module
n	a decimal number
number	a numeric value
opts	options
paramlist	a list of parameters
pathlist	a complete path to a directory or file
permission	file permission abbreviations
procID	process ID number
text	a string of characters
tickcount	a numeric value representing system clock ticks
value	a numeric value
yy/mm/dd	year/month/day

[ ] Brackets indicate that the material within them is optional and not necessary for the execution of the command.

... An ellipsis indicates that you can repeat the material immediately preceding the ellipsis. For instance, [*filename*][...] means that you can specify more than one filename to the command. Following is the syntax for the DISPLAY command:

```
display hex [...] ENTER
```

This means you can include more than one hex number with DISPLAY, such as:

```
display 54 48 49 53 20 49 53 20 41 20 53 45 43
52 45 54 20 4D 45 53 53 41 47 45 ENTER
```

Command syntaxes do not include the shell's built-in options (for instance I/O redirection) because the shell filters out its options before it passes the command line to the program being called.

Command Summary

This section describes the format and use of OS-9 commands.

The following list is a summary of these commands:

ATTR ..... Changes a file's attributes
BACKUP ..... Makes a copy of a diskette
BUILD ..... Builds a text file
CHD ..... Changes the working data directory
CHX ..... Changes the working execution directory
CMP ..... Compares files
COBBLER ... Makes an OS9Boot file
CONFIG ..... Creates a system diskette to your specifications
COPY ..... Copies data
DATE ..... Displays the system date and (optionally) the time
DCHECK .... Checks a disk file structure
DEINIZ ..... Deinitializes a device previously initialized with INIZ
DEL ..... Deletes a file or files
DELDIR ..... Deletes a directory's files, then deletes the directory
DIR ..... Displays the names of all files in a directory
DISPLAY ..... Displays the characters represented by hexadecimal values
DSAVE ..... Generates a procedure file to copy files

ECHO ..... Echoes text to the screen
EDIT ..... Calls the OS-9 Macro Text Editor
ERROR ..... Displays a description of the last error code
EX ..... Causes the shell process to execute another process
FORMAT .... Prepares a disk for data storage
FREE ..... Displays the amount of free space on a disk
HELP ..... Displays the syntax and use of commands
IDENT ..... Displays OS-9 module identification
INIZ ..... Initializes and attaches devices
KILL ..... Terminates a process
LINK ..... Links a module into memory
LIST ..... Lists the contents of disk data files
LOAD ..... Loads a module into memory
MAKDIR .... Creates a directory
MDIR ..... Displays the names of the modules in memory
MERGE ..... Copies and combines files
MFREE ..... Displays a list of free RAM
MODPATCH .. Makes changes to a module in memory
MONTYPE ... Establishes the type of monitor in use
OS9GEN ..... Builds and links a bootstrap file
PROCS ..... Displays the names of the current processes
PWD ..... Displays the name of the current data directory
PXD ..... Displays the name of the current execution directory
RENAME .... Changes the name of a file or directory
SETIME ..... Activates and sets the system clock
SETPR ..... Sets a process's priority
SHELL ..... Creates a child shell to process one or more commands
TMODE ..... Changes the terminal's operating mode
TUNEPORT Adjusts the loop delay for the baud rate of /P or /T1 devices
UNLINK ..... Unlinks memory modules
WCREATE ... Creates a window
XMODE ..... Displays or changes a device's initialization parameters

ATTR

Syntax: **attr *filename* [*permission*]**

Function: Lets you examine or change a file's security permissions.

Parameters:

filename	The name of the file you want to examine or change.
permission	One or more of the following attribute options.

Options:

The file permission abbreviations you can use are:

-d	Changes a file directory file to not a non-directory file.
s	Specifies that the file is not single-user and can serve only one user at a time.
r	Specifies that only the owner can read the file.
w	Specifies that only the owner can write to (change) the file.
e	Specifies that only the owner can execute the file.
pr	Specifies that the public (anyone) can read the file.
pw	Specifies that the public (anyone) can write to the file.
pe	Specifies that the public (anyone) can execute the file.
-a	Tells ATTR not to display the attributes. Use this option when you wish to change attributes without displaying them.

Notes:

- To use ATTR, type the command name followed by the name of the file you want to change. Next, type a list of the permissions to turn on or off. Turn a permission on by typing its abbreviation or off by typing its abbreviation preceded by a minus sign. ATTR has no effect on permissions you do not name.
- If you do not specify any permissions, ATTR displays the file's current attributes.
- You cannot change the attributes of a file you don't own. User 0 can change the attributes of any file in the system.
- Use ATTR to change a directory into a file after deleting all the directory's files. You cannot change a file to a directory with ATTR. (See MAKDIR.)

Examples:

- To remove public read and write permission from a file named Myfile, type:

```
attr myfile -pr -pw ENTER
```

- To give read, write, and execute permission to everyone for the file Myfile, type:

```
attr myfile r w e pr pw pe ENTER
```

- To display the current permissions of a file named Datalog, type:

```
attr datalog ENTER
```

BACKUP

Syntax: **backup [*opts*][*devname1*][*devname2*]**

Function: Copies all data from one disk to another.

Parameters:

devname1	The drive containing the disk files you want to back up.
devname2	The drive containing the disk to which you want to transfer the files.
opts	One or more of the following options.

Options:

e	Cancels the backup if a read error occurs.
s	Lets you backup a diskette using only one drive.
-v	Tells BACKUP not to verify the data written to the destination diskette.
#*n*K	Increases to *n* the amount of memory that BACKUP can use. Increasing the amount of memory assigned to BACKUP speeds the procedure. *n* can be either in pages of 256 bytes or in kilobytes (1024 bytes). Include K to indicate kilobytes.

Notes:

- BACKUP performs a sector by sector copy, ignoring file structures. In all cases, the devices specified *must* have the same format (size, density, and so forth) and the destination disk must **not** have defective sectors.

- If you omit both device names, the system assumes you are copying from /D0 to /D1. If you omit only the second device name, OS-9 performs a single-drive backup on the specified drive.

- The following demonstrates a complete backup of /D0 to /D1. In the example, the diskette in Drive /D1 is a formatted diskette with the name MYDISK. *Scratched*, which appears in one of the following messages, means erased. You type:

```
backup [ENTER]
```

The screen display and your input are:

```
Ready to backup from /d0 to /d1 ?: [Y]
MYDISK
 is being scratched
OK?: [Y]
Sectors copied: $0276
Verify pass
Sectors verified: $0276
```

- Following is an example of a single-drive back up. BACKUP reads a portion of the source diskette (the diskette you are copying) into memory. It then prompts you to remove the source diskette and put the destination diskette (the diskette receiving the copy) into the drive.

  After BACKUP writes to the destination diskette, remove the destination diskette and put the source diskette back into the drive. Continue swapping as prompted until BACKUP copies the entire diskette.

  Giving BACKUP as much memory as possible means you have to make fewer diskette exchanges. If enough free memory is available, you can assign up to 56 kilobytes for the backup operation. An Error 207 means that your computer does not have the specified amount of memory free. To assign 32 kilobytes to backup, type:

```
backup /d0 #32k [ENTER]
```

The screen display and your responses are as follows:

```
Ready to backup from /dØ to dØ ?: [Y]
Ready Destination, hit a key: [ENTER]
MYDISK
 is being scratched
OK?: [Y]
Ready Source, hit a key: [ENTER]
Ready Destination, hit a key: [ENTER]
Ready Source, hit a key: [ENTER]
Ready Destination, hit a key: [ENTER]
```



```
Ready Destination, hit a key: [ENTER]
Sectors copied: $Ø276
Verify pass
Sectors verified: $Ø276
```

In this procedure, the dollar symbol ($) indicates hexadecimal numbers. BACKUP copied 276 hexadecimal (or 630 decimal) sectors.

Examples:

- To back up the diskette in Drive /D2 to the diskette in Drive /D3, type:

```
backup /d2 /d3 [ENTER]
```

- To back up from Drive /D0 to Drive /D1, without verification, type:

```
backup -v [ENTER]
```

BUILD

Syntax: **build *filename***

Function: Builds a text file by copying input from the standard input device (the keyboard) into the file specified by *filename*.

Parameters:

filename	The name of the file you are creating.

Notes:

- BUILD creates a file, naming it *filename*. It then displays a question mark (?) and waits for you to type a line. When you type a line and press [ENTER], BUILD writes the line to the disk.
- When you finish entering the lines for the new file, press [ENTER], without any preceding text, to close the file and terminate the operation.
- The following example demonstrates how to build a text file named Newfile:

```
build newfile [ENTER]

? THE POWERS OF THE OS-9 [ENTER]
? OPERATING SYSTEM ARE TRULY [ENTER]
? FANTASTIC. [ENTER]
? [ENTER]
```

- To view the newly created file, type:

```
list newfile [ENTER]
```

The screen displays:

```
THE POWERS OF THE OS-9
OPERATING SYSTEM ARE TRULY
FANTASTIC.
```

Examples:

- To create a new file called Small_file and put into it whatever you type at the keyboard, type:

  ```
  build small_file ENTER
  ```

- To direct whatever you type to the printer, type:

  ```
  build /p ENTER
  ```

- You can use BUILD to transfer, or redirect, data from one file to another. Instead of the keyboard, this example uses a file named Mytext file for the input device. The output device is Terminal 1.

  ```
  build <mytext /t1 ENTER
  ```

CHD
CHX

Syntax: **chd *pathlist***
chx *pathlist*

Function: CHD changes the current working (data) directory, and CHX changes the current execution directory.

Parameters:

pathlist Specifies the directory for the current working or execution directory.

Notes:

- CHD and CHX do not appear in the CMDS directory because they are built into the shell.

Examples:

- To change the current working (data) directory to the PROGRAMS data directory located on the diskette in Drive /D1, type:

```
chd /d1/programs ENTER
```

- To change the execution directory to the parent directory of the current execution directory, type:

```
chx .. ENTER
```

- To change the execution directory to TEXT_PROGRAMS, a subdirectory of BINARY_FILES, type:

```
chx binary_files/text_programs ENTER
```

- To return the execution directory and the data directory back to the default directories, type:

```
chx /d0/cmds; chd /d0 ENTER
```

Or, if you are using a hard disk, type:

```
chx /h0/cmds; chd /h0 ENTER
```

CMP

Syntax: **cmp *filename1 filename2***

Function: Opens two files and compares the binary values of corresponding data bytes in the files. If CMP encounters any differences in the file, it displays the file offset (address) and the values of the bytes from each file.

Parameters:

filename1 are the files to compare.
filename2

Notes:

- The comparison ends when CMP encounters an end-of-file marker in either file. CMP then displays a summary of the number of bytes compared and the number of differences found.

Examples:

- To compare two files named Red and Blue, type:

```
cmp red blue [ENTER]
```

Following is a sample screen display:

```
Differences

byte        #1   #2
--------    --   --
00000013    00   01
00000022    B0   B1
0000002A    9B   AB
0000002B    3B   36
0000002C    6D   65

Bytes compared:  0000002D
Bytes different: 00000005
```

- To compare two files that are identical, such as Red1 and Red2, type:

```
cmp red1 red2 [ENTER]
```

The screen display might be:

```
Differences

None ...

Bytes compared:  0000002D
Bytes different: 00000000
```

COBBLER

Syntax: **cobbler *devname***

Function: Creates the OS9Boot file required on any OS-9 boot diskette.

Parameters:

devname	The disk drive containing the diskette on which you want to create a new OS9Boot file.

Notes:

- COBBLER creates the new OS9Boot file with the *same* modules loaded during the most recent bootstrap. (To add modules to the bootstrap file, use OS9GEN.) COBBLER also writes the OS-9 kernel on Track 34 and excludes these sectors from the diskette allocation map. If any files are present on these sectors, COBBLER displays an error message.
- The new boot file must be contiguous on the diskette. For this reason, you should use COBBLER only with a newly formatted diskette. If you use COBBLER on a diskette that does not have a storage block large enough to hold the boot file, COBBLER destroys the old boot file, and OS-9 cannot boot from that diskette.
- To change device attributes permanently, use XMODE before using COBBLER.

Examples:

- To save the attributes of the current device on the system diskette, type:

```
cobbler /d0 [ENTER]
```

If you use COBBLER on a diskette that is not newly formatted, the screen displays:

```
WARNING - FILE(S) OR KERNEL
PRESENT ON TRACK 34 - THIS
TRACK NOT REWRITTEN
```

CONFIG

Syntax: config

Function: Lets you create a system diskette that includes only the device drivers and commands you select. CONFIG automatically adjusts its screen display for either 32- or 80-column display.

Notes:

- When executed, CONFIG displays menus of all I/O options and system commands. You select only those options and commands you want to include on a new system diskette.

  Creating such a system diskette lets you make the most efficient use of computer memory and system diskette storage.

- The CONFIG utility is on the BASIC09/CONFIG diskette. Copy this diskette, using the OS-9 BACKUP command. Make the copy your working diskette. Keep the original in a safe place to use for future backups. After you boot your system, you can put the working copy of the BASIC09/CONFIG diskette in drive /d0. Then, type these commands:

  ```
  chx /d0/cmds; chd /d0/modules ENTER
  ```

- CONFIG does not require initial parameters. You establish parameters during the operation of the command. Be sure the execution directory is /D0/CMDS before executing CONFIG.

- You could save time by using BACKUP to create a system disk, using CONFIG to create a new boot file, and then deleting unwanted commands. However, this process causes fragmentation of diskette space and results in slower disk access. CONFIG causes no fragmentation.

- The MODULES directory of the BASIC09/CONFIG diskette contains all the device drivers and device descriptors supported by OS-9. The filename extension describes the type of file, as noted in the following table:

Extension	Module Type
.dd	Device Descriptor module
.dr	Device Driver module
.io	Input/Output subroutine module
.hp	Help file
.dw	Window Device Descriptor module
.dt	Terminal Device Descriptor module
.mn	File Manager module

Examples:

The following steps take you through the complete CONFIG process:

1. With the BOOT/CONFIG diskette in the current drive, type:

   ```
   config [ENTER]
   ```

2. CONFIG asks whether you want to use one or two disk drives. Press [1] for single- or [2] for two-drive operation.

   If you specify one drive, continue with Step 3.

   If you specify two drives, a display asks you to:

   ```
   ENTER NAME OF SOURCE DISK:
   ```

   Type `/d0` [ENTER]

   A display now asks you to:

   ```
   ENTER NAME OF DEST. DISK:
   ```

   Type `/d1` [ENTER]

3. After a pause to build a descriptor list, the program displays a list of the various devices from the MODULES directory. Use [↑] and [↓] to move to a device. To include the device on the system diskette, press [S] once. CONFIG displays an X by the selected device. To exclude a selected device, press [S] again to erase the X.

A special help command provides information about each device. To display information about the current device (the device indicated by the arrow (→)), press [H].

The list of devices might require more than one screen. Use [→] to move ahead page by page and [←] to move back.

The devices you can select and their descriptions are listed in Chapter 2 under the section "Device Names."

You must select a "D0" device as your first disk drive. Select from the list of D devices for other floppy disk drives. Select P to use a printer with OS-9, T1 to use a terminal, M1 to use a modem, and so on.

4. After selecting the devices you desire, press [D]. The screen displays, `ARE YOU SURE (Y/N) ?` If you are satisfied with your selections, press [Y]. If you want to make changes, press [N].

5. To use your computer keyboard and video display, you must select either TERM_VDG or TERM_WIN. You use TERM_VDG for a 32-column display. For a TERM window that enables you to select character displays up to 80-columns, select TERM_WIN.

6. CONFIG builds a boot list from the selected devices and their associated drivers and managers in the MODULES directory of the current drive. It next displays two clock options:

```
1 - 60HZ (AMERICAN POWER)
2 - 50HZ (EUROPEAN POWER)
```

7. If you live in the United States, Canada, or any other country with 60hz electrical power, press [1]. If you live in a country with 50hz power, press [2].

   If you have a single disk drive, a screen prompt asks you to swap diskettes and press [C]. When asked for the SOURCE diskette, insert the BASIC09/CONFIG diskette. When asked for the DESTINATION diskette, insert the diskette that is to be your new OS-9 system diskette.

   If you have more than one drive, a screen prompt asks you to insert a blank formatted diskette (the DESTINATION diskette) in the destination drive. The rest of the boot file creation is automatic.

8. After creating the boot file, CONFIG displays a menu of the commands you can include on your system diskette. You have the following choices:

`[N]o Commands, Stop Now`	— Do not add any commands
`[F]ull Command Set`	— Add all OS-9 commands from the current CMDS directory
`[I]ndividually Select`	— Select commands one by one
`[H] Receive Help`	— Get help on the command set

   Press [N] if you want to transfer a new boot file to a diskette on which you have previously copied the OS-9 system. If you have only one disk drive, this procedure is quicker than using the CONFIG utility to complete the entire system transfer, because it requires fewer disk swaps.

   Press [F] to make an exact copy of the CMDS directory on your source diskette with a new boot file.

   Press [I] to individually select commands to copy on the new diskette. The [I] option displays a menu similar to the device selection screen. Press [S] to select or exclude commands, and use the arrow keys to move among the commands in the menu. CONFIG selects files marked with an X for inclusion on the new system diskette. If a command does not have an X beside it, CONFIG excludes it from the new system diskette.

9. If you have a multi-drive system, a prompt appears asking you to insert your OS-9 system diskette in the destination drive. Press the space bar. The process finishes the CONFIG operation, and returns to OS-9.

   If you have a single-drive system, you swap diskettes during the final process. This time, the SOURCE diskette is the OS-9 System diskette. The DESTINATION diskette is the system diskette you are creating. The number of swaps depends on the number of options you select.

   Note: When using CONFIG you do not have to use your system diskette as the source diskette to install the commands. The program can use any diskette that contains a CMDS directory.

COPY

Syntax: **copy *pathlist1 pathlist2* [opts]**

Function: Copies data from one file or device to another file or device.

Parameters:

pathlist1 — The name of the existing file or device from which you want to copy.

pathlist2 — The name of the device or file to receive the copy. If you are copying data to a file, the file must not already exist.

Options:

-s — Causes COPY to perform a single-drive copy operation. *pathlist2* must be a full pathlist if you use -s. In a single-drive procedure, COPY reads a portion of the source disk into memory and then asks you to exchange the source and the destination diskette and press [C]. COPY might ask you to exchange diskettes several times before it completes duplicating the entire file.

#*n*[K] — Allows the use of more memory for the COPY procedure. If you specify K, *n* represents the amount of memory you want to use, in units of 1024 bytes. If you do not specify K, *n* represents the number of 256-byte memory pages. Using this option can increase speed and reduce the number of diskette swaps required for single-drive copies.

copy *.* -w=destination

Notes:

- If *pathlist2* is a disk file, COPY automatically creates it. Data can be of any type, and COPY does not modify the file in any way.
- COPY does not add important codes (for example, line feeds). Use LIST instead of COPY when sending a text file to a terminal or printer.
- Following is an example of the screen display and your responses for COPY using a single drive:

```
copy /d0/cat /d0/animals/cat -s #32k [ENTER]
Ready DESTINATION, hit C to continue: [C]
Ready SOURCE, hit C to continue: [C]
Ready DESTINATION, hit C to continue: [C]
```


This example assigns 32 kilobytes of memory for COPY to use. If enough free memory is available, you can specify up to 56 kilobytes. Copy continues asking you to swap the source and destination diskettes until the transfer is complete.

Examples:

- To copy File1 to File2 using 15K of memory, type:

```
copy file1 file2 #15k [ENTER]
```

- To copy the News file on the diskette in Drive /D1 to a new file named Messages on the diskette in Drive /D0, type:

```
copy /d1/joe/news /d0/peter/messages [ENTER]
```

DATE

Syntax: **date [t]**

Function: Displays the current date.

Options:

t	Causes the time to appear with the date.

Notes:

- Following is an example of how to use SETIME to set a new date and time for the system and how to use DATE to check system date and time:

  ```
  setime [ENTER]
  ```

  A possible screen display and your responses follows:

  ```
          yy/mm/dd hh.mm.ss
  Time?   86/08/22  14.19.00 [ENTER]

  date [ENTER]

  August 22, 1986

  date t [ENTER]

  August 22, 1986 14.20.20
  ```

Examples:

- To display the system date and time, type:

  ```
  date t [ENTER]
  ```

- To direct the DATE command's output to the printer, type:

  ```
  date t >/p [ENTER]
  ```

DCHECK

Syntax: **dcheck [*-opts*] *devname***

Function: Checks a disk's file structure.

Parameters:

devname	The disk drive to check.
opts	One or more of the following options.

Options:

-s	Counts the number of directories and files and displays the results. This option causes DCHECK to check only the file descriptors for accuracy.
-b	Suppresses listing of unused clusters (clusters allocated but not in the file structure).
-p	Prints pathlists for questionable clusters.
-w=*pathlist*	Specifies a path to a directory for work files.
-m	Saves allocation map work files.
-o	Prints DCHECK's valid options.

Notes:

- Sometimes the system allocates sectors on a disk that are not actually associated with a file or with the disk's free space. This situation can happen if you remove a disk from a drive while files are open. You can use DCHECK to detect this condition, as well as check the general integrity of directory/file links.

- After verifying and printing some vital file structure parameters, DCHECK follows pointers down the disk's file system tree to all directories and files on the disk. As it does so, it verifies the integrity of the file descriptor sectors, reports any discrepancies in the directory/file links, and builds a sector allocation map from the segment list associated with each file. If any file descriptor sectors (FDS) describe a segment with a cluster not within the file structure of the disk, DCHECK displays a message like this:

```
*** Bad FD segment ($xxxxxx-$yyyyyy) for file:
  (pathlist)
```

  This message indicates that a segment starting at sector *xxxxxx* and ending at sector *yyyyyy* is not on the disk. If any of the file descriptor sectors are bad, the entire FD might be defective. DCHECK does not update the allocation map for corrupt FDS.

- While building the allocation map, DCHECK also ensures that each disk cluster appears once and only once in the file structure. If it discovers duplication, DCHECK displays a message like this:

```
Cluster $xxxxxx was previously allocated
```

  This message indicates that DCHECK has found cluster *xxxxxx* more than once in the file structure. DCHECK reprints the message each time a cluster appears in more than one file.

  Then, DCHECK compares the newly created allocation map with the allocation map stored on the disk and reports any differences with messages like these:

```
Cluster $xxxxxx in allocation map but not in file
                structure

Cluster $xxxxxx in file structure but not in
                allocation map
```

  The first message indicates that sector number *xxxxxx* (hexadecimal) is not part of the file system, but the disk's allocation map has assigned it. FORMAT might exclude some sectors from the allocation map because they are defective.

The second message indicates that the cluster starting at sector *xxxxxx* is part of the file structure, but the disk's allocation map has not assigned it. Later operations might allocate this cluster, overwriting the contents of the cluster with data from the newly allocated file. (Clusters that DCHECK previously allocated can have this problem.)

- DCHECK builds its disk allocation map in a file called *pathlist*/DCHECK*pp*0, where *pathlist* is specified by the -w option and *pp* is the process number in hexadecimal. Each bit in this bitmap file corresponds to a cluster of sectors on the disk. If you use the -p option, DCHECK creates a second bitmap file (*pathlist2*/DCHECKpp1) that has a bit set for each cluster DCHECK finds as "previously allocated" or "in file structure but not in allocation map." DCHECK then makes another pass through the directory structure to determine the pathlists for these questionable clusters. You can save the bitmap work files by specifying the -m option on the command line.
- For best results, DCHECK should have exclusive access to the disk being checked. Otherwise, the command might be fooled by a change in the disk allocation map while DCHECK is building a bitmap file. DCHECK cannot process disks with more than 39 levels of directories.
- -p causes DCHECK to make a second pass through the file structure and print pathlists for clusters that are not in the allocation map but are allocated or existing in a file structure.

  -w tells DCHECK where to place its allocation map work file(s). The specified pathlist must be a full pathlist for a directory. (DCHECK uses directory /D0 if you do not specify -w.) If you doubt the structure integrity of the diskette being checked, do not place the allocation map work files on that diskette.

Examples:

- The following two examples demonstrate DCHECK sessions:

```
dcheck /d2 [ENTER]
```

A sample screen display might be:

```
Volume - 'My system disk' on device /d2
$009A bytes in allocation map
1 sector per cluster
$000276 total sectors on media
Sector $000002 is start of Root directory
FD
$0010 sectors used for id, allocation map
and Root directory
Building allocation map work file...
Checking allocation map file...

'My system disk' file structure is intact
1 directory
2 files
```

```
dcheck -mpw=/d2 /d0 ENTER
```

A sample screen display might be:

```
Volume - 'System diskette' on device /d0
$0046 bytes in allocation map
1 sector per cluster
$00022A total sectors on media
Sector $000002 is start of Root directory
FD
$0010 sectors used for id, allocation map
and Root directory
Building allocation map work file...
Cluster #00040 was previously allocated
*** Bad FD segment ($111111-$23A6F0) for
file: /D0/TEXT/junky.file
Checking allocation map file...
Cluster $000038 in file structure but not
in allocation map
Cluster $00003B in file structure but not
in allocation map
Cluster $0001B9 in allocation map but not
in file structure
Cluster $0001BB in allocation map but not
in file structure
```

```
Pathlists for questionable clusters:
Cluster $000038 in path: /d0/OS9boot
Cluster $00003B in path: /d0/OS9boot
Cluster $000040 in path: /d0/OS9boot
Cluster $000040 in path: /d0/test/
double.file

1 previously allocated clusters found
2 clusters in file structure but not in
allocation map
2 clusters in allocation map but not in
file structure
1 bad file descriptor sector

'System diskette' file structure is not
intact
5 directories
25 files
```

DEINIZ

Syntax: **deiniz *devname* [...]**

Function: Deinitializes and detaches a device.

Parameter:

devname The name of one or more devices you want to deinitialize.

Notes:

- Use DEINIZ with INIZ. For example, you can use INIZ to initialize a window, then redirect information to the window. View the information by pressing CLEAR until it appears. When you no longer need the window, use DEINIZ to remove the window and return its memory to the system.
- DEINIZ performs an OS-9 I$Detach call for all specified devices.

Example:

To deinitialize the /w1 (Window 1) device after it has been initialized, type:

```
deiniz w1 ENTER
```

DEL

Syntax: **del [-x] *filename* [...]**

Function: Deletes the file(s) specified.

Parameter:

filename — The name of the file to delete. Include as many filenames as you want.

Option:

-x — Causes DEL to assume the file is in the current execution directory.

Notes:

- You can delete only files for which you have write permission.

  You can delete a directory in two ways: (1) Delete all the files in the directory, change it to a non-directory file using ATTR, then use DEL to remove the directory, or (2) Use the DELDIR command.

- The following example shows what appears on the screen when you display a directory, delete one of the directory's files, then display the directory again:

```
dir /d1 [ENTER]

directory of /d1 14.29.46
myfile    newfile

del newfile [ENTER]
dir /d1 [ENTER]

directory of /d1 14.30.37
myfile
```

Examples:

- To delete files named Text_program and Test_program, type:

```
del text_program test_program ENTER
```

- To delete a file on a drive other than the current working drive, use a complete pathlist, such as:

```
del /d1/number_five ENTER
```

- To delete a file named Cmds.subdir in the current execution directory, type:

```
del -x cmds.subdir ENTER
```

DELDIR

Syntax: **deldir *dirname***

Function: Deletes all subdirectories and files in a directory; then, deletes the directory itself.

Parameter:

dirname The pathlist to the directory you want to delete.

Notes:

- DELDIR is a convenient alternative to individually deleting all the files and subdirectories from a directory before deleting the directory itself.
- When DELDIR runs, it displays a prompt after the command line:

```
deldir oldfiles [ENTER]
Deleting directory file.
List directory, delete directory, or quit ?
(l/d/q)
```

  Pressing [L] causes a DIR E command to run so you can see the directory files before DELDIR removes them.

  Pressing [D] starts the deletion process.

  Pressing [Q] cancels the command.

- The directory to be deleted might include other directories, which in turn might include other directories, and so forth. In this case, DELDIR begins with the lower directories and works its way upward.

  You must have write permission to delete any files and directories in this substructure. If not, DELDIR terminates when it encounters the first file for which you don't have write permission.

- DELDIR automatically calls DIR and ATTR. Therefore, these files must reside in the current execution directory.

DIR

Syntax: **dir [*opts*][*dirname* or *pathlist*]**

Function: Displays a formatted list of filenames in a directory. The output format adjusts itself for 80- or 32-column displays.

Parameters:

dirname	The name of the directory you want to view.
pathlist	The pathlist to the directory you want to view.
opts	Either or both of the following options.

Options:

If you don't specify any parameters, DIR shows the current data directory.

x	Displays the current execution directory.
e	Displays the entire description for each file: size, address, owner, permissions, date and time of last modification.

Examples:

- To display the current data directory, type:

  ```
  dir [ENTER]
  ```

- To display the current execution directory, type:

  ```
  dir x [ENTER]
  ```

- To display the entire description of all files in the current execution directory, type:

  ```
  dir x e [ENTER]
  ```

- To display the parent of the current data directory, type:

```
dir .. ENTER
```

- To display a directory named NEWSTUFF, type:

```
dir newstuff ENTER
```

- Following is a sample 80-column DIR display using the e option:

```
dir e ENTER
```

The screen might display:

```
Directory of  . 16:50:12

  Owner  Last modified  Attributes  Sector  Bytecount  Name
  -----  -------------  ----------  ------  ---------  ---------
     2F  85/05/20 1631   ------wr        A       3A6C  OS9Boot
      0  85/05/20 1345   d-ewrewr       48        640  CMDS
      0  85/05/20 1350   d-ewrewr      177         A0  SYS
      0  85/05/20 1351   ----r-wr      192          E  startup
      0  85/05/20 1351   d-ewrewr      194         E0  DEFS
```

- Following is an 80-column DIR display using no options:

```
dir ENTER
```

The screen might display:

```
Directory of  . 16:50:37

OS9Boot     CMDS     SYS     startup
DEFS
```

- Following is a 32-column DIR display using the e option:

```
dir e [ENTER]

Directory of  . 16:52:04

Modified on   Owner     Name
   Attr       Sector    Size
============================
85/05/20  1643     2F   OS9Boot
------wr          A      3A6C
85/05/20  1345     0    CMDS
d-ewrewr         48        640
85/05/20  1350     0    SYS
d-ewrewr        177         A0
85/05/20  1351     0    startup
----r-wr        192          E
85/05/20  1351     0    DEFS
d-ewrewr        194         E0
```

- Following is a 32-column DIR display using no options:

```
dir [ENTER]


Directory of  . 16:52:29
OS9Boot    CMDS     SYS
startup    DEFS
```

DISPLAY

Syntax: **display *hex* [...]**

Function: Reads one or more hexadecimal numbers (you type as parameters), converts them to ASCII characters, and writes them to the standard output (normally the screen).

Parameters:

hex	A list of one or more hexadecimal numbers.

Notes:

- Use DISPLAY to send special characters (such as cursor and screen control codes) to terminals and other I/O devices.
- Following is an example of a command and the resulting output. ABCDEF are ASCII characters corresponding to hex 41 42 43 44 45 46.

```
display 41 42 43 44 45 46 [ENTER]
ABCDEF
```

Examples:

- To reroute a *form feed* (hex 0C) to the printer, type:

```
display 0C >/p [ENTER]
```

- To ring the *bell* through the video speaker, type:

```
display 07 [ENTER]
```

DSAVE

Syntax: **dsave [*opts*][*devname*][*dirname*] > *pathlist***

Function: Copies or *backs up* all files in one or more directories.

Parameters:

devname	The drive on which the source directory exists. If you do not specify *devname* DSAVE assumes Drive /D0.
dirname	The name of the destination directory. Use CHD to make the current directory the directory to receive the copies.
pathlist	A command procedure file in which DSAVE stores its output.
opts	One or more of the following options.

Options:

-b	Makes the destination or target diskette a system diskette by copying the source diskette's OS9Boot file, if present.
-i	Indents for directory levels.
-l	Tells DSAVE not to process directories below the current level.
-m	Tells DSAVE not to include MAKDIR commands in the procedure file it creates.
-s*integer*	Sets memory for the copy parameter to *integer* kilobytes.
-v	Verifies copies by forking to CMP after copying each file.

Notes:

- DSAVE does not directly affect the system. Instead, it generates a procedure file that you execute later to do the work.
- When you run DSAVE, it creates a procedure file (a file of commands). You then execute the newly created file by typing its pathlist. The procedure contains all the commands to create and change directories as needed in order to copy the specified directory. DSAVE copies the files in the current data directory. It also copies the current data directory subdirectories, unless you specify the -l option.
- To use DSAVE, first change the data directory to the directory you wish to copy. Execute DSAVE by specifying the drive from which to copy and then redirecting output to a file to receive the copy commands. Be sure to name a file that does not already exist.

  When DSAVE completes the procedure, use CHD to change to the data directory to receive the copied files. Then, execute the procedure file.
- If DSAVE encounters a directory file, it automatically includes MAKDIR and CHD commands in the output before generating COPY commands for files in the subdirectory. The procedure file exactly replicates all levels of the file system from the current data directory downward.
- If the current data directory is the ROOT directory of the disk, DSAVE creates a procedure file that backs up the entire disk file by file. This is useful when you need to copy a number of files from either disks formatted differently or from floppy diskettes to a hard disk.

Examples:

- In the following series of commands, CHD positions you in the ROOT directory of /D2, the directory to be copied. Then, DSAVE makes the procedure file Makecopy. Using CHD /D1 causes the copy to go in the /D1 ROOT directory. The final command executes the procedure file.

```
chd /d2 [ENTER]
dsave /d2 >/d0/makecopy [ENTER]
chd /d1 [ENTER]
/d0/makecopy [ENTER]
```

- The following command copies all files from /D0 to /D1. It pipes the procedure file output of DSAVE into a shell for immediate execution.

```
dsave  /d0  /d1  !  shell [ENTER]
```

- The following command lets you view the output generated by a DSAVE command. It uses 48 kilobytes of memory and indents directories. Because output goes to the screen, this command does not create a procedure file to copy any files:

```
dsave -s48 -i [ENTER]
```

- This command operates in the same manner as the previous command. However, because it specifies a procedure file pathlist, it stores the generated commands in a procedure file rather than displaying them on the screen:

```
dsave -s48 -i > copyfile [ENTER]
```

ECHO

Syntax: **echo *text***

Function: Echoes text to the screen.

Parameters:

text The character or characters you type.

Notes:

- Use ECHO to generate messages in shell procedure files or to send an initialization character sequence to a terminal. The text should not include punctuation characters used by the shell.
- The following example prints the message LISTING ERROR MESSAGES to the screen and lists the file SYS/errmsg to the printer as a background task.

```
echo LISTING ERROR MESSAGES; list sys/
errmsg >/p& ENTER
```

Examples:

- To display a message on the screen, type:

```
echo This text is echoing ENTER
```

- To echo text to the console, type:

```
echo >/term **WARNING DATA ON DISK WILL BE
LOST ENTER
```

- The following combines the ECHO and LIST commands to echo the entered text to the printer and to direct the contents of the Trans file to the printer.

```
echo >/p LISTING OF TRANSACTION; list trans
>/p& ENTER
```

ERROR

Syntax: **error *errnumber* [...]**

Function: Displays the text error message that corresponds with the specified OS-9 error number.

Parameters:

errnumber Is an OS-9 error code in the range 1-255.

Notes:

- ERROR opens the Errmsg file in the SYS directory and reads through the file for an error code that matches the specified number. It then displays the text that corresponds to the error code.
- The Errmsg file contains descriptions of the standard OS-9 errors. The order of the file is arranged to provide quick access to operation system error descriptions.

Example:

- To display a description of the OS-9 error Numbers 215 and 216, type:

```
error 215 216 ENTER
```

The screen displays:

```
215 - Bad Path Name
216 - Path Name Not Found
```

EX

Syntax: **ex *filename***

Function: Starts a process by *chaining* from the current shell to the new process. Chaining means that execution control is turned over to the new process.

Parameters:

filename	The name of the program or module you want to execute.

Notes:

- Because EX is a built in Shell command, it does not appear in the CMDS directory.
- Using EX causes the shell from which you are operating to terminate. If the new process also terminates and you do not have another shell running on another terminal or window, OS-9 is left without any processes, and you must reboot your computer and OS-9.
- If a shell is running on another window or device, you can restart a new shell from that window or device. For instance, if you use EX to initialize BASIC09 from /TERM then exit BASIC09, /TERM is dead and cannot accept keyboard input. However, if you also have a shell operating in a window, you can type the following from that window:

  ```
  shell i=/term& ENTER
  ```

  This reinitializes a shell on /TERM. It can now accept keyboard input and OS-9 commands.
- Use EX to save memory when the shell is not needed, for instance when using BASIC09.

- If you use EX on a command line with other commands, it must be the last command. Any commands following EX are not processed.

Example:

- To run BASIC09 without a resident shell, type:

```
ex basic09 ENTER
```

FORMAT

Syntax: **format *devname* [name] [*opts*]**

Function: Establishes and verifies an initial file structure on a floppy diskette or a hard disk. You must format all disks before you can use them on an OS-9 system.

Parameters:

devname	The drive name of the disk you want to format.
name	The name you want to assign the newly formatted disk. Enclose the disk name in double quotation marks.
opts	One or more of the following options.

Options:

l	Writes system format information only, does not physically format disk.
r	Causes the format to proceed automatically, without issuing prompts.
1	Formats single-sided. Use with single-sided drives or single-sided diskettes in double-sided drives.
2	Causes a double-sided format. Use with double-sided drives and double-sided diskettes.
'cylinders'	The number of cylinders (in decimal) that you want formatted.
:interleave:	The number of the sector interleave value (in decimal).

Notes:

- Be sure the disk you want to format is NOT write-protected. Otherwise, FORMAT generates error code #242 (write protect), and the system returns to the OS-9 prompt without formatting the diskette.
- If you are formatting a hard disk, first type:

  ```
  tmode -pause ENTER
  ```

  This command turns off the screen pause function. Otherwise, the process stops whenever the sector verification process fills the display screen. If you forget to turn off the screen pause, press the space bar whenever the screen fills. Execution then continues.

  When formatting finishes, type:

  ```
  tmode pause ENTER
  ```

  This re-establishes the screen pause function.
- The formatting process works this way:

  1. FORMAT physically initializes a disk and divides its surface into sectors.
  2. FORMAT reads back and verifies each sector. If a sector fails to verify after several attempts, FORMAT excludes it from the initial free space on the diskette. As verification proceeds, the process displays track numbers.
  3. FORMAT writes the disk allocation map, ROOT directory, and identification sector to the first few sectors of Track 0. These sectors must not be defective.
- FORMAT asks for a disk volume name, which can be up to 32 characters long and can include spaces or punctuation. (Later, you can use the FREE command to display the name.)

- For step-by-step instructions on formatting, refer to *Getting Started With OS-9*.

Examples:

- To format a diskette in Drive /D1, type:

```
format /d1 ENTER
```

- To format a diskette in Drive /D1 with the name Test Disk and without prompts, type:

```
format /d1 r "test disk" ENTER
```

- To format hard Disk /H0, type:

```
tmode -pause ENTER
format /hØ ENTER
tmode pause ENTER
```

- To format a double-sided diskette in Drive /D2 with 27 cylinders and the name Database, type:

```
format /d1 2 "database" '27' ENTER
```

FREE

Syntax: **free** ***[devname]***

Function: Displays the number of unused sectors (256-byte storage areas) on a disk drive. These sectors are available for new files or for expanding existing files.

Parameters:

devname — The disk drive for which you want to display the number of free sectors.

Notes:

- The device name you specify must be a disk drive. FREE also displays the disk's name, creation date, and cluster size. If you don't specify a drive, FREE selects the drive that contains the current data directory.
- The cluster size for the Color Computer is one sector.

Examples:

- To display the number of free sectors on the current disk, type:

```
free [ENTER]
```

  The screen is similar to this:

```
"COLOR COMPUTER DISK'' created on: 83/05/28
Capacity: 630 sectors (1-sector clusters)
15 Free sectors, largest block 12 sectors
```

- To display the number of free sectors on the diskette in Drive /D1, type:

```
free /d1 [ENTER]
```

A sample screen display is:

```
"DATA DISK" created on: 83/06/16
Capacity: 630 sectors (1-sector clusters)
445 Free sectors, largest block 442 sectors
```

HELP

Syntax: **help [*command name*] [-?]**

Function: Displays the use and syntaxes of OS-9 commands.

Parameters:

command name	The command(s) for which you want help. Include as many command names as you want.
-?	Gives a list of help topics.

Notes:

- HELP uses a file named Helpmsg, which is located in the SYS directory on your system diskette.

Examples:

- To obtain help for the BACKUP command, type:

```
help backup [ENTER]
```

  The screen displays:

```
Syntax: backup [e][s][-v][dev][dev]
Usage: Copies all data from one device to
another
```

- If you try to obtain help for a non-existent command, HELP displays an error message. For instance, if you type:

```
help me [ENTER]
```

```
me: no help available
```

- You can also obtain help for the HELP command. To do so, type:

```
help help [ENTER]
```

  The screen displays:

```
Syntax: Help [subject][-?]
Usage: Give on-line help to users
       Will prompt if no subjects given
Opts: -? give list of help topics
```

IDENT

Syntax: **ident *filename [opts]***

Function: Displays header information for memory modules.

Parameters:

filename	The name of the file or module for which you want to view identification information.
opts	One or more of the following options.

Options:

-m	Causes IDENT to assume that *filename* is a module in memory
-v	Tells IDENT not to verify module cyclic redundancy check
-x	Causes IDENT to assume that *filename* is in the execution directory
-s	Displays the following module information on a single line: the edition byte (first byte after module name), the type/language byte, the module CRC and the module name. A period (.) indicates that the CRC verifies. A question mark (?) indicates that the CRC does not verify.

Notes:

- IDENT displays the module size, CRC bytes (with verification), and—for program and device driver modules—the execution offset and the permanent storage requirement bytes.

- IDENT displays and interprets the type/language and attribute revision bytes. IDENT displays the byte immediately following the module name because most Microware®-supplied modules set this byte to indicate the module edition.
- IDENT displays all modules contained in a disk file.

Examples:

- To display header information for a file named Ident that resides in computer memory, type:

```
ident -m ident [ENTER]
```

The screen might display:

```
Header for: IDENT
Module size:$06E7          #1767
Module CRC: $540BB2 (Good)
Hdr parity: $C9
Exec. off:  $0230          #573
Data Size:  $099C          #2460
Edition:          $07       #7
Ty/La At/Rv:$11 $81
Prog mod, 6809 obj, re-en, R/O
```

In the example, Hdr parity = header parity; Exec. off = execution offset; Data size = permanent storage requirements; Edition = first byte after module name; Ty/La At/Rv = type/language attribute/revision; and Prog mod, 6809 obj, re-en = module type, language, attribute.

- To display header information for the OS9Boot file,type:

```
ident /D0/OS9boot -s [ENTER]
```

The display might include:

```
17 $C0 $F2922F . OS9p2
67 $C0 $0B2322 . Init
12 $C1 $2E9EDB . IOMan
27 $D1 $B665E3 . RBF
 6 $E1 $055580 . CC3Disk
82 $F1 $FC1918 . D0
82 $F1 $9F4210 . D1
82 $F1 $E6B118 . DD
11 $D1 $10A3FA . SCF
14 $E1 $8524F1 . CC3IO
 1 $C1 $B53D94 . VDGInt
 3 $C1 $792B7E . GrfInt
83 $F1 $AB5AE5 . TERM
83 $F1 $7AB2DB . W
83 $F1 $C3E38A . W1
83 $F1 $948878 . W2
83 $F1 $36016B . W3
83 $F1 $0AE2B6 . W4
83 $F1 $123B9A . W5
83 $F1 $1CF164 . W6
83 $F1 $B71DF5 . W7
11 $E1 $C8F073 . ACIAPAK
82 $F1 $9E655D . T2
12 $E1 $CC3EA4 . PRINTER
83 $F1 $FE3BAE . P
 4 $D1 $AD6718 . PipeMan
 2 $E1 $5B2B56 . Piper
80 $F1 $CC06AF . Pipe
 9 $C1 $BE93F4 . Clock
 3 $11 $CA1F99 . CC3Go
```

Since the -s option appears in the command line, IDENT displays each module's information on a single line. In the first line of the output, for instance, 17 = edition byte (first byte after name), $C0 = type/language byte, $A366DC = CRC value, . = OK CRC check, and OS9p2 = module name.

INIZ

Syntax: **iniz *devname*** **[...]**

Function: Initializes the specified device driver.

Parameters:

devname The name of one or more devices to initialize.

Notes:

- You can use INIZ in the Startup file or at the system start-up to initialize devices and allocate their static storage at the top of memory to reduce memory fragmentation.
- INIZ attaches the specified device to OS-9, places the device address in a new device table entry, allocates the memory needed by the device driver, and calls the device driver initialization routine. INIZ does not reinitialize a device that you or the system previously installed.
- If you change the printer (/p) to a non-shareable device (a device that is not re-entrant), do not initialize it with INIZ.

Examples:

- To initialize the /P (printer) and /T2 (terminal 2) devices, type:

```
iniz p t2 ENTER
```

KILL

Syntax: kill *procID*

Function: Terminates the process specified by *procID*.

Parameters:

procID The ID number of the process to kill.

Notes:

- Unless you are the *Super User* (User Number 0), you can only terminate a process that has your user number. (Use PROCS to obtain the process ID numbers.) The Super User can terminate any process.
- If a process is waiting for I/O, you cannot cancel it until the current I/O operation terminates. Therefore, if you KILL a process and PROCS shows it still exists, it is probably waiting to receive a line of data from a terminal.
- Because KILL is a built-in shell command, it does not appear in the CMDS directory.

Examples:

- To KILL the process with the ID number 5, type:

```
kill 5 ENTER
```

- The following commands: (1) use PROCS to determine that the ID number of the process to be killed is 3, (2) terminate process 3, and (3) use PROCS to confirm that KILL has cancelled the process.

```
procs [ENTER]

          User                          Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ----------------
  2   1     0   128 128 $80    0    3 $78B2 Shell
  3   5     0   128 128 $80    0    2 $74AC Tsmon
  4   5     0   128 128 $80    0    6 $05F3 Procs
  5   0     0   128 129 $80    0    3 $6FE2 Shell

kill 3 [ENTER]
procs [ENTER]

          User                          Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ----------------
  2   1     0   128 128 $80    0    3 $78B2 Shell
  3   5     0   128 128 $80    0    6 $05F3 Procs
  5   0     0   128 129 $80    0    3 $6FE2 Shell
```

LINK

Syntax: **link *modname***

Function: Locks a previously loaded module into memory.

Parameters:

modname The name of the memory module to link.

Notes:

- If the module is not already in memory, you must use LOAD prior to using LINK. The link count of the module increases by one each time the system *links* it. Use UNLINK to *unlock* the module when you no longer need it.

Examples:

- To lock the Edit module into memory, type:

```
link edit ENTER
```

LIST

Syntax: **list *filename*** **[...]**

Function: Lists the contents of a text file or files.

Parameters:

filename — The name of the file you want to list. Include as many filenames on one line as you want, up to the maximum line length of 199 characters.

Notes:

- LIST copies text lines from a file to the standard output path. The program terminates upon reaching the end-of-file of the last input path. If you specify more than one file, LIST copies the files in the order in which you list them.
- Use LIST to examine or print text files.
- Do not LIST executable files. Doing so can cause your system to lock or crash. To view executable files, use DUMP.

Examples:

- To list the contents of the Startup file to the printer, type:

```
list /d0/startup >/p& ENTER
```

  The ampersand makes the printing job a concurrently executed task.

- To list three files to the screen, type:

```
list /d1/user5/document /d0/myfile /d0/
bob/text ENTER
```

- To copy everything you type at the keyboard to the printer, type:

```
list /term >/p ENTER
```

To go back to the standard output path (the video display) press CTRL BREAK.

- The following commands create a file called Animals, consisting of six entries. LIST, with the filename Animals as a parameter, displays the contents of the new file.

```
build animals ENTER
? cat ENTER
? cow ENTER
? dog ENTER
? elephant ENTER
? bird ENTER
? fish ENTER
? ENTER

list animals ENTER
```

The screen displays:

```
cat
cow
dog
elephant
bird
fish
```

LOAD

Syntax: **load *pathlist***

Function: Loads a module (program) from a file into memory.

Parameters:

pathlist Specifies the module to load.

Notes:

- LOAD opens the path you specify, then loads into memory one or more modules from it. The process adds the names of the new modules to the module directory. If LOAD finds that a specified module has the same name and type as a module already in memory, it keeps the module with the highest revision level.
- If the pathlist for LOAD does not include a drive name, LOAD uses the current execution directory. To LOAD a module from a directory other than the current execution directory, specify a full pathlist, beginning with a drive name if applicable.

Examples:

- In the following example, MDIR displays the names of modules currently resident in memory. Then, LOAD loads the Edit module into memory. MDIR again lists memory modules, and this time shows that Edit is successfully added to memory.

```
mdir [ENTER]
```

The screen display is similar to the following:

```
   Module Directory at 12:49:52
REL       Boot      OS9p1     OS9p2     Init
IOMan     RBF       CC3Disk   DØ        D1
DD        SCF       CC3IO     VDGInt    GrfInt
TERM      W         W1        W2        W3
W4        W5        W6        W7        ACIAPAK
T2        PRINTER   P         PipeMan   Piper
Pipe      Clock     CC3Go     CC3HDisk  HØ
Shell     Copy      Date      DeIniz    Del
Dir       Display   Echo      Iniz      Link
List      Load      MDir      Merge     Mfree
Procs     Rename    Setime    Tmode     Unlink
Basic09   GrfDrv

load edit [ENTER]
mdir [ENTER]
```

The screen displays:

```
   Module Directory at 12:51:12

REL       Boot      OS9p1     OS9p2     Init
IOMan     RBF       CC3Disk   DØ        D1
DD        SCF       CC3IO     VDGInt    GrfInt
TERM      W         W1        W2        W3
W4        W5        W6        W7        ACIAPAK
T2        PRINTER   P         PipeMan   Piper
Pipe      Clock     CC3Go     CC3HDisk  HØ
Shell     Copy      Date      DeIniz    Del
Dir       Display   Echo      Iniz      Link
List      Load      MDir      Merge     Mfree
Procs     Rename    Setime    Tmode     Unlink
Basic09   GrfDrv    Edit
```

MAKDIR

Syntax: **makdir *pathlist* or *dirname***

Function: Creates a directory according to the pathlist given. You must have write permission for the parent directory of the directory you are creating.

Parameters:

pathlist	The path to the directory you want to create.
dirname	The name of the directory you want to create.

Notes:

- When MAKDIR initializes the new directory, the directory contains only the "." and ".." files.
- MAKDIR enables all permissions for the directory it creates.
- To follow OS-9 convention, capitalize all directory names.

Examples:

- To create a directory on Drive /D1, use the directory's full pathlist from the root, such as:

```
makdir /d1/STEVE/PROJECT ENTER
```

- To create a directory called DATAFILES within the current data directory, type:

```
makdir DATAFILES ENTER
```

- To create a directory called SAVEFILES in the parent of the current directory, type:

```
makdir ../SAVEFILES ENTER
```

MDIR

Syntax: **mdir [e]**

Function: Displays the names of modules currently in memory. MDIR automatically adjusts its output for 32- or 80-column displays.

Options:

e	Causes a full listing of the extended physical address (block number and offset within the block), size, type, revision level, re-entrant attribute, user count, and name of each module. MDIR shows numbers in hexadecimal. The display adjusts for 80 or 32 columns.

Notes:

- Many of the modules displayed by MDIR are OS-9 system modules and are not executable as programs. **Always check the module type code before running a module with which you are not familiar.**

Examples:

- Because MDIR adjusts to either 32 or 80 columns, the following command produces a full module listing in either format:

```
mdir e ENTER
```

The 80-column display of MDIR e is:

```
   Module Directory at 03:03:53

Block Offset Size Typ Rev Attr  Use Module Name
----- ------ ---- --- --- ---- ---- ------------
  3F    D06   12A  C1   1 r...    0 REL
```

```
3F    E30   1D0  C1   1 r...    1 Boot
3F   1000   ED9  C0   8 r...    0 OS9p1
 1    200   CA1  C0   2 r...    1 OS9p2
 1    EA1    2E  C0   1 r...    1 Init
 1    ECF   993  C1   1 r...    1 IOMan
 1   1862  122B  D1   1 r...   6B RBF
 1   2A8D   476  E1   1 r...    2 CC3Disk
 1   2F03    30  F1   1 r...    2 D0
 1   2F33    30  F1   1 r...    0 D1
 1   2F63    30  F1   1 r...    0 DD
 1   2F93   5B6  D1   1 r...   22 SCF
 1   3549   B91  E1   1 r...    D CC3IO
 1   40DA   CE7  C1   1 r...    1 VDGInt
 1   4DC1   BF2  C1   1 r...    1 GrfInt
 1   59B3    45  F1   1 r...    8 TERM
 1   59F8    42  F1   1 r...    0 W
 1   5A3A    43  F1   1 r...    0 W1
 1   5A7D    43  F1   1 r...    0 W2
 1   5AC0    43  F1   1 r...    0 W3
 1   5B03    43  F1   1 r...    0 W4
 1   5B46    43  F1   1 r...    0 W5
 1   5B89    43  F1   1 r...    0 W6
 1   5BCC    43  F1   1 r...    5 W7
 1   5C0F   3B5  E1   1 r...    8 ACIAPAK
 1   5FC4    3F  F1   1 r...    9 T2
 1   6003   17A  E1   1 r...    D PRINTER
 1   617D    3C  F1   1 r...    D P
 1   61B9   219  D1   1 r...   12 PipeMan
 1   63D2    28  E1   1 r...   12 Piper
 1   63FA    26  F1   1 r...   12 Pipe
 1   6420   174  C1   1 r...    1 Clock
 1   6594   1AA  11   1 ....    1 CC3Go
 6      0   5F2  11   1 r...    3 Shell
 6    5F2   2DC  11   1 r...    0 Copy
 6    8CE    FD  11   1 r...    0 Date
 6    9CB    76  11   1 r...    0 DelIniz
 6    A41    A5  11   1 r...    0 Del
 6    AE6   365  11   1 r...    0 Dir
 6    E4B    84  11   1 r...    0 Display
 6    ECF    22  11   1 r...    0 Echo
 6    EF1    6A  11   1 r...    0 Iniz
 6    F5B    2C  11   1 r...    0 Link
 6    F87    4F  11   1 r...    0 List
 6    FD6    24  11   1 r...    0 Load
 6    FFA   2F1  11   1 r...    1 MDir
```

```
  6  12EB   68  11   1 r...    0 Merge
  6  1353  1EB  11   1 r...    0 Mfree
  6  153E  319  11   1 r...    0 Procs
  6  1857  11D  11   1 r...    0 Rename
  6  1974  118  11   1 r...    0 Setime
  6  1A8C  301  11   1 r...    0 Tmode
  6  1D8D   2D  11   1 r...    0 Unlink
```

- The 32-column display of MDIR is:

```
    Module Directory at 03:06:49

Blk Ofst Size Ty Rv At Uc Name
--- ---- ---- -- -- -- -- ------
  3F  D06  12A C1  1 r     0 REL
  3F  E30  1D0 C1  1 r     1 Boot
  3F 1000  ED9 C0  8 r     0 OS9p1
   1  200  CA1 C0  2 r     1 OS9p2
   1  EA1   2E C0  1 r     1 Init
   1  ECF  993 C1  1 r     1 IOMan
   1 1862 122B D1  1 r    70 RBF
   1 2A8D  476 E1  1 r     2 CC3Disk
   1 2F03   30 F1  1 r     2 D0
   1 2F33   30 F1  1 r     0 D1
   1 2F63   30 F1  1 r     0 DD
   1 2F93  5B6 D1  1 r    24 SCF
   1 3549  B91 E1  1 r     D CC3IO
   1 40DA  CE7 C1  1 r     1 VDGInt
   1 4DC1  BF2 C1  1 r     1 GrfInt
   1 59B3   45 F1  1 r     8 TERM
   1 59F8   42 F1  1 r     0 W
   1 5A3A   43 F1  1 r     0 W1
   1 5A7D   43 F1  1 r     0 W2
   1 5AC0   43 F1  1 r     0 W3
   1 5B03   43 F1  1 r     0 W4
   1 5B46   43 F1  1 r     0 W5
   1 5B89   43 F1  1 r     0 W6
   1 5BCC   43 F1  1 r     5 W7
   1 5C0F  3B5 E1  1 r     A ACIAPAK
   1 5FC4   3F F1  1 r     B T2
   1 6003  17A E1  1 r     D PRINTER
   1 617D   3C F1  1 r     D P
   1 61B9  219 D1  1 r    12 PipeMan
   1 63D2   28 E1  1 r    12 Piper
   1 63FA   26 F1  1 r    12 Pipe
   1 6420  174 C1  1 r     1 Clock
```

```
1 6594  1AA 11  1 .    1 CC3Go
6    Ø  5F2 11  1 r    3 Shell
6  5F2  2DC 11  1 r    Ø Copy
6  8CE   FD 11  1 r    Ø Date
6  9CB   76 11  1 r    Ø DEIniz
6  A41   A5 11  1 r    Ø Del
6  AE6  365 11  1 r    Ø Dir
6  E4B   84 11  1 r    Ø Display
6  ECF   22 11  1 r    Ø Echo
6  EF1   6A 11  1 r    Ø Iniz
6  F5B   2C 11  1 r    Ø Link
6  F87   4F 11  1 r    Ø List
6  FD6   24 11  1 r    Ø Load
6  FFA  2F1 11  1 r    1 MDir
6 12EB   68 11  1 r    Ø Merge
6 1353  1EB 11  1 r    Ø Mfree
6 153E  319 11  1 r    Ø Procs
6 1857  11D 11  1 r    Ø Rename
6 1974  118 11  1 r    Ø Setime
6 1A8C  301 11  1 r    Ø Tmode
6 1D8D   2D 11  1 R    Ø Unlink
```

MERGE

Syntax: **merge [*filename*][...]**

Function: Copies files to the standard output path. By redirecting the output of the MERGE command, you can combine several files into one file, or direct several files to the printer.

Parameters:

filename Specifies the files to combine.

Notes:

- Use MERGE to combine several files into a single output file. It copies data in the order in which you type the filenames.
- MERGE does not output line editing characters (such as the automatic line feed).
- You normally use MERGE with the standard output redirected to a file or device.
- You can use MERGE to append or copy any type or mixture of files to another device.

Examples:

- To merge four files into a new file called Combined.file, and send the results to the new file instead of to the video display, type:

```
merge file1 file2 file3 file4 >combined.file
ENTER
```

- To merge two files, and send the output to the printer, type:

```
merge compile.list asm.list >/P ENTER
```

MFREE

Syntax: mfree

Function: Displays a list of memory areas not presently in use and, therefore, available for assignment.

Notes:

- MFREE displays the block number, physical (extended) beginning and ending addresses, and the size of each contiguous area of unassigned RAM. It gives the size in number of blocks and in kilobytes. The block size is 8 kilobytes per block. Free memory for user data areas does not need to be contiguous because the MMU can map scattered free blocks to be logically contiguous.

Examples:

- Type this command:

  ```
  mfree ENTER
  ```

  The screen shows a display similar to this:

```
Blk Begin  End   Blks  Size
--- ----- ----- ---- -------
10  12000 10FFF    1      8K
18  18000 1DFFF    3     24K
20  20000 3FFFF   16    128K
                  ==== ======
       Total:     20    160
```

MODPATCH

Syntax: **modpatch [*options*] *filename* [*options*]**

Function: modifies modules residing in memory. MODPATCH reads a *patchfile* and executes the commands in the patchfile to change the contents of one or more modules.

Parameters:

filename	The name of a file containing instructions for MODPATCH
options	One of the following options that change MODPATCH's function

Options:

-s	Silent mode, does not display patchfile command lines as they are executed.
-w	Does not display warnings, if any
-c	Compares only, does not change the module

Notes:

- Before using MODPATCH, you must create a patchfile to supply the data to control MODPATCH's operation. This file contains single-letter commands and the appropriate module addresses. The commands are:

1 *modulename*	Link to the module specified by *modulename.*
c *offset origval rewval*	Change the byte at the offset address specified by *offset* from the value specified by *origval* to the new value specified by *newval*. If the original value does not match origval, MODPATCH displays a message.
v	Verify the module—update the modules CRC. If you plan to save the patched module to a file that the system can load, you must use this command.
m	Mask IRQ's. Turns off interrupt requests (for patching service routines).
u	Unmask IRQ's. Turns on interrupt requests (for patching service routines).

- You can use the BUILD command or any word processing program to create patchfiles.
- Module byte addresses begin at 0. MODPATCH changes values pointed to by an offset address (offset from 0) rather than an absolute memory address.

- To view the contents of a memory module, use SAVE and DUMP to copy the module to a file and display its contents. Also use SAVE to copy the patched module to a disk file.
- Changing a memory module might not produce an immediate effect. You have to duplicate the initialization procedure for that module. This means, if the module loads during bootup, you have to create a new boot file that includes the changed module, then reboot using the new boot file.
- To use the patched module in future system boots, use SAVE to store the module in the MODULES directory of your system disk. You can then use OS9GEN to create a new system disk using the patched module. If you are using the patched module to replace another module, rename the original module and then give the patched module the original name.
- If you patch a module that is loaded during the system boot, you can use COBBLER to make a new system boot that uses the patched module.

Examples:

The following example shows the commands, the screen prompts, and the entries you make to patch the standard 40-column term window descriptor to be an 80-column screen rather than the standard 40-column screen:

```
OS9:build termpatch [ENTER]
? I term [ENTER]
? c 002c 28 50 [ENTER]
? c 0030 01 02
? v [ENTER]
? [ENTER]
OS9: modpatch termpatch [ENTER]
```

To change the size, columns, and colors of Device Window W1, create the following procedure file and name it W180:

```
I w1
c 0030 01 02
c 002c 1b 50
c 002d 0b 18
```

If the W1 module is not already in memory, load it from the MODULES directory of your system disk. Then, before initializing W1, run MODPATCH:

```
modpatch w180 [ENTER]
```

Next, initialize W1:

```
iniz w1 [ENTER]
shell i=/w1& [ENTER]
```

Press [CLEAR] to display the new window with 80 columns, 24 lines, and a white background.

MONTYPE

Syntax: **montype** *type*

Function: Sets your system for the type of monitor you are using

Parameters:

Parameters:

type A single letter indicating the monitor type:

- c for composite monitors or color televisions
- r for RGB monitors
- m for monochrome monitors or black and white televisions

Notes:

- Different types of color monitors display colors differently. For the best results, set your system to the type of monitor you are using.
- If you are using a monochrome monitor or black and white television, you can obtain a sharper image by setting your monitor type to monochrome.
- Include the MONTYPE command in your system's Startup file to automatically boot in the proper monitor mode.
- If you do not use MONTYPE, the system defaults to c (composite monitor).

Example:

To set your system for an RGB monitor, type:

```
montype r ENTER
```

To add a MONTYPE command to your existing Startup file, first use BUILD to create the new command. For example:

```
build temp [ENTER]
montype r [ENTER]
[ENTER]
```

Next, append the file to Startup. Type:

```
merge startup temp > startup.new [ENTER]
```

Delete the temp file:

```
del temp [ENTER]
```

To enable the system to use Startup.new when booting, rename the original Startup file:

```
rename Startup Startup.old
```

Then rename Startup.new:

```
rename Startup.new Startup
```

OS9GEN

Syntax: **os9gen *devname* [*opts*]**

Function: Creates and links the required OS9Boot file to a diskette making it a bootable diskette.

Parameters:

devname	The disk drive containing the diskette to receive the new boot file.
opts	One or more of the following options.

Options:

-s	Causes OS9GEN to use only one drive to generate the boot file. In a single-drive operation, OS9GEN reads the modules from the source diskette and asks you to exchange diskettes and press [C] as it reads and copies the modules.
#*n*[K]	reserves *n* kilobytes of memory for use by the OS9GEN command. By setting aside as much memory as possible, you can increase the speed of OS9GEN and, on single-drive systems, reduce the number of diskette swaps. If you type K after #*n*, the memory specified by *n* is in kilobytes (1024 bytes), otherwise *n* is in 256-byte pages.

Notes:

- OS9Boot files can only exist on contiguous sectors. Therefore, use OS9GEN only with newly formatted diskettes. If OS9Boot is fragmented, the system warns you not to use the diskette to bootstrap OS-9.

- OS9GEN creates a working file called Tempboot on the device specified by *devname*. Next, it reads filenames (pathlists) either from the keyboard (the standard input path) or redirected from a file. If you enter names manually, OS9GEN does not display a prompt. Type each filename and press [ENTER]. After typing the last filename and pressing [ENTER], press [ENTER] again, or press [CTRL][BREAK] to complete the list.

  OS9GEN opens each file and copies it to Tempboot. The process repeats until it reaches a blank line or an end-of-file marker. All of the modules listed in Chapter 5 are not required in a boot file. These modules must be included in a boot File:

  OS9p2, Init, IOMan, RBF, SCF, CC3IO, VDGInt (or GrfInt), CC3Disk, D0, TERM, Clock, CC3G0.

- You must have RENAME in the current execution directory or in memory for OS9GEN to work properly.

- With all input files copied to Tempboot, OS9GEN deletes the OS9Boot file, if it exists. It renames Tempboot as OS9Boot, and writes the file's starting address and size in the diskette's Identification Sector (LSN 0) for use by the OS-9 bootstrap firmware. OS-9 writes its kernel on diskette Track 34. If there is not room for the kernel, an error message appears, and the operation terminates.

- If you have only one drive, you can generate a new boot file more easily using the CONFIG utility. CONFIG is designed to make custom system diskettes using either single- or multiple-drives.

Examples:

- The following commands manually install a boot file on device /D1 that is an exact copy of the OS9Boot file on device /D0. The first command line runs OS9GEN, the second enters the name of the file to install, and the third enters an end-of-file marker.

```
os9gen /d1 [ENTER]
/d0/os9boot [ENTER]
[CTRL][BREAK]
```

- The following commands let you manually install a boot file on device /D1 that is a copy of the OS9Boot file on device /D0 and the modules stored in the files /D0/Tape.driver and /D2/Video.driver. Line 1 executes OS9GEN. Line 2 enters the main boot filename. Lines 3 and 4 enter the names of the two additional files, and Line 5 enters an end-of-file marker.

```
os9gen /d1 [ENTER]
/d0/os9boot [ENTER]
/d0/tape.driver [ENTER]
/d2/video.driver [ENTER]
[CTRL][BREAK]
```

- The following commands generate a new boot file on Drive /D1 that includes all the old boot file modules. Line 1 uses BUILD to create a file called Bootlist. The next three lines enter the names of the three files into Bootlist. Line 5 terminates BUILD, and Line 6 runs OS9GEN with input redirected from the new Bootlist file.

```
build /d0/bootlist [ENTER]
? /d0/os9boot [ENTER]
? /d0/tape.driver [ENTER]
? /d0/video.driver [ENTER]
? [ENTER]
os9gen /d1</d0/bootlist [ENTER]
```

- To install a custom boot file on a single-drive system, build a Bootlist to drive the OS9GEN program. You need a directory that contains the required file managers, device drivers, descriptors, and other files for the boot file. For example, to make a new boot file containing only the /TERM, /D0, /D1, and /P devices, first build a Bootlist such as:

```
build /d0/bootlist [ENTER]
? term_vdg.dt [ENTER]
? p.dd [ENTER]
? d0_35s.dd [ENTER]
? d1_35s.dd [ENTER]
? os9p2 [ENTER]
? Init [ENTER]
? IOMan [ENTER]
? RBF.mn [ENTER]
? CC3Disk.dr [ENTER]
? SCF.mn [ENTER]
? CC3IO.dr [ENTER]
? vdgint.io [ENTER]
? printer.dr [ENTER]
? clock.60hz [ENTER]
? cc3go [ENTER]
```

Then use OS9GEN to create the new boot file on a separate diskette by typing:

```
os9gen /d0 -s #25K </d0/bootlist [ENTER]
```

This command causes OS9GEN to use only one drive, 25K of buffer space, and the filenames previously stored in the Bootlist file.

You can expand this basic bootlist file to include other standard OS-9 modules such as window device descriptors, other disk drivers, descriptors, and terminal or modem descriptors.

All of the standard bootlist modules are contained in the MODULES directory on the BASIC09/CONFIG diskette.

PROCS

Syntax: **procs [e]**

Function: Displays a list of the processes running on the system. PROCS automatically adjusts its output for 32-or 80-column displays.

Options:

e	Causes PROCS to display the processes of all users.

Notes:

- Normally PROCS lists only processes having the user's ID. The list is a *snapshot* taken at the instant PROCS executes. Processes switch states rapidly, usually many times per second.
- PROCS shows the user and process ID numbers, priority, state (process status), memory size (in 256 byte pages), primary program module, and standard input path.
- PROCS adjusts its output for 80 or 32 columns.

Examples:

- Because PROCS automatically adjusts for either 32- or 80-column displays, the following command can produce either format:

```
procs e ENTER
```

Following is a possible 32-column display of PROCS:

```
Id  PId  User#  Pty  Age  Sta
 Sigl  Mem     StPtr    Primary
==============================
  2   1      Ø   128 128 $8Ø
    Ø     3     $78E2   Shell
  3   6      Ø   128 128 $8Ø
    Ø    16     $74B2   Basic09
  4   2      Ø   128 128 $8Ø
    Ø     6     $Ø5F3   Procs
  5   Ø      Ø   128 128 $8Ø
    Ø     3     $6FB2   Shell
  6   Ø      Ø   128 129 $8Ø
    Ø     3     $68E2   Shell
```

Following is a possible 80-column display of PROCS:

```
          User                      Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ---------------
  2   1     Ø   128 128 $8Ø    Ø   3 $78B2 Shell
  3   6     Ø   128 128 $8Ø    Ø  16 $74B2 Basic09
  4   5     Ø   128 128 $8Ø    Ø   3 $72E2 Shell
  5   Ø     Ø   128 129 $8Ø    Ø   3 $6FB2 Shell
  6   Ø     Ø   128 129 $8Ø    Ø   3 $68E2 Shell
  7   4     Ø   128 128 $8Ø    Ø   6 $Ø5F3 Procs
```

PWD
PXD

Syntax: **pwd**
pxd

Function: PWD shows the path from the ROOT directory to the current data directory. PXD shows the path from the ROOT directory to the current execution directory.

Notes:

- OS-9 keeps a current data directory and current execution directory for each process. Use PWD and PXD to show where your current data and execution directories are located on the disk or disks you are using.

Examples:

- The following example uses a full pathlist. CHD changes the current data directory to the MANUALS directory.

  ```
  chd /d1/steve/textfiles/manuals ENTER
  ```

  To display the full path to the data directory, type:

  ```
  pwd ENTER
  ```

  The screen displays the data directory path:

  ```
  /D1/STEVE/TEXTFILES/MANUALS
  ```

- The following commands cause the current data directory to move up one level in the directory hierarchy and then display the data directory.

  ```
  chd .. ENTER
  pwd ENTER

  /D1/STEVE/TEXTFILES
  ```

- The following commands change the current data directory to the parent directory and then display the current data directory.

```
chd .. ENTER
pwd ENTER

/D1/STEVE
```

- The following command displays the current execution directory, CMDS.

```
pxd ENTER

/DØ/CMDS
```

RENAME

Syntax: **rename *pathlist filename***

Function: Gives the specified file or directory a new name.

Parameters:

pathlist	The current name of the file or directory.
filename	The new name.

Notes:

- You must have write permission for the file.

Examples:

- To change a file's name from Blue to Purple, type:

```
rename blue purple [ENTER]
```

- To rename a file in the USER9 directory of Drive /D3, type:

```
rename /d3/user9/test temp [ENTER]
```

- In the following example, DIR displays the names of the files in the current data directory. RENAME changes the filename Animals to Mammals. Another DIR command shows that RENAME has performed properly.

```
dir [ENTER]
```

The screen displays:

```
Directory of . 16:22:53
myfile      animals

rename animals mammals [ENTER]
dir [ENTER]
```

The screen now shows:

```
Directory of . 16:23:22
myfile      mammals
```

SETIME

Syntax: **setime [*yy/mm/dd hh:mm*[:*ss*]]**

Function: Sets the system date and time, and activates the real time clock.

Parameters:

yy	The year in a two-digit format (86 for 1986).
mm	The month in a one or two-digit format (01 or 1 for January, 12 for December).
dd	The day of the month in a one- or two-digit format, such as 21.
hh	The hour in a one- or two-digit, 24-hour format (15 for 3 p.m.).
mm	Minutes in a one- or two-digit format, such as 03, 5, or 55.
ss	Seconds in a one- or two-digit format, such as 04, 5, or 25.

Options:

Specifying seconds in the new time entry is optional.

Notes:

- You can include the date and time parameters. If you do not, SETIME asks you for them.
- Numbers are one- or two- decimal digits using the space, colon, semicolon, or slash as delimiters.
- The CC3go module starts the clock on system startup, so multitasking is possible without use of the SETIME utility.

- If you do not set the date and time when booting OS-9, the system cannot accurately update the "Last modified" date and time for files.

Examples:

- To set the date and time to August 15, 1986, 3:45 p.m., type:

```
setime 86,08,15,15,45 ENTER
```

- To set the same date using a slightly different but equally acceptable format, type:

```
setime 86/08/15 15/45/00 ENTER
```

SETPR

Syntax: **setpr *procID number***

Function: Changes the CPU priority of a process. The priority of a process determines the CPU time allotted to it under multi-tasking conditions.

Parameters:

procID	The number of the process for which you want to change the priority.
number	The new priority number.

Notes:

- The process priority number is a decimal number in the range 1 (lowest priority) to 255. If you need information about the process ID number and current priority, use PROCS.
- You can use SETPR only on processes that have your user number.
- SETPR does not appear in the CMDS directory because it is built into the shell.
- A Super User (User 0) can set any process priorities.

Examples:

- To set or change the priority of Process 8 to 250, type:

```
setpr 8 250 ENTER
```

- In the following commands PROCS displays process ID numbers and other information. Then, SETPR sets Process 3 to a priority of 255. The final command confirms the change.

```
procs [ENTER]
```

Following is a sample screen display:

```
          User                          Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ---------------
  2   1     0   128 128 $80    0    3 $78E2 Shell
  3   6     0   128 128 $80    0   16 $74B2 Basic09
  4   2     0   128 128 $80    0    6 $05F3 Procs
  5   0     0   128 128 $80    0    3 $6FB2 Shell
  6   0     0   128 129 $80    0    3 $68E2 Shell

setpr 3 255 [ENTER]
procs [ENTER]

          User                          Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ---------------
  2   1     0   128 128 $80    0    3 $78B2 Shell
  3   6     0   255 128 $80    0   16 $74B2 Basic09
  4   5     0   128 128 $80    0    3 $72E2 Shell
  5   0     0   128 129 $80    0    3 $6FB2 Shell
  6   0     0   128 129 $80    0    3 $68E2 Shell
  7   4     0   128 128 $80    0    6 $05F3 Procs
```

SHELL

Syntax: **shell *arglist***

Function: The shell is OS-9's command interpreter program. It reads data from its standard input path, processes it and sends the output to its standard output path, and sends error messages (and some prompts) via the standard error output. Any or all of these paths may be redirected. It interprets the data as a sequence of commands. The function of the shell is to initiate and control execution of other OS-9 programs.

Parameters:

arglist	The commands, parameters, and options given SHELL in a command line.

Notes:

- The shell reads and interprets one text line at a time from the standard input path until it reaches an end-of-file marker. At that time it terminates itself.
- When another program calls the shell, a special case occurs in which the shell takes the argument list as its first line of input. If this command line consists of *built-in* commands only, the shell reads and processes more lines. Otherwise, control returns to the calling program after the shell processes the single command line.
- When operating from the shell, you do not need to specify the SHELL command to execute a program, a command, or a built-in shell function. Using SHELL before a command causes the existing shell to fork an additional shell, which then forks the specified process, such as:

  ```
  shell dir e ENTER
  ```

  Issuing a command without SHELL causes the existing shell to fork the specified process, such as:

  ```
  dir e ENTER
  ```

The following two commands also have identical effects:

```
shell x ENTER
```

```
x ENTER
```

- The shell command separators are:

;	Sequential execution separator
&	Concurrent execution separator
!	Pipeline separator
ENTER	end-of-line (sequential execution separator)

- The Shell command modifiers are:

<	Redirect standard input
>	Redirect standard output
>>	Redirect standard error output
<>	Redirects standard input and standard output
<>>	Redirects standard input and standard error output
>>>	Redirects standard output and standard error output
<>>>	Redirects standard input, standard output and standard error output
#n	Set the process memory size in pages
*#n*K	Set the program memory size in 1 kilobyte units.

- The following built-in Shell command parameters tell OS-9 to:

chd *pathlist*	Change the data directory
kill *procID*	Send the termination signal to process
setpr *procID number*	Change the specified process priority

chx *pathlist*	Change the execution directory
i=*devicename*	Create an immortal process
w	Wait for any process to die
p	Turn on prompting
-p	Turn off prompting
t	Echo input lines to standard output
-t	Not echo input lines
-x	Not terminate on an error
x	Terminate on error
*	Not process the following text

- See Chapter 3 for more information on the operation of the shell.

TMODE

Syntax: **tmode [*pathnum*] [*paramlist*] [...]**

Function: Displays or changes the initialization parameters of the terminal. TMODE automatically adjusts its output for 32- or 80-column displays.

Common uses include changing baud rates and control key definitions.

Parameters:

pathnum — One of the standard path numbers:

.0 = standard input path
.1 = standard output path
.2 = standard error output path

paramlist — One of the following options.

Options:

upc — Displays uppercase characters only. Lowercase characters automatically convert to uppercase.

-upc — Displays both upper- and lowercase characters.

bsb — Causes a backspace to erase characters. Backspace characters echo as a backspace-space-backspace sequence. This setting is the system default.

-bsb — Causes backspace not to erase. Only a single backspace echoes.

bsl — Enables *backspace over a line*. Deletes lines by sending backspace-space-backspace sequences to erase a line (for video terminals). This setting is the system default.

-bsl	Disables *backspace over a line*. To delete a line, TMODE prints a *new line* sequence (for hard-copy terminals).
echo	Input characters *echo* on the terminal. This setting is the system default.
-echo	Turns off the echo default.
lf	Turns on the auto line feed function. Line feeds automatically echo to the terminal on input and output carriage returns. The auto line feed setting is the system default.
-lf	Turns off the auto line feed default.
null=*n*	Sets the null count—the number of null ($00) characters transmitted after carriage returns for the return delay. The value *n* is in decimal. The default is 0.
pause	Turns on the screen pause. This setting suspends output when the screen fills. See the *pag* parameter for a definition of screen size. Resume output by pressing the space bar. This setting is the system default.
-pause	Turns off the screen pause mode.
pag=*n*	Sets the length of the video display page to *n* (decimal) lines. This setting affects the *pause* mode.
bsp=*h*	Sets the backspace character for input. The value *h* is in hexadecimal. The default is 08.
del=*h*	Sets the delete line character for input. The value *h* is in hexadecimal. The default is 18.
eor=*h*	Sets the end-of-record (carriage return) character for input. This setting requires a value in hexadecimal. The default is 0D.
eof=*h*	Sets the end-of-file character for input. The value *h* is in hexadecimal. The default is 1B.

reprint=*h* Sets the reprint line character. The value *h* is in hexadecimal.

dup=*h* Sets the character to duplicate the last input line. The value *h* is in hexadecimal. The default is 01.

psc=*h* Sets the pause character. The value of the character is in hexadecimal. The default is 17.

abort=*h* Sets the terminate character (normally CONTROL C). The value of the character is in hexadecimal.

quit=*h* Sets the quit character (normally CONTROL E). The value of the character is in hexadecimal.

bse=*h* Sets the backspace character for output. The value *h* is in hexadecimal. The default is 08.

bell=*h* Sets the bell (alert) character for output. The value *h* is in hexadecimal. The default is 07.

type=*h* For external devices, use type for ACIA (asynchronous communications interface adapter) initialization values (hexadecimal). The default is 00. Bits 5-7 set either MARK, SPACE, or no parity on all devices. Codes for these are:

000 = no parity

101 = MARK parity transmitted, no checking

111 = SPACE parity transmitted, no checking

011 = even parity (available only with the external ACIA pak and Modpak devices)

001 = odd parity (available only with the external ACIA pak and Modpak devices)

Bit 4 selects auto-answer modem support features.

1 = on

0 = off

See "Technical Information for the RS232 Port" in Chapter 5 for more information.

For TERM-VDG, the type byte has a different use:

Bit 0 specifies a machine with true lowercase capability. Set Bit 0 to turn on true lowercase.

For TERM-WIN, use a value of 80 to specify a window device.

xon=*h* Sets the character to be used as a signal for resuming transmission of data after an xoff signal is received. Default is 0 (not active).

xoff=*h* Sets the character to be used for stopping data transmission. Default is 0 (not active).

baud=*h* Sets the baud rate, word length, and stop bits for a software-controllable interface. The codes for the baud rate are:

0=110 3=1200 6= 9600
1=300 4=2400 7=19200 (ACIAPAK only)
2=600 5=4800 7=32000 (SIO only)

Bits 0-3 determine the baud rate.
Bit 4 is reserved for future use.
Bits 5-6 determine the word length:
00 = 8 bits
01 = 7 bits
Bit 7 determines the number of stop bits:
0 = 1 stop bit
1 = 2 stop bits

See "Technical Information for the RS232 Port" in Chapter 5 for further information.

Notes:

- You can specify any number of parameters from the options list, separating them by spaces or commas. If you don't specify parameters, TMODE displays the current values of the available options.
- You can use a period and a number to specify the pathnumber on which to read or set options. If you don't specify a path, TMODE affects the standard input path.
- TMODE works only if a path to the file/device is open. Use XMODE to alter device descriptors and set device initial operating parameters.
- TMODE can also alter the baud rate, word length, stop bits, and parity for devices already initialized.
- If you use TMODE in a procedure file, you must specify one of the standard output paths (.1 or .2). This procedure is necessary, because the command redirects the SHELL's standard input path to come from a disk file. (You can use TMODE only on SCFMAN-type devices.) For example, to set lines per page for standard output, use this line:

```
TMODE .1 pag=24 [ENTER]
```

Examples:

- The following command line sets the terminal to display upper- and lowercase, sets the null count to 4, and turns on the screen pause function.

```
tmode -upc lf null=4 pause [ENTER]
```

- The next command sets the screen page length (number of lines) to 24, turns on the screen pause function and the backspace-over-line function, and sets the backspace character value to 8 and turns off the echo default.

```
tmode pag=24 pause bsl -echo bsp=8 [ENTER]
```

TUNEPORT

Syntax: **tuneport [*device*] [-s = *value*]**

Function: Lets you test and set delay loop values for the current baud rate and select the best value for your printer or terminal.

Parameters:

device	The device you want to test, either your printer (/p) or terminal (/t1).
value	A new delay loop value.

Options:

-s=	Sets a new delay loop value.

Examples:

- The following command provides a test operation for your printer.

  ```
  tuneport /p ENTER
  ```

  After a short delay, TUNEPORT displays the current baud rate and sends data to the printer to see if it is working properly. The program then displays the current delay value and asks for a new value. Enter a decimal delay value and press ENTER. Again, TUNEPORT sends data to the printer as a test. Continue this process until you find the best value. When you are satisfied, press ENTER instead of entering a value at the prompt. A closing message displays your new value.

  Use the same process to set a new delay loop value for the /T1 terminal.

- The following command line sets the delay loop value for your printer to 255.

```
tuneport /p -s=255 ENTER
```

Use such a command on future system boots to set the optimum delay value determined with the TUNEPORT test function. Then, using OS9GEN or COBBLER, generate a new boot file for your system diskette. You can also use the -s option with TUNEPORT in your system Startup file to set the value.

UNLINK

Syntax: **unlink *modname*** **[...]**

Function: Tells OS-9 that the named memory module(s) is no longer needed by the user.

Parameters:

modname One or more modules you want to unlink.

Options:

In one command line, you can specify as many modules as you want to unlink.

Notes:

- Whether OS-9 destroys the modules and reassigns their memory depends on whether the module is in use by other processes. Each process using a module increases its link-count by one. Each UNLINK you issue decreases its link-count by 1. When the link-count reaches 0, OS-9 deallocates the module.
- You should unlink modules whenever possible to make most efficient use of available memory resources. Modules you have loaded and linked might need to be unlinked twice to remove them from memory.

- **Warning:** Never attempt to unlink a module you didn't load or link, and never unlink a module that is in use by programs (displayed by the PROCS command).

Examples:

- To unlink three modules named Pgm1, Pgm5, and Pgm99, type:

```
unlink pgm1 pgm5 pgm99 [ENTER]
```

- In the following command sequence, MDIR first displays the modules in memory. The next command unlinks the edit module. The output of the final command (MDIR) shows that UNLINK is successful—Edit no longer appears on the list.

```
mdir [ENTER]
```

A possible screen display is:

```
   Module Directory at 00:01:08
REL         Boot        OS9p1       OS9p2       Init
IOMan       RBF         CC3Disk     D0          D1
DD          SCF         CC3IO       VDGInt      GrfInt
TERM        W           W1          W2          W3
W4          W5          W6          W7          ACIAPAK
T2          PRINTER     P           PipeMan     Piper
Pipe        Clock       CC3Go       CC3HDisk    H0
Shell       Copy        Date        DEIniz      Del
Dir         Display     Echo        Iniz        Link
List        Load        MDir        Merge       Mfree
Procs       Rename      Setime      Tmode       Unlink
Basic09     GrfDrv      Edit

unlink edit [ENTER]
mdir [ENTER]
```

The new screen display is:

```
   Module Directory at 00:03:15
REL         Boot        OS9p1       OS9p2       Init
IOMan       RBF         CC3Disk     D0          D1
DD          SCF         CC3IO       VDGInt      GrfInt
TERM        W           W1          W2          W3
W4          W5          W6          W7          ACIAPAK
T2          PRINTER     P           PipeMan     Piper
Pipe        Clock       CC3Go       CC3HDisk    H0
Shell       Copy        Date        DeIniz      Del
Dir         Display     Echo        Iniz        Link
List        Load        MDir        Merge       Mfree
Procs       Rename      Setime      Tmode       Unlink
Basic09     GrfDrv
```

WCREATE

Syntax: **wcreate** ***/wpath*** **[-s=*type*]** ***xpos ypos xsize ysize foreground background*** **[*border*]**

Function: Initializes and creates a window.

Parameters:

/wpath	The window device name of the window you are creating (W, W1, W2, W3, and so on).
xpos	The x co-ordinate (in decimal) for the starting position of the upper left corner of the screen.
ypos	The y co-ordinate (in decimal) for the starting position of the upper left corner of the screen.
xsize	The horizontal size of the screen in columns; 1 to 80 (in decimal) for screen types 2, 5, and 7, and 1 to 40 (decimal) for screen types 1, 6, and 8.
ysize	The vertical size of the screen in lines, in the range 1 to 24 (in decimal).
foreground	The window foreground color.
background	The window background color.
border	An optional window border color. The default is black.

Options:

-s=*type* The screen type, chosen from the following list:

Type	Description
1 =	40-column hardware text screen
2 =	80-column hardware text screen
5 =	640 x 192 two-color screen
6 =	320 x 192 four-color screen

7 = 640 x 192 four-color screen
8 = 320 x 192 sixteen-color screen

If you use the -s=*type* option, you must specify a border color in the command line. The -s option is only used to create a window on a new screen. When creating additional windows on the currently displayed screen, omit the -s and border color options.

-z Directs WCREATE to accept input from the standard input (redirected from a file).

-? Produces a help message for the command.

Examples:

- To create a full screen, 80-column text window on /w1, type:

```
wcreate /w1 -s=2 0 0 80 24 7 4 1 ENTER
```

- To create two windows (/w2 and /w3) on a 640 x 192 graphics screen in which /w2 is the upper left of the display and /w3 is the right half of the display, first use build to create an input file:

```
build wfile ENTER
? /w2 -s=07 0 0 40 12 7 4 1 ENTER
? /w3 40 0 40 24 4 7 ENTER
? ENTER
```

Then, create the windows using Wfile as input:

```
wcreate -z < wfile ENTER
```

- You can use the -z option to create windows in your system startup file. For example, the following startup file sets up several windows, along with the usual SETIME.

```
* lock the shell in memory and set the time
  link shell
setime < /1

* create the new windows
wcreate -z
* set up an 80-column full window for /w1
/w1 -s=2 0 0 80 24 7 4 1
* set up a 40 column full window for /w2
/w2 -s=1 0 0 40 24 7 4 1
* set up /w3 and /w4 as halves of a
*640 x 192 display
/w3 -s=7 0 0 40 24 7 4 1
/w4 40 0 40 24 4 7
* the following blank line terminates input
* from wcreate

* get the graphics fonts loaded
merge sys/stdfonts > /w1
```

Now, when the system boots, it has four windows defined, besides TERM. As shown, you can use an asterisk as the first character on a line in order to allow comments in the file.

XMODE

Syntax: **xmode *devname* [*paramlist*]**

Function: Displays or changes the initialization parameters of any SCF-type device such as the video display, printer, RS-232 port, and others. XMODE automatically adjusts its output for 32- or 80-column displays.

Common uses include changing baud rates and control key definitions.

Parameters:

pathnum	The device name to change, such as /term, /w7, /t2, and so on.
paramlist	One of the following options.

Options:

upc	Displays uppercase characters only. Lowercase characters automatically convert to uppercase.
-upc	Displays both upper- and lowercase characters.
bsb	Causes a backspace to erase characters. Backspace characters echo as a backspace-space-backspace sequence. This setting is the system default.
-bsb	Causes backspace not to erase. Only a single backspace echoes.
bsl	Enables *backspace over a line*. Deletes lines by sending backspace-space-backspace sequences to erase a line (for video terminals). This setting is the system default.
-bsl	Disables *backspace over a line*. To delete a line, you must print a *new line* sequence (for hard-copy terminals).

echo	Input characters *echo* on the terminal. This setting is the system default.
-echo	Turns off the echo default.
lf	Turns on the auto line feed function. Line feeds automatically echo to the terminal on input, and they output carriage returns. The auto line feed setting is the system default.
-lf	Turns off the auto line feed default.
null = *n*	Sets the null count—the number of null ($00) characters transmitted after carriage returns for the return delay. The value *n* is in decimal. The default is 0.
pause	Turns on the screen pause. This setting suspends output when the screen fills. See the *pag* parameter for a definition of screen size. Resume output by pressing the space bar. This setting is the system default.
-pause	Turns off the screen pause mode.
pag = *n*	Sets the length of the video display page to *n* (decimal) lines. This setting affects the *pause* mode.
bsp = *h*	Sets the backspace character for input. The value *h* is in hexadecimal. The default is 08.
del = *h*	Sets the delete line character for input. The value *h* is in hexadecimal. The default is 18.
eor = *h*	Sets the end-of-record (carriage return) character for input. This setting requires a value in hexadecimal. The default is 0D.
eof = *h*	Sets the end-of-file character for input. The value *h* is in hexadecimal. The default is 1B.
reprint = *h*	Sets the reprint line character. The value *h* is in hexadecimal.
dup = *h*	Sets the character to duplicate the last input line. The value *h* is in hexadecimal. The default is 01.

psc=*h* Sets the pause character. The value of the character is in hexadecimal. The default is 17.

abort=*h* Sets the terminate character (normally CONTROL C). The value of the character is in hexadecimal.

quit=*h* Sets the quit character (normally CONTROL E). The value of the character is in hexadecimal.

bse=*h* Sets the backspace character for output. The value *h* is in hexadecimal. The default is 08.

bell=*h* Sets the bell (alert) character for output. The value *h* is in hexadecimal. The default is 07.

type=*h* For external devices, use type for ACIA (asynchronous communications interface adapter) initialization values (hexadecimal). The default is 00. Bits 5-7 set either MARK, SPACE, or no parity on all devices. Codes for these are:

000 = no parity

101 = MARK parity transmitted, no checking

111 = SPACE parity transmitted, no checking

011 = even parity (available only with the external ACIA pak and Modpak devices)

001 = odd parity (available only with the external ACIA pak and Modpak devices)

Bit 4 selects auto-answer modem support features.

1 = on

0 = off

See "Technical Information for the RS232 Port" in Chapter 5 for more information.

For TERM-VDG, the type byte has a different use:

> Bit 0 specifies a machine with true lowercase capability. Set Bit 0 to turn on true lowercase.

For TERM-WIN, use a value of 80 to specify a window device.

baud=*h* Sets the baud rate, word length, and stop bits for a software-controllable interface. The codes for the baud rate are:

0=110	3=1200	6= 9600
1=300	4=2400	7=19200 (ACIAPAK only)
3=600	5=4800	7=32000 (SIO only)

Bits 0-3 determine the baud rate
Bit 4 is reserved for future use
Bits 5-6 determine the word length:
  00 = 8 bits
  01 = 7 bits
Bit 7 determines the number of stop bits:
  0 = 1 stop bit
  1 = 2 stop bits.

See "Technical Information for the RS232 Port" in Chapter 5 for further information.

xon=*h* Sets the character to be used as a signal for resuming transmission of data after an xoff signal is received. Default is 0 (not active).

xoff=*h* Sets the character to be used for stopping data transmission. Default is 0 (not active).

Notes:

- XMODE is similar to TMODE, but there are differences. TMODE operates only on open paths, so its effect is temporary. XMODE updates the device descriptor. Its change persists as long as the computer is running, even if you or the system repeatedly open and close the paths to the device.
- If you use XMODE to change parameters and the COBBLER program to make a new system diskette or to remake the boot tracks on the current system diskette, the process permanently changes the parameters on the new system diskette.
- XMODE requires that you specify a device name. If you do not specify parameters, XMODE displays the present value for each parameter. You can use any number of parameters, separating them with spaces or commas.

Examples:

- The following command sets the term (video) for upper- and lowercase, the null count to 4, the backspace character value to 1F hexadecimal, and turns on the screen pause function.

```
xmode /term -upc null=4 bse=1F pause ENTER
```

Chapter 7

Macro Text Editor

Overview

The OS-9 Macro Text Editor is a powerful, easy-to-learn text-preparation system. Use it to prepare text for letters and documents or text to be used by other OS-9 programs, such as the assembler and high-level languages. The text editor includes the following features:

- Compact size
- Capability of having multiple read and write files open at the same time
- All OS-9 commands usable inside the text editor
- Adjustable workspace size
- Repeatable command sequences
- Edit macros (special utility functions)
- Multiple text buffers
- Powerful commands

The Macro Text Editor is about 5 kilobytes in size and requires at least 2K bytes of free RAM to run.

Text Buffers

As you enter text, the editor places it in a temporary storage area called a text buffer. A text buffer acts as a scratch pad for saving text that you can manipulate with various edit commands. The Macro Text Editor can use multiple text buffers, one at a time.

A buffer in use is called the *edit buffer*. Edit also has another default buffer called the *secondary buffer*. As well, you can create additional buffers up to the capacity of your computer's memory.

Edit Pointers

The Macro Text Editor has an edit *pointer* that identifies your position in the buffer, in a manner similar to holding your place in a book with your finger.

The pointer is invisible to you, but Edit commands can reposition it and display the text to which it points. Each buffer has its own edit pointer, and you can move from buffer to buffer without losing your place in any of them.

Entering Commands

The Macro Text Editor is interactive. This means you and the editor carry on a two-way conversation. You issue a command, and the editor carries out the command and displays the result. When you are through making changes, you can save your edited file, then press [Q] [ENTER] to quit editing.

When the editor displays `E:` on the screen, it is waiting for you to type a command. You type a line that includes one or more commands, then press [ENTER]. Edit carries out the commands and again displays `E:`.

If you enter more than one command on a line, separate the commands with a space. If, however, a space is the first character on a line, the editor considers the space to be an insert command and not a separator.

Correct a typing error by backspacing over it or by deleting the entire line. Note, you cannot correct a line after pressing [ENTER].

Control Keys

You can use the same special control keys with Edit that you use with OS-9. See Appendix D for a complete listing of these keys. Following is a list of some of the control keys that are especially useful with Edit:

Control Key(s)	Function
[CTRL] [A]	Repeats the previous input line.
[CTRL] [C]	Terminates the editor and returns to command entry mode.
[CTRL] [D]	Displays the current input on the next line.
[CTRL] [H] or [←]	Backspaces and erases the previous character.

Control Key(s)	Function
[Q] [ENTER]	Interrupts the editor and returns to command entry mode.
[CTRL] [W]	Temporarily halts the data output to your terminal so that you can read the screen before the data scrolls off. Output resumes when you press any other key.
[CTRL] [X] or [SHIFT] [←]	Deletes the line.
[CTRL] [BREAK]	Terminates the editor, and returns to command entry mode.

Command Parameters

There are two types of edit parameters, "numeric" and "string."

Numeric Parameters. Numeric parameters specify an amount, such as the number of times to repeat a command or the number of lines affected by a command. If you do not specify a numeric parameter, the editor uses the default value of one. Specify all other numeric parameters in one of the following ways.

- Enter a positive decimal integer in the range 0 to 65,535. For example:

  0
  10
  5250
  65532
  31

- Enter an asterisk (*) as a shorthand for *all* (all the way to the beginning, all the way to the end, all of the lines, and so on). To the editor, an asterisk means infinity. Use the asterisk to specify all remaining lines, all characters, or repeat forever.

- Use a numeric variable. (See "Parameter Passing" later in this chapter.)

String Parameters. String parameters specify a single character, group of characters, word, or phrase. Specify string parameters in either of the following ways.

- Enclose the group of characters with delimiters (two matching characters). You can use any characters, but they must match. If one string immediately follows another, separate the two with a single delimiter that matches the others. For example:

```
"string of characters"
/STRING/
: my name is Larry :
"first string"second string"
/string 1/ string 2/
```

- Use a string variable. (See "Using Macros" later in this chapter.)

Syntax Notation

Syntax descriptions indicate what to enter and the order in which to do it. The command name is first; type it exactly as shown. Follow the command name with the correct parameters. Enter each as it is described in the section on parameters.

The syntax descriptions for each command use the following notations:

n = numeric parameter
str = string parameter
␣ = space character. When you see ␣, press the space bar.
text = one or more characters terminated by pressing ENTER

Getting Started

From the OS-9 prompt, start Edit by typing:

```
edit ENTER
```

Enter a command when the screen shows `E:`.

You can quit Edit at any time by pressing Q ENTER. The Q command terminates the editor and returns you to the OS-9 Shell, which responds with the `OS9:` prompt.

Following is a list of ways you can start the editor, including the effect of each. The examples call a file that already exists *oldfile*. They call a file to be created *newfile*.

EDIT

OS-9 loads the editor and starts it. The command does not establish an initial read or write files, but you can perform text file operations by opening files after the editor is started.

EDIT *newfile*

OS-9 loads the editor and starts it, creating the file called *newfile*. *Newfile* is the initial write file. There is no initial read file. However, you can open files to read later.

EDIT *oldfile*

OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called SCRATCH as the initial write file. When you end the edit session, OS-9 deletes *oldfile* and renames SCRATCH to *oldfile*. This gives the appearance of *oldfile* being updated.

Note: The two OS-9 utilities DEL and RENAME must be present on your system if you wish to start the editor in this manner.

EDIT *oldfile newfile*

OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates *newfile*—the initial write file. The terms *oldfile* and *newfile* refer to any properly constructed OS-9 pathlist.

Edit Commands

Displaying Text

L*n*

Lists (displays) the next *n* lines, starting at the current position of the edit pointer. The edit pointer position does not change.

l ENTER

displays the current line. If the edit pointer is not at the beginning of the line, only the portion of the line to the right of the edit pointer shows on the screen.

l 3 ENTER

displays the current line and the next two lines.

l * ENTER

displays all text from the current position of the edit pointer to the end of the buffer.

The L command displays text regardless of which verify mode is in effect.

X*n*

Displays the *n* lines that precede the edit pointer. The position of the edit pointer does not change. For example:

x ENTER

displays any text on the current line that precedes the edit pointer. If the edit pointer is at the beginning of the line, the command displays nothing.

x 3 ENTER

displays the two preceding lines and any text on the current line that precedes the edit pointer.

The X command displays text regardless of which verify mode is in effect.

Manipulating the Edit Pointer

[CTRL] [7] or [▲] on an external terminal

Moves the edit pointer to the beginning (first character) of the text buffer. The screen shows the up arrow when you hold down [CTRL] and press [7]. For example,

[CTRL] [7] [ENTER]

moves the edit pointer to the beginning of the buffer.

/

Moves the edit pointer to the end (last character) of the buffer. For example,

/ [ENTER]

moves the edit pointer past the end of the buffer.

[ENTER]

Moves the edit pointer to the beginning of the next line and displays it. Use this command to go through text one line at a time. You can look at each line, correct any mistakes, and then move to the next line.

+n Moves the edit pointer either to the end of the line or forward *n* lines and displays the line. Entering a value of zero moves the edit pointer to the end of the current line. For example:

```
+0 ENTER
```

Entering a value other than zero moves the pointer forward *n* lines and displays the line. For example,

```
+ ENTER
```

moves the pointer to the next line and displays the line. This command performs the same function as ENTER.

```
+10 ENTER
```

moves the pointer ahead 10 lines and displays the line.

```
+* ENTER
```

moves the edit pointer to the end of the buffer.

-n Moves the edit pointer either to the beginning of the line or backward *n* lines. For example:

```
-0 ENTER
```

moves the edit pointer to the beginning of the line and displays the line. Entering a value other than zero moves the edit pointer back *n* lines. For example,

```
- ENTER
```

moves the edit pointer back one line and displays the line.

```
-5 ENTER
```

moves the edit pointer back five lines and displays the line.

```
-* ENTER
```

moves the edit pointer to the beginning (top) of the buffer and displays the first line.

>*n*

Moves the edit pointer to the right *n* characters. Use this command to move the edit pointer to some position in the line other than the first character. For example,

> > [ENTER]

moves the edit pointer to the right one character.

> >25 [ENTER]

moves the edit pointer to the right 25 characters.

> >* [ENTER]

moves the edit pointer to the end of the buffer.

<*n*

Moves the edit pointer to the left *n* characters. Use this command to move the edit pointer to some position in a line other than the first character. For example:

> < [ENTER]

moves the edit pointer to the left one character.

> <10 [ENTER]

moves the edit pointer to the left 10 characters.

> <* [ENTER]

moves the edit pointer to the beginning of the buffer.

Inserting and Deleting Lines

␣*text* Preceding text lines with a space inserts the text as a new line ahead of the edit pointer. The position of the edit pointer does not change. For example,

```
␣Insert this line ENTER
```

inserts the line.

```
␣Line one ENTER
␣Line two ENTER
␣Line three ENTER
```

inserts three lines.

I*n str* Inserts a line of *n* copies of the specified string immediately before the position of the edit pointer. The position of the edit pointer does not change. For example,

```
i40/*/ ENTER
```

inserts a line containing 40 asterisks. You can also use the "I" command to insert a line containing a single copy of the string. This function is important when you want to use a macro to insert lines, since the space bar cannot be used within a macro. For example,

```
i"Line to insert" ENTER
```

inserts the line.

D*n*

Deletes (removes) *n* lines from the edit buffer, starting with the current line. This command displays the lines to be deleted. For example:

d ENTER

deletes the current line, regardless of the position of the edit pointer, and displays it.

d4 ENTER

deletes the current line and the next three lines.

d* ENTER

deletes everything from the current line to the end of the buffer.

K*n*

Kills (deletes) *n* characters, starting at the current position of the edit pointer. This command displays all deleted characters. For example,

k ENTER

deletes the character at the edit pointer.

k4 ENTER

deletes the character at the current position of the edit pointer and the next three characters.

k* ENTER

deletes everything from the current position of the edit pointer to the end of the buffer.

E*n str* Extends *n* lines by adding a string to the end of each line. E extends a line, displays it, and then moves the pointer past it. For example,

```
e/this is a comment/ ENTER
```

adds the string "this is a comment" to the end of the current line and moves the edit pointer to the next line.

```
e3/xx ENTER
```

adds the string xx to the end of the current line and the next two lines. It moves the pointer past these lines.

U Unextends (deletes) the remainder of a line from the current position of the pointer. Use U to remove extensions, such as comments, from a line. For example,

```
u ENTER
```

deletes all the characters from the current position of the pointer up to the end of the current line.

For some practice in using the commands that display text, manipulate the edit pointer, and insert and delete lines, turn to Sample Session 1 in this chapter.

Searching and Substituting

S*n string*

Searches for the next *n* occurrences of *string*. When Edit finds an occurrence, it displays the line and moves the edit pointer to the line. If Edit does not find the string or if all the occurrences have been found, the edit pointer does not move. For example,

```
s/my string/ ENTER
```

searches for the next occurrence of "my string".

```
s3"strung out" ENTER
```

searches for the next three occurrences of "strung out".

```
s*/seek and find/ ENTER
```

searches for all occurrences of "seek and find" between the edit pointer and the end of the text.

C*n string1 string2*

Changes the next *n* occurrences of *string1* to *string2*. When Edit finds *string1*, it moves the edit pointer past it and changes *string1* to *string2*, then it displays the updated line. If it does not find *string1* it displays "NOT FOUND." If all the occurrences have been found, the edit pointer does not move. For example,

```
c/this/that/ ENTER
```

changes the next occurrence of "this" to "that".

```
c2/in/out/ ENTER
```

changes the next two occurrences of "in" to "out".

```
c*!seek and find!sought and
found! ENTER
```

changes all occurrences of "seek and find" that are between the edit pointer and the end of text to "sought and found".

A*n*

Sets the SEARCH/CHANGE anchor to Column *n*. To find a string that begins in a specific column, set the anchor to the column position before using the search command to find it. If you do not include a value for *n*, Edit assumes Column 1. For example:

a [ENTER]

finds a string only if it begins in Column 1.

a20 [ENTER]

finds a string only if it begins in Column 20. If you use the A command to set the anchor, this setting remains in effect only for the current command line. After Edit executes the command, the anchor automatically returns to its normal value of zero.

For some practice in using the commands that search and substitute, turn to Sample Session 2 in this chapter.

Miscellaneous Commands

T*n*

Tabs (moves) the edit pointer to Column *n* of the current line. If *n* exceeds the line length, this command extends the line with spaces. For example,

t [ENTER]

moves the edit pointer to Column 1 of the current line.

t5 [ENTER]

moves the edit pointer to Column 5 of the current line.

.SHELL *command line*

Lets you use any OS-9 command from within the editor. The remainder of the command line following .SHELL passes to the OS-9 Shell for execution. For example,

```
.shell dir /d1 ENTER
```

calls the OS-9 Shell to display the directory of D1.

```
.shell basic09 ENTER
```

starts BASIC09.

```
.shell edit oldfile newfile ENTER
```

restarts the editor.

M*n*

Adjusts the amount of memory available for buffers and macros. If the workspace is full and the editor does not allow you to enter more text, increase the workspace size. If you need only a small amount of the available workspace, decrease the workspace size so that other OS-9 programs can use the memory. For example,

```
m5000 ENTER
```

sets the workspace size to 5000 bytes.

```
m10000 ENTER
```

sets the workspace size to 10000 bytes.

Before leaving Edit, you can increase the workspace. This decreases the time the editor takes to copy the input file to the output file, because the editor can read and write more data at one time. Edit changes memory in 256-byte pages. For the M command to have any effect, a new workspace size must differ from the current size by at least 256 bytes. The M command does not let you deallocate any workspace that Edit needs for buffers or macros.

.SIZE Displays the size of the workspace and the amount that has been used. For example:

```
.size
  521     15328
```

521 is the amount of workspace Edit uses for buffers and macros. 15328 is the amount of available memory.

Q Ends editing and returns to the OS-9 Shell. If you specified files when you started, Edit writes the text in Buffer 1 to the initial write file (specified when you start Edit). Next it copies the remainder of the initial input file (specified when you start Edit) to the initial write file. The editor then terminates, and control returns to the OS-9 Shell.

V*mode* Turns the verify mode on or off. Edit always starts with the verify mode on. Therefore, the editor displays the results of all the commands for which verify is appropriate. If you do not want to see the results of commands, turn off the verify mode by specifying 0 (zero) for *mode*. To turn verify back on, specify any non-zero number. For example,

v0 [ENTER]

turns off the verify mode.

v2 [ENTER]

turns on the verify mode.

v13 [ENTER]

turns on the verify mode.

If the verify mode is on when you switch to a macro, it remains on. If you turn off verify while in the macro, it is restored when you return to the editor.

Manipulating Multiple Buffers

.DIR Displays the directory of the editor's buffers and macros. For example:

```
BUFFERS:
$       0 (secondary buffer)
*       1 (primary buffer)
        5 (another buffer)

MACROS:

    MYMACRO
    LIST
    COPY
```

B*n* Makes buffer *n* the primary buffer. When you switch from one buffer to another, the old one becomes the secondary buffer, and the new one becomes the primary buffer. For example,

```
b5 [ENTER]
```

makes Buffer 5 the primary buffer. If Buffer 5 does not exist, Edit creates it.

P*n* Puts (moves) *n* lines into the secondary buffer. This command removes the lines from the primary buffer, starting at the position of the edit pointer, and inserts them into the secondary buffer before the current position of the edit pointer. It displays the text that is moved. For example,

```
p [ENTER]
```

moves one line to the secondary buffer.

```
p5 [ENTER]
```

moves five lines to the secondary buffer.

```
p* [ENTER]
```

moves all lines that are between the current position of the edit pointer and the end of text to the secondary buffer.

G*n*

Gets (moves) *n* lines from the secondary buffer. This command takes the lines from the top of the secondary buffer and inserts them into the primary buffer before the current position of the edit pointer. Edit then displays the moved lines. When used with the P command, G moves text from one place to another. For example,

```
g ENTER
```

gets one line from the secondary buffer.

```
g5 ENTER
```

gets five lines from the secondary buffer.

```
g* ENTER
```

gets all lines from the secondary buffer.

For some practice in using miscellaneous commands and the commands that manipulate multiple buffers, turn to Sample Session 3 in this chapter.

Text File Operations

This section of the manual describes the group of commands related to reading and writing OS-9 text files.

.NEW

Gets new text. Use .NEW when editing a file that is too large to fit into the editor's workspace. .NEW writes out all lines that precede the current line, then appends an equal amount of new text to the end of the buffer.

.NEW always writes text to the initial output file (created when you start the editor) and always reads text from the initial input file (specified when you start the editor).

If you have finished editing the text currently in the buffer, you can "flush" this text and fill the buffer with new text by moving the edit pointer to the bottom of the buffer and then using the .NEW command. For example:

```
/.new ENTER
```

If you wish to retain part of the text that is already in the buffer, move the edit pointer to the first line you wish to retain and then type `.new`. This command "flushes" all lines that precede the edit pointer. It then tries to read in new text that is the same size as the portion flushed out.

.READ *str* Prepares an OS-9 text file for reading. *str* specifies the pathlist. For example.

```
.read "myfile" ENTER
```

closes the current input file and opens "myfile" for reading.

You can specify an empty pathlist. For example,

```
.read "" ENTER
```

closes the current input file and restores the initial input file (specified when you start the editor) for reading.

An open file remains attached to the primary buffer until you close the file. You can have more than one input file open at any time by using the .READ command to open them in different buffers.

To read these files, switch to the proper buffer, and then use the R command to read from that buffer's input file. To close a file, you must be in the same buffer where the file was opened.

.WRITE *str* Opens a new file for writing. The *string* specifies the pathlist for the file you wish to create. For example,

```
.write "newfile" ENTER
```

closes the current write file and creates one called "newfile". You can specify an empty pathlist. For example:

```
.write "" ENTER
```

closes the current write file and restores the initial write file (specified when you start the editor).

.WRITE attaches a new write file to the primary buffer that remains attached until you close the file. You can have more than one write file open by using .WRITE to open them in different buffers. To write these files, switch to the proper buffer. To close a file, you must be in the same buffer where the file was opened.

R*n* Reads (gets) *n* lines of text from the buffer's input file. It displays the lines and inserts them before the current position of the edit pointer. For example,

```
r ENTER
```

reads one line from the input file.

```
r10 ENTER
```

reads 10 lines from the input file.

```
r* ENTER
```

reads the remaining lines from the input file.

If a file contains no more text, the screen shows the *END OF FILE* message.

W*n* Writes *n* lines to the output file, starting with the current line. It displays all lines that are deleted from the buffer. For example,

```
w ENTER
```

writes the current line to the output file.

```
w5 ENTER
```

writes the current line and the next four lines to the output file.

```
w* ENTER
```

writes all lines from the current line to the end of the buffer to the output file.

For some practice in using the commands that read and write OS-9 text files, turn to Sample Session 4 in this chapter.

Conditionals and Command Series Repetition

When a command cannot be executed, the editor sets an internal flag, and the screen shows *FAIL*. For example, if you try to read from a file that has no more text, the editor sets the fail flag. A set fail flag means that the editor cannot execute any more commands until Edit encounters one of the following:

- The end of a command line typed from the keyboard.
- The end of the current loop. Any loops that are more deeply nested are skipped. (See the repeat command.)
- A colon (:) command. Since loops nested deeper than the current level are skipped, any occurrences of : that are in a more deeply nested loop are also skipped.

Following are the commands and conditions that set the fail flag:

<	Trying to move the edit pointer beyond the beginning of the edit buffer.
>	Trying to move the edit pointer beyond the + end of the buffer.
S,C	Not finding a string that was searched for.
G	No text left in the secondary buffer.
R	No text left in the read file.
P,W	No text left in the primary buffer.

If you specify an asterisk for the repeat count on these commands, Edit does not set the fail flag, because an asterisk usually means continue until there is nothing more to do. The following commands explicitly set the fail flag if some condition is not true.

.EOF	Tests for end-of-file. .EOF succeeds if there is no more text to read from a file. Otherwise, it sets the fail flag.
.NEOF	Tests for not end-of-file. .NEOF succeeds if there is text to read from the file. Otherwise, it sets the fail flag.
.EOB	Tests for end-of-buffer. .EOB succeeds if the edit pointer is at the end of the buffer. Otherwise, it sets the fail flag.
.NEOB	Tests for not end-of-buffer. .NEOB succeeds if the edit pointer is not at the end of the buffer. Otherwise, it sets the fail flag.
.EOL	Tests for end-of-line. This test succeeds if the edit pointer is at the end of the line. Otherwise, it sets the fail flag.
.NEOL	Tests for not end-of-line. .NEOL succeeds if the edit pointer is not at the end of the line. Otherwise, it sets the fail flag.
.ZERO *n*	Tests for zero value. .ZERO succeeds if *n* equals zero. Otherwise, it sets the fail flag.

.STAR *n* Tests for star (asterisk). .STAR succeeds if *n* equals 65,535 ("*"). Otherwise, it sets the fail flag.

.STR *str* Tests for string match. .STR succeeds if the characters at the current position of the edit pointer match the string. Otherwise, it sets the fail flag.

.NSTR *str* Tests for string mismatch. .NSTR succeeds if the characters at the current position of the edit pointer do not match the string. Otherwise, it sets the fail flag.

.S Exits and succeeds. This is an unconditional exit from the innermost loop or macro. The fail flag clears after the exit.

.F Exits and fails. This is an unconditional exit from the innermost loop or macro. The fail flag sets after the exit.

[*commands*]*n* Repeats the *commands n* times. Left and right brackets form a loop that repeats the enclosed *commands n* times. (The loop must be repeated at least once.) If you enter the loop command from the keyboard, it must all be on one line. If it is part of a macro, however, it can span several command lines. For example,

```
[ l ] 5 ENTER
```

repeats the L command five times.

Note: This is not the same as L5, which executes the L command only once and has 5 as its parameter.

[+]* Displays lines starting with the next line up to the end of the buffer and moves the edit pointer to the end of the buffer.

This command repeats until the operation reaches the end of the buffer. Then, when the command tries to move the edit pointer past the end of the buffer, Edit sets the fail flag, terminates the loop, then clears the fail flag.

: *commands*

Executes the commands following the colon based on the state of the fail flag. For example:

FAIL FLAG CLEAR	Skips all commands that follow the colon (:) up to the end of the current loop or macro.
FAIL FLAG SET	Clears the fail flag, and executes the commands that follow the colon (:).

Below is a command line that deletes all lines that do not begin with the letter A.

```
CTRL 7 [ .neob [ .str"A" + : d ]
] * ENTER
```

^ moves the edit pointer to the beginning of the buffer. The outer loop tests for the end of the buffer and terminates the loop when it is reached.

The inner loop tests for A at the beginning of the line. If there is an A, the + command is executed. Otherwise, it executes the D command.

Below is a command that searches the current line for "find it". If the command finds the text, it displays the line. Otherwise, the command line fails and the screen shows * FAIL *.

```
[ .eol v0 -0 v .f : .str"find it"
-0 .s : [>] ]* ENTER
```

.EOL V0 -0 V .F tests to determine if the edit pointer is at the end of the line. If it is, Edit turns off the verify mode to prevent -0 from displaying the line. Then it turns verify back on, and .F ends the loop.

If the edit pointer is not at the end of the line, the .STR command searches for "find it" at the current position of the edit pointer. If it is at the end of the line, Edit executes the -0 .S commands. This execution moves the edit pointer back to the beginning of the line, displays the line, and terminates the loop. Otherwise, the > command moves the edit pointer to the next position in the line.

The brackets prevent the command from failing and terminating the main loop if the end of the buffer is reached.

Edit Macros

Edit macros are commands you create to perform a specialized or complex task. For example, you can replace a frequently used series of commands with a single macro. First, save the series in a macro. Then each time you need it, type a period followed by the macro's name and parameters. The editor responds as if you had typed the series of commands.

Macros consist of two main parts, the header and the body. The header gives the macro a name and describes the type and order of its parameters. The body consists of any number of ordinary commands. (Except for a space character and [ENTER], you can use any command in a macro).

Note: Macros cannot create new macros.

To create a macro, first define it with the .MAC command. Then enter the header and body in the same manner as you enter text into an edit buffer. When you are satisfied with the macro, close its definition by pressing [Q] [ENTER]. This command returns you to the normal edit mode.

Macro Headers. A macro header must be the first line in each macro. It consists of a name, and a "variable list" that describes the macro's parameters, if there are any. The name consists of any number of consecutive letters and underline characters. Following are possible macro names:

del_all
trim_spaces
LIST
CHANGE_X_TO_Y

Although you can make a macro name any length, it is better to keep it short, because you must spell it the same way each time you use it. You can use upper- and lowercase letters or a mixture.

Using Macros. Like other commands, you can give parameters to macros so that they are able to work with different strings and with different numbers of items. Macros are unable to use parameters directly. Instead, Edit passes the parameters on to the commands that make up the macro.

To pass the macro's parameters to these commands, use the variable list in the macro header to tell each command which of the macro's parameters to use. Each variable in the variable list represents the value of the macro parameter in its corresponding position. Use the corresponding variable wherever the parameter's value is needed.

The two types of variables are numeric and string. A numeric variable is a variable name preceded by the # character. A string variable is a variable name preceded by a $ character. Variable names, like macro names, are composed of any number of consecutive letters and underline characters. Examples of numeric variables are:

```
#N
#ABC
#LONG_NUMBER_VARIABLE
```

Examples of string variables are:

```
$A
$B
$STR
$STR_A
$lower_case_variable_name
```

The function of the edit macro below is the same as that of the S command, to search for the next *n* occurrences of a string.

The first line of the macro is the macro header. It assigns the macro's name as SRCH. It also specifies that the macro needs one numeric parameter (#N) and one string parameter ($STR). The entire body of the macro is the second line. This example passes both of the macro's parameters to the S command, which does the actual searching.

```
SRCH #N $STR
S #N $STR
```

Here is an example of how to execute this macro:

```
.SRCH 15 "string" [ENTER]
```

In the next example, the order of the parameter is reversed. Therefore, when executing the macro, use the reverse order. The macro structure is:

```
SRCH $STR #N
S #N $STR
```

Specify the parameters for the "S" command in the proper order since it is only the "SRCH" macro that is changed. The following example shows how to execute this macro. The order of the parameters corresponds directly to the order of the variables in the variable list.

```
.SRCH "string" 15 [ENTER]
```

Macro Commands

Although macro editing has the same functions as text editing, the macro mode also includes some special commands. The macro commands you can use are as follows:

! *text* Places comments inside a macro. Ignores the remainder of the line following the ! command. This command lets you include, as part of a macro, a short description of what it does. Comments can help you remember the function of a macro. For example:

```
!
<^>! Move the pointer to the top of the
     buffer.
1*!  Display all lines of text.
!
```

In this example there are four comments. Two are empty, and two describe the commands that precede them.

.macro name Executes the macro specified by the name following the period (.). For example:

```
.mymacro ENTER
.list 0 ENTER
.trim " " ENTER
.merge " file_a " file b b" ENTER
```

.MAC *str* Creates a new macro or opens the definition of an existing one so that it can be edited. To create a new macro, specify an empty string. For example,

```
.mac"" ENTER
```

creates a new macro and puts you into the macro mode.

The screen shows M: instead of E: when the editor is in the macro mode. To edit a macro that already exists, specify the macro's name. For example,

```
.mac "mymacro" ENTER
```

opens the macro "MYMACRO" for editing.

When a macro is open, edit it, or enter its definition with the same commands you use in a text buffer. After you edit the macro, press Q ENTER to close its definition and return to the edit mode. The first line of the macro must begin with a name that is not already used in order to close the definition and return to Edit.

.SAVE *str1 str2*

Saves macros on an OS-9 file. *Str1* specifies a list of macros to be saved. Separate the macro names with spaces. *Str2* specifies the pathlist for the file on which you want to save the macros. For example:

```
.save "mymacro"myfile" ENTER
```

saves the macro "MYMACRO" on the file "MYFILE".

```
.save "maca macb macc"mfile" ENTER
```

saves the macros "MACA," "MACB," and "MACC" on the file "MFILE".

.SEARCH *n str*

Searches the text file buffer for the specified string. When a match is found, it stops and displays that line. The *n* option permits a search for the *nth* occurrence of a string match. This command is the same as S *n str*.

.LOAD *str* Loads macros from an OS-9 file. As each macro loads, Edit verifies that no other macro already exists with the same name. If one does, the macro with the duplicate name does not load, and Edit skips to the next macro on the file. Edit displays the names of all macros it loads. For example,

```
.load "macrofile" ENTER
```

loads the macros in the file called MACROFILE.

```
.load "myfile" ENTER
```

loads the macros in the file called MYFILE.

.DEL *str* Deletes the macro specified by the string. For example,

```
.del "mymacro" ENTER
```

deletes the macro called MYMACRO.

```
.del "list" ENTER
```

deletes the macro called LIST.

.DIR Displays the current edit buffer area. All edit buffers and macros currently in memory are displayed.

.CHANGE *n str1 str2* Changes the occurrence of *str1* to *str2*. The *n* option permits *n* occurrences of *str1* to be changed to *str2*.

Q Ends a macro edit session and returns you to the normal edit mode. For example:

```
Search_and_Delete #N $STR
!This example MACRO is used to
!check
!the string at the beginning of
!an #N number of lines. If the
!string matches, it will delete
!that line from the text buffer
!file.
!
!NOTE: The way the editor
!processes a MACRO causes it to
!see any parameters in the outer
!loop first. Thus, the #N
!parameter is processed before
!the STR parameter.
!
[^]          !Move to start of
             !edit buffer
[            !start of outer loop
.neob        !test for buffer end
[            !start of inner loop
.nstr $str   !test for not string
             !match
+            !go to next line if
             !no match
:            !if flag clear skip
             !next command
D            !delete line if flag
             !set
]            !end of inner loop
]#N          !end of outer loop
! End of Macro
```

For practice in using macro commands, turn to Sample Session 5 in this chapter.

Sample Session 1

Clear the buffer by deleting its contents.

You Type: CTRL 7 D* ENTER
Screen Shows: ^D*

Insert three lines into the buffer. Begin each line with a space, which is the command for inserting text.

You Type:
```
 MY FIRST LINE ENTER
 MY SECOND LINE ENTER
 MY THIRD LINE ENTER
```
Screen Shows:
```
MY FIRST LINE
MY SECOND LINE
MY THIRD LINE
```

Move the edit pointer to the top of the text. The editor always considers the first character you type a command.

Note: CTRL 7 always shows ^ on the screen. Typing -* also moves the edit pointer to the beginning of a buffer.

You Type: CTRL 7 ENTER
Screen Shows: ^

List (display) the first line you inserted into the buffer.

You Type: L ENTER
Screen Shows:
```
L
 MY FIRST LINE
```

Display the first two lines you inserted into the buffer.

You Type: L2 ENTER
Screen Shows:
```
L2
 MY FIRST LINE
 MY SECOND LINE
```

Move to the next line and display it.

You Type: ENTER
Screen Shows: MY SECOND LINE

Move to the next line and display it.

You Type: ENTER
Screen Shows: MY THIRD LINE

Using L, display text beginning at the position of the edit pointer.

You Type: L [ENTER]
Screen Shows:
```
L
 MY THIRD LINE
```

Insert a line into the buffer.

Note: In the next sample you see that the insert comes before the current position of the edit pointer.

You Type: □INSERT A LINE [ENTER]
Screen Shows:
```
INSERT A LINE
```

The following command line consists of more than one command. [CTRL][7] [ENTER] moves the edit pointer to the top of the text. L displays the text, and the asterisk (*) following L indicates that text is displayed through to the end of the buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 MY FIRST LINE
 MY SECOND LINE
 INSERT A LINE
 MY THIRD LINE
```

Show the position of the edit pointer.

You Type: L [ENTER]
Screen Shows:
```
L
 MY FIRST LINE
```

Move the edit pointer forward two lines and display the lines.

You Type: +2 [ENTER]
Screen Shows:
```
+2
 INSERT A LINE
```

Display all lines from the edit pointer to the end of the buffer.

You Type: L* [ENTER]
Screen Shows:
```
L*
 INSERT A LINE
 MY THIRD LINE
```

Move the edit pointer to the end of the buffer.

You Type: / [ENTER]
Screen Shows:
```
/
```

Determine if the edit pointer is at the end of text. Since the screen shows no more lines, the edit pointer is at the end-of-text.

You Type: L* [ENTER]
Screen Shows:
```
L*
```

Insert two more lines.

You Type: ␣FIFTH LINE [ENTER]
␣LAST LINE [ENTER]

Screen Shows:
```
FIFTH LINE
LAST LINE
```

Move the edit pointer back one line, and display the line.

You Type: -2 [ENTER]

Screen Shows:
```
-2
 FIFTH LINE
```

Move the edit pointer back two lines, and display the line.

You Type: -3 [ENTER]

Screen Shows:
```
-3
 MY SECOND LINE
```

Move the edit pointer three characters to the right and display the remainder of the line.

Note: You must put spaces between commands.

You Type: >3 L [ENTER]

Screen Shows:
```
>3 L
 SECOND LINE
```

Display the characters that precede the edit pointer on the current line.

You Type: X [ENTER]

Screen Shows:
```
X
 MY
```

Move the edit pointer to the end of the current line.

You Type: +Ø [ENTER]

Screen Shows:
```
+Ø
```

Determine if the edit pointer is at the end of the line. It is, since the screen shows no lines.

You Type: L [ENTER]

Screen Shows:
```
L
```

Display the characters that precede the edit pointer on the current line.

You Type: X [ENTER]

Screen Shows:
```
X
 MY SECOND LINE
```

Move the edit pointer back to the beginning of the current line.

You Type: -0 [ENTER]
Screen Shows: -0
MY SECOND LINE

Determine if the edit pointer is at the beginning of the line. Since the screen shows no lines, the pointer is at the beginning.

You Type: X [ENTER]
Screen Shows: X

Go to the beginning of the text.

You Type: [CTRL][7] [ENTER]
Screen Shows: ^

Insert a line of 14 asterisks.

You Type: I14"*" [ENTER]
Screen Shows: I14"*"
****

Insert an empty line.

You Type: I"" [ENTER]
Screen Shows: I""

Move to the top of the text, and display all lines in the buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows: ^L*
****

MY FIRST LINE
MY SECOND LINE
INSERT A LINE
MY THIRD LINE
FIFTH LINE
LAST LINE

Move the edit pointer forward two lines.

You Type: +2 [ENTER]
Screen Shows: +2
MY FIRST LINE

Extend the line with XXX.

You Type: E" XXX" [ENTER]
Screen Shows: E" XXX"
MY FIRST LINE XXX

Display the current line.

Note: The previous E command moved the edit pointer to the next line.

You Type: L [ENTER]
Screen Shows:
```
L
 MY SECOND LINE
```

Extend three lines with YYY.

You Type: E3"␣YYY" [ENTER]
Screen Shows:
```
E3" YYY"
 MY SECOND LINE YYY
 INSERT A LINE YYY
 MY THIRD LINE YYY
```

Move back 2 lines.

You Type: -2 [ENTER]
Screen Shows:
```
-2
 INSERT A LINE YYY
```

Move the edit pointer to the end of the line and then move the edit pointer back four characters. Display the current line, starting at the edit pointer.

You Type: +0 <4 L [ENTER]
Screen Shows:
```
+0 <4 L
  YYY
```

Truncate the line at the current position of the edit pointer. This command removes the YYY extension.

You Type: U [ENTER]
Screen Shows:
```
U
 INSERT A LINE
```

Go to the top of the text and display the contents of the buffer.

You Type: [CTRL][7] L* [ENTER]
Screen Shows:
```
^L*
 ***************

 MY FIRST LINE XXX
 MY SECOND LINE YYY
 INSERT A LINE
 MY THIRD LINE YYY
 FIFTH LINE
 LAST LINE
```

Delete the current line and the next line.

You Type: D2 [ENTER]
Screen Shows:
```
D2
 **************
```

Move the edit pointer forward two lines.

You Type: +2 [ENTER]
Screen Shows:
```
+2
 INSERT A LINE
```

Delete this line.

You Type: D [ENTER]
Screen Shows:
```
D
 INSERT A LINE
```

Display the current line.

You Type: L [ENTER]
Screen Shows:
```
L
  MY THIRD LINE YYY
```

Move the edit pointer to the right three characters and display the text.

You Type: >3 L [ENTER]
Screen Shows:
```
>3 L
 THIRD LINE YYY
```

Kill (delete) the 11 characters that constitute THIRD LINE.

You Type: K11 [ENTER]
Screen Shows:
```
K11
 THIRD LINE
```

Go to the beginning of the line and display it.

You Type: -Ø [ENTER]
Screen Shows:
```
-Ø
 MY YYY
```

Concatenate (combine) two lines. Move the edit pointer to the end of the line; delete the character at the end of the line; move the edit pointer back to the beginning of the lines. Display the line.

You Type: +Ø K -Ø [ENTER]
Screen Shows:
```
Ø K -Ø
 MY YYYFIFTH LINE
```

Separate the two lines by inserting an end-of-line character.

You Type: >6 I/ / [ENTER]
Screen Shows:
```
>6 I/ /
 MY YYY
```

Note: The end of line character is inserted before the current position of the edit pointer.

You Type: L [ENTER]

Screen Shows:
```
L
 FIFTH LINE
```

Sample Session 2

Clear the buffer by deleting its contents.

You Type: [CTRL][7] D* [ENTER]

Insert lines.

You Type:
```
□ONE TWO THREE 1.0 [ENTER]
□ONE [ENTER]
□□TWO [ENTER]
□□□THREE [ENTER]
□ONE TWO THREE 2.0 [ENTER]
□ONE [ENTER]
□□TWO [ENTER]
□□□THREE [ENTER]
□ONE TWO THREE 3.0 [ENTER]
```

Screen Shows:
```
ONE TWO THREE 1.0
ONE
  TWO
   THREE
ONE TWO THREE 2.0
 ONE
  TWO
   THREE
ONE TWO THREE 3.0
```

Go to the top of the text, and display all lines in the buffer.

You Type: [CTRL][7] L* [ENTER]

Screen Shows:
```
^L*
ONE TWO THREE 1.0
ONE
 TWO
  THREE
ONE TWO THREE 2.0
ONE
 TWO
  THREE
ONE TWO THREE 3.0
```

Search for the next occurrence of TWO.

You Type: `S "TWO" ENTER`

Screen Shows:

```
S"TWO"
 ONE TWO THREE 1.0
```

Search for all occurrences of TWO that are between the edit pointer and the end of the buffer.

You Type: `S*/TWO/ ENTER`

Screen Shows:

```
S*/TWO/
 ONE TWO THREE 1.0
  TWO
 ONE TWO THREE 2.0
  TWO
 ONE TWO THREE 3.0
```

Go to the top of the buffer, and change the first occurrence of THREE to ONE.

You Type: `CTRL 7 C/THREE/ONE/ ENTER`

Screen Shows:

```
^ C/THREE/ONE/
 ONE TWO ONE 1.0
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 2, and then use the search command to find each occurrence of TWO that begins in Column 2. Skip all other occurrences.

You Type: `CTRL 7 A2 S*/TWO/ ENTER`

Screen Shows:

```
^ A2 S*/TWO/
  TWO
  TWO
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 1, and change each occurrence of ONE that begins in that column to XXX.

Note: ONE in Line 1 is not changed, since it does not begin in Column 1.

You Type: `CTRL 7 AC*/ONE/XXX/ ENTER`

Screen Shows:

```
^ AC*/ONE/XXX/
 XXX TWO ONE 1.0
 XXX
 XXX TWO THREE 2.0
 XXX
 XXX TWO THREE 3.0
```

Go to the top of the buffer, and display the text.

You Type: CTRL 7 L* ENTER
Screen Shows:

```
L*
 XXX TWO ONE 1.0
 XXX
  TWO
   THREE
 XXX TWO THREE 2.0
 XXX
  TWO
   THREE
 XXX TWO THREE 3.0
```

Change the remaining ONE to XXX.

Note: The anchor is no longer set. It is reset to zero after each command is executed.

You Type: C/ONE/XXX/ ENTER
Screen Shows:

```
C/ONE/XXX/
XXX TWO XXX 1.0
```

Move to the beginning of the current line.

You Type: -0 ENTER
Screen Shows:

```
-0
XXX TWO XXX 1.0
```

Change three occurrences of XXX to ZZZ.

You Type: C3/XXX/ZZZ/ ENTER
Screen Shows:

```
C3/XXX/ZZZ/
ZZZ TWO XXX 1.0
ZZZ TWO ZZZ 1.0
ZZZ
```

Sample Session 3

Clear the buffer by deleting its contents:

You Type: CTRL 7 D* ENTER

Display the directory of buffers and macros. The dollar sign ($) identifies the secondary buffer as Buffer 0. The asterisk (*) identifies the primary buffer as Buffer 1. Edit has no macros defined. This is the initial environment when you start Edit.

You Type: .DIR [ENTER]

Screen Shows:

```
.DIR

 BUFFERS:
 $      0
 *      1

 MACROS:
```

Insert some lines into Buffer 1 so that later you can identify it.

You Type:

```
□BUFFER ONE 1.0 [ENTER]
□BUFFER ONE 2.0 [ENTER]
□BUFFER ONE 3.0 [ENTER]
□BUFFER ONE 4.0 [ENTER]
```

Screen Shows:

```
BUFFER ONE 1.0
BUFFER ONE 2.0
BUFFER ONE 3.0
BUFFER ONE 4.0
```

Display the text in Buffer 1.

You Type: [CTRL][7] L* [ENTER]

Screen Shows:

```
^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Make Buffer 0 the primary buffer. Buffer 1 becomes the secondary buffer.

You Type: B0 [ENTER]

Screen Shows:

```
B0
```

Display the directory of buffers and macros.

Note: The symbols identifying the buffers are now reversed.

You Type: .DIR [ENTER]

Screen Shows:

```
.DIR

 BUFFERS:
 $      1
 *      0

 MACROS:
```

Insert some lines into Buffer 0.

You Type:
```
□BUFFER ZERO 1.0 [ENTER]
□BUFFER ZERO 2.0 [ENTER]
□BUFFER ZERO 3.0 [ENTER]
□BUFFER ZERO 4.0 [ENTER]
```
Screen Shows:
```
BUFFER ZERO 1.0
BUFFER ZERO 2.0
BUFFER ZERO 3.0
BUFFER ZERO 4.0
```

Display the text in Buffer 0.

You Type: [CTRL][7] L* [ENTER]

Screen Shows:
```
^L*
 BUFFER ZERO 1.0
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Switch to Buffer 1.

You Type: B [ENTER]

Screen Shows:
```
B
```

Display the text in Buffer 1.

You Type: [CTRL][7] L* [ENTER]

Screen Shows:
```
^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Move the edit pointer to Line 3 in this buffer.

You Type: +2 [ENTER]

Screen Shows:
```
+2
 BUFFER ONE 3.0
```

Switch to Buffer 0.

You Type: B0 [ENTER]

Screen Shows:
```
B0
```

Display the text in Buffer 0.

You Type: L* [ENTER]

Screen Shows:
```
L*
 BUFFER ZERO 1.0
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Move the edit pointer to Line 2 in this buffer.

You Type: + [ENTER]

Screen Shows:
```
+
 BUFFER ZERO 2.0
```

Switch to Buffer 1.

You Type: B [ENTER]

Screen Shows:
```
B
```

Display the text in Buffer 1 from the current position of the edit pointer.

Note: The position of the edit pointer has not changed since you switched to Buffer 0.

You Type: L* [ENTER]

Screen Shows:
```
L*
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Switch to Buffer 0.

You Type: B0 [ENTER]

Screen Shows:
```
B0
```

Display the text in Buffer 0 from the current position of the edit pointer.

Note: The position of the edit pointer has not changed since you switched to Buffer 1.

You Type: L* [ENTER]

Screen Shows:
```
L*
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Delete the contents of Buffer 0.

You Type: [CTRL] [7] D* [ENTER]

Screen Shows:
```
^D*

 BUFFER ZERO 1.0
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Make Buffer 1 the primary buffer and Buffer 0 the secondary buffer.

You Type: B [ENTER]

Screen Shows:
```
B
```

Move two lines from the primary buffer (Buffer 1) into the secondary buffer (Buffer 0).

You Type: [CTRL][7] P2 [ENTER]

Screen Shows:
```
^P2
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Switch to Buffer 0, and show that the lines were moved to it.

You Type: B0[CTRL][7]L* [ENTER]

Screen Shows:
```
B0^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Switch to Buffer 1. Go to the bottom of the buffer, and get the text out of the secondary buffer.

You Type: B/G* [ENTER]

Screen Shows:
```
B/G*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Show the contents of the buffer.

Note: The order of the lines is changed as a result of moving the text.

You Type: [CTRL][7]L* [ENTER]

Screen Shows:
```
^L*
 BUFFER ONE 3.0
 BUFFER ONE 4.0
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Move two lines into the secondary buffer.

You Type: P2 [ENTER]

Screen Shows:
```
P2
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Move to the bottom of the buffer, and get the lines back out of the secondary buffer.

You Type: /G* [ENTER]

Screen Shows:
```
/G*
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Show that the order of the lines is restored.

You Type: CTRL 7 L*

Screen Shows:

```
L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Sample Session 4

Clear the buffer by deleting its contents:

You Type: CTRL 7 D* ENTER

Enter some lines of text.

You Type:

```
□LINE ONE ENTER
□SECOND LINE OF TEXT ENTER
□THIRD LINE OF TEXT ENTER
□FOURTH LINE ENTER
□FIFTH LINE ENTER
□LAST LINE ENTER
```

Screen Shows:

```
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FOURTH LINE
FIFTH LINE
LAST LINE
```

Open the file Oldfile for writing.

You Type: .WRITE"oldfile" ENTER

Screen Shows:

```
.WRITE"oldfile"
```

Write all lines to the file.

You Type: CTRL 7 W* ENTER

Screen Shows:

```
^W*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
 FIFTH LINE
 LAST LINE

*END OF TEXT*
```

Close the file.

You Type: .WRITE// ENTER

Screen Shows:

```
.WRITE//
```

Verify that the buffer is empty.

You Type: CTRL 7 L* ENTER
Screen Shows:
```
^L*
```

Open the file Oldfile for reading.

You Type: .READ"oldfile" ENTER
Screen Shows:
```
.READ"oldfile"
```

Create a new file called Newfile for writing.

You Type: .WRITE"newfile" ENTER
Screen Shows:
```
.WRITE"newfile"
```

Read four lines from the input file. The screen shows the lines as they are read in.

You Type: R4 ENTER
Screen Shows:
```
R4
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
```

Read all the remaining text from the file. The screen shows the lines. When there is no more text, the screen shows the *END OF FILE* message.

You Type: R* ENTER
Screen Shows:
```
R*
 FIFTH LINE
 LAST LINE

*END OF FILE*
```

Go to the top of the buffer, and display the text to make sure it is inserted into the buffer.

You Type: CTRL 7 L* ENTER
Screen Shows:
```
^L*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
 FIFTH LINE
 LAST LINE
```

Write three lines to the output file, and display the lines.

You Type: W3 [ENTER]

Screen Shows:
```
W3
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
```

Move to the next line and display it.

You Type: + [ENTER]

Screen Shows:
```
+
 FIFTH LINE
```

Show that when writing lines, the editor starts at the current line and not at the top of the buffer.

You Type: W [ENTER]

Screen Shows:
```
W
 FIFTH LINE
```

Go to the top of the buffer, and display the text to be sure that the lines were written to the output file.

You Type: [CTRL][7]L* [ENTER]

Screen Shows:
```
^L*
 FOURTH LINE
 LAST LINE
```

Clear the buffer.

You Type: [CTRL][7]D* [ENTER]

Screen Shows:
```
^D*
 FOURTH LINE
 LAST LINE
```

Switch to Buffer 2. Open the input file Oldfile, and read two lines from it.

You Type: B2 .READ"oldfile" R2 [ENTER]

Screen Shows:
```
B2 .READ"oldfile" R2
 LINE ONE
 SECOND LINE OF TEXT
```

Switch to Buffer 1. Open the input file Oldfile and read one line of text.

You Type: B .READ"oldfile" R [ENTER]

Screen Shows:
```
B .READ"oldfile" R
 LINE ONE
```

Switch to Buffer 2, and read one line.

Note: Your place in the file was not lost.

You Type: `B2 R [ENTER]`

Screen Shows:
```
B2 R
 THIRD LINE OF TEXT
```

Switch to Buffer 1, and read one line of text.

Note: Your place in the file was not lost.

You Type: `B R [ENTER]`

Screen Shows:
```
B R
 SECOND LINE OF TEXT
```

Switch to Buffer 2, and delete its contents.

You Type: `B2 [CTRL][7]D* [ENTER]`

Screen Shows:
```
B2 ^D*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
```

Insert some extra lines into the buffer.

You Type:
```
□EXTRA LINE ONE [ENTER]
□EXTRA LINE TWO [ENTER]
```

Screen Shows:
```
EXTRA LINE ONE
EXTRA LINE TWO
```

Try to write B2 buffer to file. It fails because you have not opened a file in this buffer.

You Type: `[CTRL][7]W* [ENTER]`

Screen Shows:
```
^W*
*FILE CLOSED*
```

Close the file for Buffer 1, and return to Buffer 2.

You Type: `B .WRITE// B2 [ENTER]`

Screen Shows: `B .WRITE// B2`

Open the old "write" file for reading, and then read it back in.

You Type: `.READ"newfile" R* [ENTER]`

Screen Shows:
```
.READ"newfile" R*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FIFTH LINE

*END OF FILE*
```

Display the contents of the buffer.

Note: It read the file into the beginning of the buffer, since that was the position of the edit pointer.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FIFTH LINE
 EXTRA LINE ONE
 EXTRA LINE TWO
```

Sample Session 5

Delete all text from the edit buffer.

You Type: CTRL 7 D * ENTER

Insert three lines.

You Type:

```
□LINE ONE ENTER
□LINE TWO ENTER
□LINE THREE ENTER
```

Screen Shows:

```
LINE ONE
LINE TWO
LINE THREE
```

Create a new macro using an empty string.

You Type: .MAC// ENTER

Screen Shows: M:

Display the contents of the macro mode, which is now open.

Note: The E prompt is now M.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
```

Define the macro.

You Type:

```
□FIND ENTER
□S"TWO" ENTER
```

Screen Shows:

```
FIND
S"TWO"
```

Display the contents of the macro.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
FIND
S"TWO"
```

Close the macro's definition.

You Type: `Q [ENTER]`
Screen Shows:
```
E:
```

Display the directory of buffers and macros.

You Type: `.DIR [ENTER]`
Screen Shows:
```
.DIR
 BUFFERS:
 $        0
 *        1

 MACROS:
   FIND
```

Display the contents of the edit buffer.

You Type: `[CTRL][7]L* [ENTER]`
Screen Shows:
```
^ L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND macro to find the string TWO.

You Type: `.FIND [ENTER]`
Screen Shows:
```
.FIND
 LINE TWO
```

Reopen the definition of the FIND macro.

You Type: `.MAC/FIND/ [ENTER]`
Screen Shows:
```
.MAC/FIND/
M:
```

Show that the macro is still intact.

You Type: `[CTRL][7]L* [ENTER]`
Screen Shows:
```
^L*
 FIND
 S"TWO"
```

Add the numeric parameter and the string parameter to the macro's header.

You Type: `C/FIND/FIND #N $STR/ [ENTER]`
Screen Shows:
```
C/FIND/FIND #N $STR/
 FIND #N $STR
```

Move to the second line of the macro.

You Type: `+ [ENTER]`
Screen Shows:
```
+
 S"TWO"
```

Give the macro's parameters to the S command. Now the FIND macro will perform the same function as the S command.

You Type: `C/"TWO"/ #N $STR/ ENTER`
Screen Shows:
```
C/"TWO"/ #N $STR
 S #N $STR
```

Close the macro's definition.

You Type: `Q ENTER`
Screen Shows:
```
E:
```

Display the contents of the edit buffer.

You Type: `CTRL 7 L* ENTER`
Screen Shows:
```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND macro to find the next two occurrences of LINE.

You Type: `.FIND 2 /LINE/ ENTER`
Screen Shows:
```
.FIND 2 /LINE/
 LINE ONE
 LINE TWO
```

Create a new macro.

You Type: `.MAC// ENTER`
Screen Shows:
```
.MAC//
M:
```

Define the macro FIND_LINE, which performs the same function as the S command except that it returns the edit pointer to the head of the line after finding the last occurrence of STR.

You Type: `□FIND_LINE #N $STR ENTER`
Screen Shows:
```
FIND_LINE #N $STR
```

You Type: `□S #N $STR ENTER`
Screen Shows:
```
S #N $STR
```

Turn off the verify mode.

You Type: `□V0 ENTER`
Screen Shows:
```
V0
```

Move the edit pointer to the first character of the current line.

You Type: `-0 ENTER`
Screen Shows:
```
-0
```

Close the macro's definition.

You Type: Q [ENTER]
Screen Shows:
```
Q

E:
```

Display the contents of the edit buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND_LINE macro to search for the string TWO.

You Type: .FIND_LINE/TWO/ [ENTER]
Screen Shows:
```
.FIND_LINE/TWO/
 LINE TWO
```

Show that the FIND_LINE macro left the edit pointer at the head of the line.

You Type: L [ENTER]
Screen Shows:
```
L
 LINE TWO
```

Create a new macro.

You Type: .MAC// [ENTER]
Screen Shows:
```
.MAC//
M:
```

Use the exclamation point (!) command to comment itself. Type the following:

```
CONVERT_TO_LINES #N [ENTER]
! This is a comment [ENTER]
! [ENTER]
! This macro converts the next n [ENTER]
! space characters to new line [ENTER]
! characters. [ENTER]
V0          ! Turn verify mode off [ENTER]
            ! to prevent intermediate results [ENTER]
            ! from being displayed. [ENTER]
! [ENTER]
[           ! Begin loop [ENTER]
.SEARCH/ /  ! Search for the space character. [ENTER]
I//         ! Insert empty line (new line character). [ENTER]
-           ! Back up one line. [ENTER]
C/ //       ! Delete the next space character. [ENTER]
L +         ! Show line, move past it. [ENTER]
] #N        ! End of loop. Repeat #N times. [ENTER]
```

Close the macro's definition.

You Type: Q [ENTER]
Screen Shows:
```
Q
E:
```

Display the contents of the edit buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Convert all space characters to new line characters.

Note: The loop stops when the C command in the macro cannot find a space to delete.

You Type: .CONVERT_TO_LINES * [ENTER]
Screen Shows:
```
.CONVERT_TO_LINES *
 LINE
 LINE
 LINE
```

Display the contents of the edit buffer.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
 LINE
 ONE
 LINE
 TWO
 LINE
 THREE
```

Edit Quick Reference Summary

EDIT — OS-9 loads the editor and starts it without creating any read or write files. Perform text-file operations by opening files after the editor is running.

EDIT *newfile* — OS-9 loads the editor and starts it. If *newfile* does not exist, Edit creates it and makes it the initial write file. Although this command does not create an initial read file, you can open read files after starting Edit.

EDIT *oldfile* — OS-9 loads the editor and starts it, making the initial read file *oldfile*. The editor creates a new file called SCRATCH as the initial write file. When the edit session is complete, Edit deletes *oldfile* and renames SCRATCH to *oldfile*.

EDIT *oldfile newfile* — OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called *newfile* as the initial write file.

Edit Commands

.MACRO — Executes the macro specified by the name following the period (.).

! — Places comments inside a macro, and ignores the remainder of the command line.

□ — Inserts a line before the current position of the edit pointer.

ENTER — Moves the edit pointer to the next line, and displays it.

+n — Moves the edit pointer forward *n* lines and displays the line.

-n — Moves the edit pointer backward *n* lines and displays the line.

+0 — Moves the edit pointer to the last character of the line.

-0	Moves the edit pointer to the first character of the current line and displays it.
>*n*	Moves the edit pointer forward *n* characters.
<*n*	Moves the edit pointer backward *n* characters.
CTRL 7 or □ for external terminals	Moves the edit pointer to the beginning of the text.
/	Moves the edit pointer to the end of the text.
[commands] *n*	Repeats the sequence of commands between the two brackets *n* times.
:	Skips to the end of the innermost loop or macro if the fail flag is not on.
A*n*	Sets the SEARCH/CHANGE anchor to Column *n*, restricting searches and changes to those strings starting in Column *n*. This command remains in effect for the current command line.
A0	Returns the anchor to the normal mode of searching so that strings are found regardless of the column in which they start.
B*n*	Makes buffer *n* the primary buffer.
C*n str1 str2*	Changes the next *n* occurrences of *str1* to *str2*.
D*n*	Deletes *n* lines.
E*n str*	Extends (adds the string to the end of) the next *n* lines.
G*n*	Gets *n* lines from the secondary buffer, starting from the top. Inserts the lines before the current position in the primary buffer.
I*n str*	Inserts a line containing *n* copies of the string before the current position of the edit pointer.
K*n*	Kills *n* characters starting at the current position of the edit pointer.
L*n*	Lists (displays) the next *n* lines, starting at the current position of the edit pointer.

M*n*	Changes workspace (memory) size to *n* bytes.
P*n*	Puts (moves) *n* lines from the position of the edit pointer in the primary buffer to the position of the edit pointer in the secondary buffer.
Q	Quits editing (and terminates editor). If you specified a file(s) when you entered Edit, Buffer 1 is written to the output file. The remainder of the input file is copied to the output file. All files are closed.
R*n*	Reads *n* lines from the buffer's input file.
S*n str*	Searches for the next *n* occurrences of the string.
T*n*	Tabs to Column *n* of the present line. If *n* is greater than the line length, Edit extends the line with space.
U	Unextends (truncates) a line at the current position of the edit pointer.
V*mode*	Turns the verify mode on or off.
W*n*	Writes *n* lines to the buffer's output file.
X*n*	Displays *n* lines that precede the edit position. The current line is counted as the first line.

Pseudo Macros

.CHANGE *n str1 str2*	Changes *n* occurrences of *str1* to *str2*.
.DEL *str*	Deletes the macro specified by *str*.
.DIR	Displays the directory of buffers and macros.
.EOB	Tests for the end of the buffer.
.EOF	Tests for the end of the file.
.EOL	Tests for the end of the line.
.F	Exits the innermost loop or macro and sets the fail flag.
.LOAD *str*	Loads macros from the path specified in the string.

.MAC *str*	Opens the macro specified by the string for definition. If you give an empty string, Edit creates a new macro.
.NEOB	Tests for not end of buffer.
.NEOF	Tests for not end of file.
.NEOL	Tests for not end of line.
.NEW	Writes all lines up to the current line to the initial output file, and then attempts to read an equal amount of text from the initial input file. The text read-in is appended to the end of the edit buffer.
.NSTR *str*	Tests to see if *string* does not match the characters at the current position of the edit pointer.
.READ *str*	Opens an OS-9 text file for reading, using *string* as the pathlist.
.S	Exits the innermost loop or macro and succeeds (clears the fail flag).
.SEARCH *n str*	Searches for *n* occurrences of *str*.
.SAVE *str1 str2*	Saves the macros specified in *str1* on the file specified by the pathlist in *str2*.
.SHELL *command line*	Calls OS-9 shell to execute the command line.
.SIZE	Displays the size of memory used and the amount of memory available in the workspace.
.STAR *n*	Tests to see if *n* equals asterisk (infinity).
.STR *str*	Tests to see if *string* matches the characters at the current position of the edit pointer.
.WRITE *str*	Opens an OS-9 text for writing, using *str* as a pathlist.
.ZERO *n*	Tests *n* to see if it is zero.
[	Starts at a macro loop; press CTRL 8.
]	Ends at a macro loop; press CTRL 9.

[^]	Moves edit pointer to beginning of buffer; press CTRL 7.

Editor Error Messages

BAD MACRO NAME	You did not begin the first line in a macro with a legal name. You can close the definition of a macro after you give it a legal name.
BAD NUMBER	You have entered an illegal numeric parameter, probably a number greater than 65,535.
BAD VAR NAME	You have specified an illegal variable name, omitted the variable name, or included a $ or # character in the commands parameter list.
BRACKET MISMATCH	You have not entered brackets in pairs or the brackets are nested too deeply.
BREAK	You pressed CTRL C or E to interrupt the editor. After printing the error message, the editor returns to command entry mode.
DUPL MACRO	You attempted to close a macro definition with an existing macro name. Rename the macro before trying to close its definition.
END OF FILE	You are at the end of the edit buffer.
FILE CLOSED	You tried to write to a file that is not open. Either specify a write file when starting the editor from OS-9, or open an output file using the .WRITE pseudo macro.
MACRO IS OPEN	You must close the macro definition before using the command.
MISSING DELIM	The editor could not find a matching delimiter to complete the string you specified. You must put the entire string on one line.
NOT FOUND	The editor cannot find the specified string or macro.

UNDEFINED VAR You used a variable that is not specified in the macro's definition parameter list. A variable parameter can be used only in the macro in which it is declared.

WHAT ?? The editor does not recognize a command. You typed a command that does not exist or misspelled a name.

WORKSPACE FULL The buffer did not have room for the text you want to insert. Increase the workspace, or remove some text.

OS-9 Error Codes

The following table shows OS-9 error codes in hexadecimal and decimal. Error codes other than those listed are generated by programming languages or user programs.

OS-9 Error Codes

Code HEX	Code DEC	Code Meaning
$01	001	UNCONDITIONAL ABORT. An error occurred from which OS-9 cannot recover. All processes are terminated.
$02	002	KEYBOARD ABORT. You pressed [BREAK] to terminate the current operation.
$03	003	KEYBOARD INTERRUPT. You pressed [SHIFT][BREAK] either to cause the current operation to function as a background task with no video display or to cause the current task to terminate.
$B7	183	ILLEGAL WINDOW TYPE. You tried to define a text type window for graphics or used illegal parameters.
$B8	184	WINDOW ALREADY DEFINED. You tried to create a window that is already established.
$B9	185	FONT NOT FOUND. You tried to use a window font that does not exist.
$BA	186	STACK OVERFLOW. Your process (or processes) requires more stack space than is available on the system.
$BB	187	ILLEGAL ARGUMENT. You have used an argument with a command that is inappropriate.
$BD	189	ILLEGAL COORDINATES. You have given coordinates to a graphics command which are outside the screen boundaries.

Code HEX	Code DEC	Code Meaning
$BE	190	INTERNAL INTEGRITY CHECK. System modules or data are changed and no longer reliable.
$BF	191	BUFFER SIZE IS TOO SMALL. The data you assigned to a buffer is larger than the buffer.
$C0	192	ILLEGAL COMMAND. You have issued a command in a form unacceptable to OS-9.
$C1	193	SCREEN OR WINDOW TABLE IS FULL. You do not have enough room in the system window table to keep track of any more windows or screens.
$C2	194	BAD/UNDEFINED BUFFER NUMBER. You have specified an illegal or undefined buffer number.
$C3	195	ILLEGAL WINDOW DEFINITION. You have tried to give a window illegal parameters.
$C4	196	WINDOW UNDEFINED. You have tried to access a window that you have not yet defined.
$C8	200	PATH TABLE FULL. OS-9 cannot open the file because the system path table is full.
$C9	201	ILLEGAL PATH NUMBER. The path number is too large, or you specified a non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL. Your system cannot handle an interrupt request, because the polling table does not have room for more entries.
$CB	203	ILLEGAL MODE. The specified device cannot perform the indicated input or output function.
$CC	204	DEVICE TABLE FULL. The device table does not have enough room for another device.

Code HEX	Code DEC	Code Meaning
$CD	205	ILLEGAL MODULE HEADER. OS-9 cannot load the specified module because its sync code, header parity, or cyclic redundancy code is incorrect.
$CE	206	MODULE DIRECTORY FULL. The module directory does not have enough room for another module entry.
$CF	207	MEMORY FULL. Process address space is full or your computer does not have sufficient memory to perform the specified task.
$D0	208	ILLEGAL SERVICE REQUEST. The current program has issued a system call containing an illegal code number.
$D1	209	MODULE BUSY. Another process is already using a non-shareable module.
$D2	210	BOUNDARY ERROR. OS-9 has received a memory allocation or deallocation request that is not on a page boundary.
$D3	211	END OF FILE. A read operation has encountered an end-of-file character and has terminated.
$D4	212	RETURNING NON-ALLOCATED MEMORY. The current operation has attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT. The file structure of the specified device is damaged.
$D6	214	NO PERMISSION. The attributes of the specified file or device do not permit the requested access.
$D7	215	BAD PATH NAME. The specified pathlist contains a syntax error, for instance an illegal character.
$D8	216	PATH NAME NOT FOUND. The system cannot find the specified pathlist.

Code HEX	Code DEC	Code Meaning
$D9	217	SEGMENT LIST FULL. The specified file is too fragmented for further expansion.
$DA	218	FILE ALREADY EXISTS. The specified filename already exists in the specified directory.
$DB	219	ILLEGAL BLOCK ADDRESS. The file structure of the specified device is damaged.
$DC	220	PHONE HANGUP - DATA CARRIER DETECT LOST. The data carrier detect is lost on the RS-232 port.
$DD	221	MODULE NOT FOUND. The system received a request to link a module that is not in the specified directory.
$DF	223	SUICIDE ATTEMPT. The current operation has attempted to return to the memory location of the stack.
$E0	224	ILLEGAL PROCESS NUMBER. The specified process does not exist.
$E2	226	NO CHILDREN. The system has issued a *wait service* request but the current process has no dependent process to execute.
$E3	227	ILLEGAL SWI CODE. The system received a software interrupt code that is less than 1 or greater than 3.
$E4	228	PROCESS ABORTED. The system received a signal Code 2 to terminate the current process.
$E5	229	PROCESS TABLE FULL. A fork request cannot execute because the process table has no room for more entries.
$E6	230	ILLEGAL PARAMETER AREA. A fork call has passed incorrect high and low bounds.
$E7	231	KNOWN MODULE. The specified module is for internal use only.

Code HEX	Code DEC	Code Meaning
$E8	232	INCORRECT MODULE CRC. The cyclic redundancy code for the module being accessed is bad.
$E9	233	SIGNAL ERROR. The receiving process has a previous, unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE. The system cannot locate the specified module.
$EB	235	BAD NAME. The specified device, file, or module name is illegal.
$EC	236	BAD MODULE HEADER. The specified module header parity is incorrect.
$ED	237	RAM FULL. No free system random access memory is available: the system address space is full, or there is no physical memory available when requested by the operating system in the system state.
$EE	238	UNKNOWN PROCESS ID. The specified process ID number is incorrect.
$EF	239	NO TASK NUMBER AVAILABLE. All available task numbers are in use.

Device Driver Errors

I/O device drivers generate the following error codes. In most cases, the codes are hardware-dependent. Consult your device manual for more details.

Code HEX	Code DEC	Code Meaning
$F0	240	UNIT ERROR. The specified device unit doesn't exist.
$F1	241	SECTOR ERROR. The specified sector number is out of range.
$F2	242	WRITE PROTECT. The specified device is write-protected.

Code HEX	Code DEC	Code Meaning
$F3	243	CRC ERROR. A cyclic redundancy code error occurred on a read or write verify.
$F4	244	READ ERROR. A data transfer error occurred during a disk read operation, or there is a SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR. An error occurred during a write operation.
$F6	246	NOT READY. The device specified has a *not ready* status.
$F7	247	SEEK ERROR. The system attempted a seek operation on a non-existent sector.
$F8	248	MEDIA FULL. The specified media has insufficient free space for the operation.
$F9	249	WRONG TYPE. An attempt is made to read incompatible media (for instance an attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY. A non-shareable device is in use.
$FB	251	DISK ID CHANGE. You changed diskettes when one or more files are open.
$FC	252	RECORD IS LOCKED-OUT. Another process is accessing the requested record.
$FD	253	NON-SHARABLE FILE BUSY. Another process is accessing the requested file.

Color Computer 2 Compatibility

Color Computer 3 OS-9 Level Two provides compatibility with the Color Computer 2 and OS-9 Level One by letting you use the video display in the Alphanumeric mode (including *Semigraphic* box graphics) and in the Graphics mode. To control the display, it has many built-in functions that you activate using ASCII control characters. Any program written in a language using standard output statements (such as PUT in BASIC) can use these functions. Color Computer BASIC09 has a Graphics Interface Module that can automatically generate most of these codes using BASIC09 RUN statements.

The Color Computer's display system uses a separate memory area for each Display mode. Therefore, operations on the Alpha display do not affect the Graphics display and vice-versa. You can select either display with software control. (See *Getting Started With Extended Color BASIC* for more detailed information.)

The system interprets 8-bit characters sent to the display according to their numerical values, as shown in this chart:

Character Range (Hex)	**Mode/Function**
00 - 0E	Alpha—Cursor and screen control.
0F - 1D	Graphics—Drawing and screen control.
1B	Alpha, Graphics—Changing Palette colors. Alpha mode: 1B 31 2 h change cursor color 1B 31 c h change foreground color 1B 31 d h change background color where h is a hex number from 0 to 3F (0 to 63 decimal) which determines the color.

Character Range (Hex)	Mode/Function
	Graphics mode: 1B 31 pr h — changes foreground/background color where pr is a palette register # (0 - F, hex) where h is a hex number from 0 to 3F (0 to 63 decimal) which determines the color.
20-5F	Alpha—Uppercase characters.
60 - 7F	Alpha—Lowercase characters.
80 - FF	Alpha—Semigraphic patterns.

The device driver CC3IO calls a subroutine module named VDGInt to handle all text and graphics for the Color Computer 2 compatibility mode.

Alpha Mode Display

The Alpha mode is the standard operational mode. Use it to display alphanumeric characters and semigraphic box graphics. Use it also to simulate the operation of a typical computer terminal with functions for scrolling, cursor positioning, clearing the screen, deleting lines, and so on.

The Alpha mode assumes that each 8-bit code the system sends to the display is an ASCII character. If the high-order bit of the code is clear, the system displays the appropriate alphanumeric character. If the high-order bit is set, OS-9 generates a Semigraphic 6 graphics box. See *Getting Started With Extended Color BASIC* for an explanation of semigraphic functions.

The standard 32-column Alpha mode display is handled by the I/O subroutine module VDGInt. CC3IO calls this module (included in the standard boot file) to process all text and semigraphic output.

The following chart provides codes for screen display and cursor control. You can use the functions from the OS-9 system prompt by typing DISPLAY, followed by the appropriate codes. For instance, to clear the screen, type:

```
display 0c ENTER
```

To position the cursor at column 16, Line 5 and display the word HELLO, type:

```
display 02 30 25 48 45 4c 4c 4f ENTER
```

You can also use the following codes in a language, such as BASIC09. To do so, use decimal numbers with the CHR$ function, such as:

```
print chr$(02);chr$(48);chr$(37);chr$(72)
;chr$(69);chr$(76);chr$(76);chr$(79)
```

Using Alpha Mode Controls with Windows

The control functions in the following chart also function properly under the high resolution windowing systems. References to "screen" are also references to windows.

Alpha Mode Command Codes

Hex Control Code	Decimal Control Code	Name/Function
$01	01	HOME—Returns the cursor to the upper left corner of the screen.
$02	02	CURSOR XY—Moves the cursor to character X of line Y. To arrive at the values for X and Y, add 20 hexadecimal to the location where you want to place the cursor. For example, to position the cursor at Character 5 of Line 10 (hexadecimal A), do these calculations: 5 + 20 = 25 hexadecimal 0A + 20 = 2A hexadecimal The two coordinates are $25 and $2A.
$03	03	ERASE LINE—Erases all characters on the line occupied by the cursor.
$04	04	CLEAR TO END OF LINE—Erases all characters from the cursor position to the end of the line.

Hex Control Code	Decimal Control Code	Name/Function
$05	05	CURSOR ON-OFF—Allows alteration of the cursor based on the value of the next character. Codes are as follow:

Hex	Dec	Char	Function	Default Color
$20	32	space	Cursor OFF	
$21	33	!	Cursor ON	Blue
$22	34	"	Cursor ON	Black
$23	35	#	Cursor ON	
$24	36	$	Cursor ON	
$25	37	%	Cursor ON	
$26	38	&	Cursor ON	
$27	39	'	Cursor ON	
$28	40	(	Cursor ON	
$29	41	)	Cursor ON	
$2A	42	*	Cursor ON	

$06	06	CURSOR RIGHT—Moves the cursor to the right one character position.
$07	07	BELL—Sounds a bell (beep) through monitor speaker.
$08	08	CURSOR LEFT—Moves the cursor to the left one character position.
$09	09	CURSOR UP—Moves the cursor up one line.
$0A	10	CURSOR DOWN (linefeed)—Moves the cursor down one line.
$0C	12	CLEAR SCREEN—Erases the entire screen, and homes the cursor (positions it at the upper left corner of the screen).
$0D	13	RETURN—Returns the cursor to the leftmost character on the line.
$0E	14	DISPLAY ALPHA—Switches the screen from Graphic mode to Alphanumeric mode.

Graphics Mode Display

Use the Graphics mode to display high-resolution 2- or 4-color VDG graphics. The Graphics mode includes commands to set color, plot and erase individual points, draw and erase lines, position the graphics cursor, and draw circles.

You must execute the *display graphics* command before using any other Graphics mode command. This command displays the graphics screen and sets a display format and color.

The first time you enter the display graphics command, OS-9 allocates a 6144-byte display memory. There must be at least that much contiguous free memory available. (You can use MFREE to check free memory.) The system retains the display memory until you give the *end graphics* command, even if the program that initiated the Graphics mode finishes. Always use the end graphics command to release the display memory when you no longer need the Graphics mode.

Graphics mode supports two basic formats. The 2-color format has 256 horizontal by 192 vertical points (G6R mode). The 4-color format has 128 horizontal by 192 vertical points (G6C mode). Either mode provides both color sets. Regardless of the resolution of the selected format, all Graphics mode commands use a 256 by 192 point coordinate system. The X and Y coordinates are always positive numbers. Point 0,0 is the lower left corner of screen.

Many commands use an invisible *graphics cursor* to reduce the output required to generate graphics. You can explicitly set this cursor to any point by using the *set graphics cursor* command. You can also use any other commands that include x,y coordinates (such as *set point*) to move the graphics cursor to the specified position.

Any graphics function that *draws* on the graphics screen requires that the VDGInt module is loaded into memory during the system boot.

Graphics Mode Selection Codes

Code	Format
00	256 x 192 two-color graphics
01	128 x 192 four-color graphics

Color Set and Foreground Color Selection Codes

	2-Color Format			4-Color Format	
	Char	Back-ground	Fore-ground	Back-ground	Fore-ground
Color Set 0	00	Black	Black	Green	Green
	01	Black	Green	Green	Yellow
	02			Green	Blue
	03			Green	Red
Color Set 1	04	Black	Black	Buff	Buff
	05	Black	Buff	Buff	Cyan
	06			Buff	Magenta
	07			Buff	Orange
Color Set 2	08			Black	Black
	09			Black	Dark Green
	10			Black	Med. Green
	11			Black	Light Green
Color Set 3	12			Black	Black
	13			Black	Green
	14			Black	Red
	15			Black	Buff

Graphics Mode Control Commands

Hex Control Code	Decimal Control Code	Name/Function
$0F	15	DISPLAY GRAPHICS—Switches the screen to the Graphics mode. Use this command before any other graphics commands. The first time you use it, the system assigns a 6-kilobyte display buffer for graphics. If 6K of contiguous memory isn't available, OS-9 displays an error. Follow the *display graphics* command with two characters specifying the Graphics mode and color/color set, respectively.
$10	16	PRESET SCREEN—Presets the entire screen to the color code passed by the next character.

Hex Control Code	Decimal Control Code	Name/Function
$11	17	SET COLOR—Sets the foreground color (and color set) to the color specified by the next character but does not change the Graphics mode.
$12	18	END GRAPHICS—Disables the Graphics mode, returns the 6K byte graphics memory area to OS-9 for other use, and switches to Alpha mode.
$13	19	ERASE GRAPHICS—Erases all points by setting them to the background color, and positions the graphics cursor at the desired position.
$14	20	HOME GRAPHICS CURSOR—Moves the graphics cursor to coordinates 0,0 (the lower left corner).
$15	21	SET GRAPHICS CURSOR—Moves the graphics cursor to the given x,y coordinates. For x and y, the system uses the binary value of the two characters that immediately follow.
$16	22	DRAW LINE—Draws a line in the foreground color from the graphics cursor position to the given x,y coordinates. For x and y, the system uses the binary value of the two characters that immediately follow. The graphics cursor moves to the end of the line.
$17	23	ERASE LINE—Operates the same as the *draw line* function, except that OS-9 draws the line in the background color, thus erasing the line.
$18	24	SET POINT—Sets the pixel at point x,y to the foreground color. For x and y, the system uses the binary values of the two characters that immediately follow. The graphics cursor moves to the point set.

Hex Control Code	Decimal Control Code	Name/Function
$19	25	ERASE POINT—Operates the same as the *set point* function, except that OS-9 draws the point in the background color, thus erasing the point.
$1A	26	DRAW CIRCLE—Draws a circle in the foreground color using the graphics cursor as the center point and using the the binary value of the next character as the radius.
$1C	28	ERASE CIRCLE—Operates the same as the *draw circle* function, except that OS-9 draws the circle in the background color, thus erasing the circle.
$1D	29	FLOOD FILL—*paints* with the foreground color, starting at the graphics cursor position and extending over adjacent pixels having the same color as the pixel under the graphics cursor.

Note: When you call FILL the first time, it requests allocation of a 512-byte stack for the fill routine. The system does not return this memory until you terminate graphics with the *end graphics* command.

Note: The chart uses hexadecimal codes for compatibility with the OS-9 DISPLAY command.

Display Control Codes Summary

1st Byte Dec	1st Byte Hex	2nd Byte	3rd Byte	Function
00	00			Null
01	01			Home alpha cursor
02	02	Column + 32	Row + 32	Position alpha cursor
03	03			Erase line

1st Byte Dec	1st Byte Hex	2nd Byte	3rd Byte	Function
04	04			Erase to End of line
05	05	Cursor Code		Alter Cursor
06	06			Move cursor right
07	07			Sound terminal bell
08	08			Move cursor left
09	09			Move cursor up
10	0A			Move cursor down
11	0B			Erase to End of Screen
12	0C			Clear screen
13	0D			Carriage return
14	0E			Select Alpha mode
15	0F	Mode	Color Code	Select Graphics mode
16	10	Color Code		Preset screen
17	11	Color Code		Select color
18	12			Quit Graphics mode
19	13			Erase screen
20	14			Home Graphics cursor
21	15	X Coord	Y Coord	Move graphics cursor
22	16	X Coord	Y Coord	Draw line to x/y
23	17	X Coord	Y Coord	Erase line to x/y
24	18	X Coord	Y Coord	Set point at x/y
25	19	X Coord	Y Coord	Clear point at x/y
26	1A	Radius		Draw circle
28	1C	Radius		Erase circle
29	1D			Flood Fill

OS-9 Keyboard Codes

Key Definitions With Hexadecimal Values

NORM	SHFT	CTRL	NORM	SHFT	CTRL	NORM	SHFT	CTRL
0 30	0 30	--	@ 40	60	NUL 00	p 70	P 50	DLE 10
1 31	! 21	| 7C	a 61	A 41	SOH 01	q 71	Q 51	DC1 11
2 32	" 22	00	b 62	B 42	STX 02	r 72	R 52	DC2 12
3 33	# 23	~ 7E	c 63	C 43	ETX 03	s 73	S 53	DC3 13
4 34	$ 24	00	d 64	D 44	EOT 04	t 74	T 54	DC4 14
5 35	% 25	00	e 65	E 45	EMD 05	u 75	U 55	NAK 15
6 36	& 26	00	f 66	F 46	ACK 06	v 76	V 56	SYN 16
7 37	' 27	^ 5E	g 67	G 47	BEL 07	w 77	W 57	ETB 17
8 38	(28	[5B	h 68	H 48	BSP 08	x 78	X 58	CAN 18
9 39	) 29	] 5D	i 69	I 49	HT 09	y 79	Y 59	EM 19
: 3A	* 2A	00	j 6A	J 4A	LF 0A	z 7A	Z 5A	SUM 1A
; 3B	+ 2B	00	k 6B	K 4B	VT 0B			
, 2C	< 3C	{ 7B	l 6C	L 4C	FF 0C			
- 2D	= 3D	5F	m 6D	M 4D	DR 0D			
. 2E	> 3E	} 7D	n 6E	N 4E	CO 0E			
/ 2F	? 3F	\ 5C	o 6F	O 4F	CI 0F			

Function Keys

	NORM	SHFT	CTRL
BREAK	05	03	1B
ENTER	0D	0D	0D
SPACE	20	20	20
←	08	18	10
→	09	19	11
↓	0A	1A	12
↑	0C	1C	13

Appendix D

OS-9 Keyboard Control Functions

Key Definitions for Special Functions and Characters

Key Combination	Control Function or Character
[ALT]	Alternate key—Sets the high order bit on a character. Press [ALT] *char*.
[CTRL]	Use as a control key.
[BREAK] or [CTRL][E]	Stops the program currently executing.
[CTRL][-]	Generates an underscore (_).
[CTRL][,]	Generates a left brace ({).
[CTRL][.]	Generates a right brace (}).
[CTRL][/]	Generates a reverse slash (\).
[CTRL][BREAK]	Generates an end-of-file (EOF). This sequence is the same as pressing [ESC] on a standard terminal.
[←] or [CTRL][H]	Generates a backspace.
[SHIFT][←] or [CTRL][X]	Deletes the entire current line.
[SHIFT][BREAK] or [CTRL][C]	Interrupts the video display of a running program. This sequence reactivates the shell and then runs the program as a background task.
[CTRL][0]	Upper-/lowercase shift lock function.
[CTRL][1]	Generates a vertical bar (|) in reverse video.
[CTRL][3]	Generates a tilde (~) character.
[CTRL][7]	Generates an up arrow or caret (^).
[CTRL][8]	Generates a left bracket ([).
[CTRL][9]	Generates a right bracket (]).

Key Combination	Control Function or Character
CTRL A	Repeats the previous command line.
CTRL D	Redisplays the command line.
CTRL W	Temporarily halts output to the screen. Press any key to resume output.
CTRL CLEAR	Enable/Disable Keyboard mouse.
CLEAR	Change screens.
SHIFT CLEAR	Change screens in reverse order.
ENTER	

Index

ACIAPAK 5-6, 5-7, 6-96
active state 4-2
address 2-4
    memory 4-5
allocate memory for devices 6-55
alpha mode B-3
    select B-10
alphanumeric mode B-1
ampersand separator 3-6
append files 6-68
application program 1-3
arglist 6-2
ASCII 2-5
    control characters B-1
    convert 6-38
ASM 3-2
asterisk, editor 7-3
ATTR 2-10, 6-5
attribute 2-5, 2-8, 2-10, 6-5
auto-answer modem 6-96, 6-108

background
    color B-7
    process 3-7
    task 4-1
    screen 5-2
backspace 6-93
    character 6-94, 6-106, 6-107
    editor 7-2
    over line 6-93, 6-106
BACKUP 5-4, 6-7
backup a directory 6-39
BASIC09 2-5, 2-6, 3-13, B-1
baud rate 5-4, 5-5, 5-6, 6-96, 6-98, 6-109
begin a window 6-103
bell
    character 6-95, 6-108
    sound B-10
bit 2-1
    stop 5-5, 5-6
    user 2-11
bitmap 2-5
block
    number 4-5
    devices 1-2
bootstrap 5-1
    file 5-2
box graphics B-3
brackets 6-3
buffer 3-7, 7-2
    edit 7-1
    secondary 7-1
    text 7-1
BUILD 2-6, 3-10, 6-10, 6-71, 6-75
built-in commands 3-1, 3-11
byte 2-1

carriage return B-10
CC3Disk 5-1
CC3Go 5-2
CC3IO 5-1, B-2
chaining programs 6-44
change
    attributes 2-10, 2-11
    directory 6-12, 6-91, 6-84
    file name 6-84
    priority 3-12, 6-88
    system parameters 6-93
character
    ASCII 2-5
    delete 6-107
    devices 1-2
    backspace 6-94, 6-106, 6-107
    bell 6-95, 6-108
    delete line 6-94, 6-107
    dup 6-95, 6-107
    end-of-file 6-94, 6-107

character (*cont'd*)
end-of-record 6-94, 6-107
lowercase B-1
pause 6-95, 6-108
quit 6-95, 6-108
reprint 6-95, 6-107
terminate 6-95, 6-108
uppercase B-2
CHD 3-11, 6-12
check disk structure 6-25
child process 3-6, 4-2
CHX 3-11, 6-12
circle
draw B-9
erase B-9, B-10
clear
screen B-5
to end-of-line B-4
clock 5-2
cluster 2-4, 2-5
CMDS directory 5-1, 5-3, 5-4
CMP 6-14
COBBLER 6-16, 6-72
code
alpha mode control B-4
cursor B-5
object 2-7
position-independent 4-8
re-entrant 4-6
color
background B-7
foreground B-7, B-8
select B-10
set, graphics B-7
combine files 6-68
command
grouping 3-2, 3-9
help 6-51
interpreter 6-90
line 3-1, 3-2
parameters, editor 7-3
separator 3-1, 3-5
summary, editor 7-55
command codes
alpha mode B-4
graphics B-7
commandname 6-2
commands
ASM 3-2
ATTR 2-10, 2-11, 6-5
BACKUP 6-7
BUILD 2-6, 3-10, 6-10, 6-71, 6-75
built-in 3-11
CHD 3-11, 6-12
CHX 3-11, 6-12
CMP 6-14
COBBLER 6-16, 6-72
CONFIG 5-2, 5-3, 5-4, 6-18
COPY 2-3, 3-6, 4-8, 6-22
DATE 6-24
DCHECK 6-25
DEINIZ 6-30
DEL 6-31
DELDIR 2-3, 6-33
DIR 2-6, 2-9, 6-35
DISPLAY 6-38
DSAVE 6-39
DUMP 2-8, 6-72
ECHO 6-42
edit macro 7-28
editor 7-2
ERROR 5-2, 6-43
EX 3-11, 6-44
FORMAT 6-46
FREE 6-49
GET 2-6
HELP 6-51
i 3-11
IDENT 3-3, 6-52
INIZ 6-55
KILL 3-12, 6-56
LINK 6-58
LIST 2-3, 2-5, 2-8, 3-4, 6-59
LOAD 4-7, 6-61

commands (*cont'd*)
MAKDIR 2-3, 2-11, 6-63
MDIR 6-64
MERGE 6-68
MFREE 6-69
MODPATCH 6-70
MONTYPE 6-74
OS9GEN 5-2, 5-3, 6-76
p 3-12
PROCS 3-7, 4-2, 6-80
PUT 2-6
PWD 6-82
PXD 6-82
RENAME 6-84
RUNB 3-13
SEEK 2-6
SETIME 5-3, 6-86
SETPR 3-12, 6-88
SHELL 3-6, 6-90
t 3-12
TMODE 6-93
TUNEPORT 6-98
UNLINK 4-7, 4-8, 6-100
w 3-12
WCREATE 6-103
x 3-12
XMODE 5-4, 5-5, 5-7, 6-106
comment, in a program 3-12
compare files 6-14
concurrent
execution 3-5, 6-91
mode 3-10
process 3-9
task 4-1
CONFIG 5-2, 5-3, 5-4, 6-18
control
characters, ASCII B-1
keys, editor 7-2
convert to ASCII 6-38
COPY 2-3, 3-6, 4-8, 6-22
copy
a directory 6-39
diskettes 6-7
CPU 4-1
priority 6-88
CRC 2-7, 6-71
create
a directory 6-63
a file 6-10
OS9Boot 6-16, 6-76
process 3-6
system diskette 5-3, 5-4, 6-16, 6-18, 6-76
current
directory 4-4, 6-12
processes 6-80
cursor
on/off B-5
codes B-5
control B-1, B-4
graphics B-6, B-8
home B-4
move B-5, B-10
cyclic redundancy checksum 2-7

data format 2-1
data output, halt 7-3
data
redirect 3-4
input/output 1-2
passing 4-4
process 2-1
sending 2-1
transfer 2-1
DATE 6-24
date 2-5
set 6-86
day 6-2
DCHECK 6-25
deallocate a device 6-30
DEINIZ 6-30
DEL 6-31
delay, not ready 5-6
DELDIR 2-3, 6-33

delete
- a character 6-107
- a directory 6-33
- a line 7-3
- a memory module 4-7, 6-100
- files 6-31
- line character 6-94, 6-107
- lines, editor 7-10

descriptor
- device 1-2
- file 2-3

detach a device 6-30
device
- allocate memory 6-55
- block-oriented 1-2
- character 1-2
- deallocate 6-30
- descriptor 1-2, 5-1
- driver 1-2, 2-1, 5-1
- driver initialization 6-55
- name 2-12, 2-13
- window 2-12 - 2-13

devname 6-2
DIR 2-6, 2-9, 6-35
directory 2-2, 2-3
- attribute 2-8
- change 6-12, 6-91
- change name 6-84
- CMDS 5-1, 5-3, 5-4
- copy 6-39
- create 6-63
- current 4-4, 6-12
- delete 6-33
- list 6-35
- module 4-6
- ownership 2-8
- SYS 5-1, 5-4
- view 6-82
- working 6-12

dirname 6-2
disable echo 6-94, 6-107
disk
- cluster 2-4
- file 2-3, 2-4
- I/O 3-8
- initialization 6-46
- names 2-12
- ownership 2-8
- sector 2-4
- structure, check 6-25
- raw I/O 3-8
- unused sectors 6-49

diskette
- copy 6-7
- density 2-5
- tracks 2-5
- system 2-2

DISPLAY 6-38
display
- a directory 6-35
- current processes 6-80
- date and time 6-24
- error message 6-43
- execution directory 6-82
- file contents 6-59
- free memory 6-69
- graphics B-7
- help 6-51
- memory module names 6-64
- messages 6-42
- on next line 7-2
- text, editor 7-6
- unused disk sectors 6-49
- working directory 6-82

double density 2-5
draw
- a circle B-9
- a line B-8, B-10

drivers, device 1-2
DSAVE 6-39
DUMP 2-8, 6-72
dup character 6-95, 6-107
duplicate
- last line 6-95
- line 6-107

ECHO 6-42
echo 6-92
    enable/disable 6-94, 6-107
edit
    buffer 7-1
    commands, EDIT 7-5
    pointer 7-1, 7-2, 7-7
EDIT, editor 7-5
editor 7-1
    backspace 7-2
    command summary 7-55
    command syntax 7-4
    commands 7-2
    control keys 7-2
    delete lines 7-10
    error messages 7-59
    getting started 7-4
    insert lines 7-10
    interrupt 7-3
    numeric parameters 7-3
    quick reference 7-55
    searching 7-13
    substituting 7-13
    terminate 7-2
    text file operations 7-18
    using the asterisk 7-3
ellipsis 6-3
enable echo 6-94, 6-101
end graphics B-8
end-of-file
    terminate 7-2
    character 6-94, 6-101
end-of-line
    clearing B-4
    erase B-10
end-of-record character 6-94, 6-107
erase
    a circle B-9, B-10
    a line B-10
    graphics B-8
    line B-4, B-8, B-9
    point B-9
erase (*cont'd*)
    to end-of-line B-10
Errmsg 5-2
ERROR 5-2, 6-43
error 3-12, 6-92
    message file 5-2
    messages, editor 7-59
    output 6-91
    path 3-4
establish a directory 6-63
EX 3-11, 6-44
exclamation mark separator 3-8
execute
    a program 6-90
    permission 2-9, 2-10
execution
    concurrent 3-5, 6-91
    modifier 3-1, 3-3
    sequential 3-5, 3-6, 6-91

fields 2-6
file 2-2 - 2-4
    attribute 2-8
    change name 6-84
    compare 6-14
    copy 6-22
    create 6-10
    delete 6-31
    descriptor 2-3
    descriptor sector 2-5
    display contents 6-59
    load in memory 6-61
    merge 6-68
    manager 5-1
    OS9Boot 5-4
    ownership 2-8
    pointer 2-4
    procedure 2-6, 3-10, 3-11
    random access 2-6
    security 2-8
    single-user 2-8
    size 2-5

file (*cont'd*)
Startup 2-6, 5-1, 5-3, 5-4
text 2-5
filename 2-3, 6-2
fill portion of screen B-9
flood fill B-9, B-10
floppy disk names 2-12
fonts 5-2
foreground color B-7, B-8
fork 3-7, 4-6
request 4-3
FORMAT 6-46
FREE 6-49

generate messages 6-42
GET 2-6
getting started, editor 7-4
graphic window fonts 5-2
graphics B-1
color set B-7
command codes B-7
cursor B-6, B-8
mode, select B-10
end B-8
erase B-8
medium resolution B-6
VDG B-6
group 2-7
grouping, commands 3-9

halt data output 7-3
hardware 1-2
header
information 6-52
module 2-7, 3-3, 4-7
HELP 6-51
hex 6-2
hexadecimal code display 6-38
home
alpha cursor B-9
cursor B-4
hours 6-2

I-code 3-13
I/O
paths 3-4
transfers 3-8
raw 3-8
ID, process 4-4
IDENT 3-3, 6-52
images, pointer 5-2
immortal
process 6-91
shell 3-11
INIT 5-1
initialize
a disk 6-46
a window 6-103
INIZ 6-55
input 2-1
lines 3-12
path 3-4
redirect 6-91
standard 4-4
insert lines, editor 7-10
interpreter, commands 6-90
interprocess communication 3-7
interrupt editor 7-3
IOMAN 1-2, 1-3, 5-1

kernel 1-1, 1-2
keyboard 1-1
keyword 3-1 - 3-3
KILL 6-56
kill 3-12
a directory 6-33, 6-33
files 6-31

length
of video page 6-94
word 5-5, 5-6, 6-96, 6-109
line
backspace 6-93, 6-106
delete, editor 7-3
draw B-8, B-10
duplicate 6-95

line (*cont'd*)
duplication 6-107
erase B-4, B-8, B-9, B-10
syntax 6-1
linefeed 6-94, 6-107
lines, command 3-1
LINK 6-58
LIST 2-3, 2-5, 2-8, 3-4, 6-59
list
a directory 6-35
current processes 6-80
memory module names 6-64
segment 2-5
with program files 2-8
LOAD 4-7, 4-7, 6-61
lock a module 6-58
lockout 2-11
logical sector 2-3, 2-4
lowercase 6-93, 6-106
characters B-2

machine language 3-12
macro text editor 7-1
macros, edit 7-25
MAKDIR 2-11, 2-3, 6-63
management, memory 4-5
manager
pipe 1-2
random block 1-2
mark space 6-95, 6-108
MDIR 6-64
MDM kill 5-6, 5-7
medium resolution graphics B-6
memory
address 4-5
allocation 3-1
display free 6-69
load a file into 6-61
management 1-1, 4-5
size modifier 3-3
memory modules
lock 6-58
memory modules (*cont'd*)
unlink 6-100
deleting 4-7
display names 6-64
MERGE 6-68
messages with ECHO 6-42
messages, error 6-43
MFREE 6-69
minutes 6-2
MMU 4-5
mode, alpha B-3
mode
alphanumeric B-1
concurrent 3-10
semigraphic B-1
modem 1-1, 5-6, 5-7
auto-answer 6-108
name 2-12
modifier 3-1 - 3-3
execution 3-1, 3-3
memory size 3-3
redirection 3-5
modname 6-2
MODPAK 5-7
MODPATCH 6-70, 6-71, 6-72, 6-73
module 1-3
deleting memory 4-7
directory 4-6
header 2-7, 3-3, 4-7
header information 6-52
loading 4-7
lock in memory 6-58
primary 4-3
program 2-7
unlink 4-8
month 6-2
MONTYPE 6-74, 6-75
move cursor B-4, B-5, B-10
multiprogramming 4-1
multitasking 1-1

name
device 2-12, 2-13
modem 2-12

name (*cont'd*)
printer 2-12
program 3-3
terminal 2-12
next line, display 7-2
not ready delay 5-6
notations, syntax 6-1
null count 6-94, 6-107
number
priority 3-12
user 2-8, 4-4
numeric parameters, editor 7-3

object code 2-7
operating system 1-3
opts 6-2
OS9Boot 5-1, 5-4
create 6-16, 6-76
OS9Gen 5-2, 5-3, 6-72, 6-76
OS9p2 5-1
output 2-1
error 6-91
path 3-4, 4-4
redirect 3-11, 6-91
owner 2-5, 2-8

page length, video 6-107
pages 3-3
paint B-9
parameter 3-1, 4-4
change system 6-93
command editor 7-3
paramlist 6-2
parent process 4-2
parity 5-6, 5-7, 6-95, 6-108
passing data 4-4
pathlist 6-2
paths 2-1
I/O 3-4
output 4-4
standard 3-4, 6-93
patterns, semigraphic B-2
pause 6-94
character 6-95, 6-108
screen 6-107
permission 6-2
execute 2-9, 2-10
read 2-9, 2-10
write 2-9, 2-10
physical sector 2-4
PIC 4-8
pipe 1-2, 3-7, 5-1
pipelines 3-7, 3-8
PIPEMAN 5-1
Piper 5-1
point
erase B-9
set B-8, B-10
pointer
edit 7-1, 7-2
editor 7-7
file 2-4
images 5-2
port 1-2
port, RS-232 5-4, 6-109
position alpha cursor B-9
position-independent 2-7
code 4-8
prepare a disk 6-46
preset screen B-7
previous line repeat 7-2
primary module 4-3
PRINTER 5-1
printer 1-1, 1-2
name 2-12
test 6-98
priority
number 3-12
change 6-88
process 4-2, 4-4
procedure file 2-6, 3-10, 3-11
process
background 3-7
chaining 6-44
child 3-6
create 3-6
current 6-80

process (*cont'd*)
data 2-1
fork 3-7
ID 4-4
memory size 6-91
priority 4-2, 4-4
properties 4-4
sibling 4-3
state 4-2
terminate 6-56
time sharing 4-1
processor time 4-1
procID 6-2
PROCS 3-7, 4-2, 6-80
program
application 1-3
chaining 6-44
comments in 3-12
execution 6-90
modules 2-7
name 3-3
size 3-3
prompt 3-12
prompting 6-92
properties, process 4-4
public 2-9, 2-10
PUT 2-6
PWD 6-82
PXD 6-82

quick reference, editor 7-55
quit character 6-95, 6-108

RAM 4-5
random access 1-2
files 2-6
random block file manager 1-2
rate, baud 5-4, 5-5, 5-6, 6-96, 6-98, 6-109
raw I/O 3-8
RBF 5-1
read 2-1, 2-11, 2-4
permission 2-10, 2-9
readers 3-8
record 2-2, 2-6
lockout 2-11
redirect
data 3-4
input 6-91
output 6-91
redirection 3-1
modifiers 3-4, 3-5
output 3-11
symbols 3-5
re-entrant code 4-6
remove
directory 6-33
files 6-31
memory module 6-100
RENAME 6-84
repeat previous line 7-2
reprint character 6-95, 6-107
reserved characters 3-3
ROOT 2-2, 2-3
route data 3-4
RS-232 5-1, 5-4, 6-96, 6-109
run-time module 3-12
RUNB 3-13

SAVE 6-72
SCF 5-1
screen
alpha B-5
background 5-2
clear B-5
control B-1
pause 6-94, 6-107
preset B-7
scroll pause 6-94, 6-107
searching, editor 7-13
secondary buffer 7-1
seconds 6-2
sector 2-4
copy 6-7
displayed unused 6-49
file descriptor 2-5
logical 2-3

security
    file 2-8
    permission 6-5
SEEK 2-6
segment list 2-5
select
    alpha mode B-10
    color B-10
    graphics mode B-10
semicolon, sequential
  execution 3-6
semigraphic
    mode B-1
    patterns B-2
sending data 2-1
separator 3-1
    ampersand 3-6
    command 3-5
    exclamation mark 3-8
sequential execution 3-5, 3-6,
  6-91
set a window 6-103
set a point B-8, B-10
set priority 3-12
SETIME 5-3, 6-86
SETPR 6-88
share time 4-1
shell 1-3, 3-1–3-3, 3-8, 6-3
SHELL 3-6, 6-90
show
    a directory 6-35
    error message 6-43
    execution directory 6-82
    file contents 6-59
    free memory 6-69
    header information 6-52
    memory module
      names 6-64
    working directory 6-82
sibling processes 4-3
sign bit 2-2
simultaneous execution 3-5
single-user file 2-8
SIO 5-1
size
    file 2-5
    process memory 6-91
    program 3-3
slash in device names 2-13
sleeping 4-3
software fonts 5-2
sound bell B-10
standard input 4-4, 6-93
standard paths 3-4, 6-93
start a window 6-103
Startup 2-2, 2-6, 5-1, 5-3,
  5-4, 6-75
state 4-2, 4-2
Stdfonts 5-2
Stdpats 5-2
Stdptrs 5-2
stop bit 5-5, 5-6
string parameters, editor 7-4
subdirectory 2-3
    delete 6-33
submanager 1-2
subshell 3-10
substituting, editor 7-13
summary, commands 6-3, 6-4
super user 6-56
switch screen B-5
symbols, redirection 3-5
syntax 6-1
SYS directory 5-1, 5-4
system
    administrator 1-1
    date 6-86
    disk create 5-3, 5-4
    parameters 6-93
    priority 6-88
    time 6-86
system diskette 2-2
    create 6-16, 6-18, 6-76

task, background 4-1
term 1-1
TERM 5-1
TERM-VDG 6-96
TERM-WIN 6-109

terminal name 2-12
terminals 1-2
terminate
    a character 6-95, 6-108
    a process 6-56
    the editor 7-2
    on error 6-92
test delay loop 6-98
text 6-2, B-2
    buffers 7-1
    display, editor 7-6
    editing 7-1
    file operations, editor 7-18
    files 2-5
tick 4-1, 4-2
tickcount 6-2
time 2-5, 6-24
    sharing, process 4-1
    CPU 4-1
    processor 4-1
    set 6-86
timeslice 4-1, 4-2
TMODE 6-93
tracks 2-5
transfer, I/O 3-8
transferring data 3-7
TUNEPORT 6-98
turn on
    cursor B-5
    echo 6-92
    prompting 6-92
type 5-7
    ACIA 6-96, 6-108
    of window 6-103
    value 5-7

UNLINK 4-7, 4-8, 6-100
unused disk sectors 6-49
update mode 2-11
uppercase 6-93, 6-106
    characters B-2

user
    bit 2-11
    number 2-8, 4-4

value 6-2
    type 5-7
variable 6-1
VDG graphics B-6
video 1-1
    page length 6-94, 6-107
view
    current processes 6-80
    error messages 6-43
    working directory 6-82

waiting state 4-3
WCREATE 6-103
window 5-2
    alpha mode controls B-3
    descriptor 2-12
    initialization 6-103
    type 6-103
word length 5-5, 5-6, 6-96, 6-109
working directory 6-12
write 2-1, 2-4, 2-11
    permission 2-9, 2-10

XMODE 5-4, 5-5, 6-106

year 6-2