TRS-80® Color Computer

Radio Shack

Radio Shack

Radio Shack

OS-9 Technical Information

OS-9 Program Development

OS-9 Commands

READ ME FIRST

All computer software is subject to change, correction, or improvement as the manufacturer receives customer comments and experiences. Radio Shack has established a system to keep you immediately informed of any reported problems with this software, and the solutions. We have a customer service network including representatives in many Radio Shack Computer Centers, and a large group in Fort Worth, Texas, to help with any specific errors you may find in your use of the programs. We will also furnish information on any improvements or changes that are "cut in" on later production versions.

To take advantage of these services, you must do three things:

(1) Send in the postage-paid software registration card included in this manual immediately. (Postage must be affixed in Canada.)

(2) If you change your address, you must send us a change of address card (enclosed), listing your old address exactly as it is currently on file with us.

(3) As we furnish updates or "patches", and you update your software, you must keep an accurate record of the current version numbers on the logs below. (The version number will be furnished with each update.)

Keep this card in your manual at all times, and refer to the current version numbers when requesting information or help from us. Thank you.

APPLICATIONS SOFTWARE VERSION LOG

OP. SYSTEM VERSION LOG

01.01.00

Change of address

NEW ADDRESS

Name _____

Company _____

Address _____

City _____ Phone ( ) ____ - _____

State _____ Zip _____

OLD ADDRESS

Name _____

Company _____

Address _____

City _____ Phone ( ) ____ - _____

State _____ Zip _____

Change of address

NEW ADDRESS

Name _____

Company _____

Address _____

City _____ Phone ( ) ____ - _____

State _____ Zip _____

OLD ADDRESS

Name _____

Company _____

Address _____

City _____ Phone ( ) ____ - _____

State _____ Zip _____

```
┌─────────┐
│ PLACE   │
│ STAMP   │
│ HERE    │
└─────────┘
```

Software Registration
Data Processing Dept.
P.O. Box 2910
Fort Worth, Texas 76113-9965

```
┌─────────┐
│ PLACE   │
│ STAMP   │
│ HERE    │
└─────────┘
```

Software Registration
Data Processing Dept.
P.O. Box 2910
Fort Worth, Texas 76113-9965

Getting Started With OS-9

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state

GETTING STARTED WITH OS-9

OS-9 OPERATING SYSTEM

**For The 64K TRS-80 Color Computer
includes an Editor and Assembler**

To Our Customers . . .

Congratulations on purchasing the OS-9 Disk Operating System for your 64K TRS-80 Color Computer. You'll find OS-9 powerful and simple to use. It is structured after the famous UNIX operating system that is used at many colleges and universities today.

How to Use This Manual . . .

This manual is written for beginners although experienced programmers will also find it useful in starting up OS-9. It explains the important things you need to know:

- How to run an easy test on your disk drives.

- How to start OS-9.

- How to use important commands.

- What to do in case of trouble.

For more detailed information on OS-9, you can read these OS-9 manuals:

OS-9 Commands. Explains the concepts and commands of OS-9.

OS-9 Program Development. Explains how to use:

- OS-9's text editor to enter programs or to prepare text such as letters and documents.

- OS-9's Assembler.*

- OS-9's Interactive Debugger to debug your assembly language programs.*

OS-9 Technical Information. Provides all the information necessary to install, maintain, expand or write assembly-language programs for OS-9.*

*These manuals assume that you are familiar with the 6809 architecture, instruction set, and assembly language.

Table of Contents

1 / WHAT IS OS-9?

What is OS-9? That's a simple question with an interesting answer. However, before you can understand OS-9, you need to understand operating systems.

What Is an Operating System?

An operating system acts as a manager for the computer. It sends information to the disk drives, printers and video. It manages the storage space on the disks and in memory. It also answers your commands.

This illustration shows the relationship between the operating system and the hardware. (Hardware is the computer, drives and printer.) It also shows how application programs and languages fit into the picture.
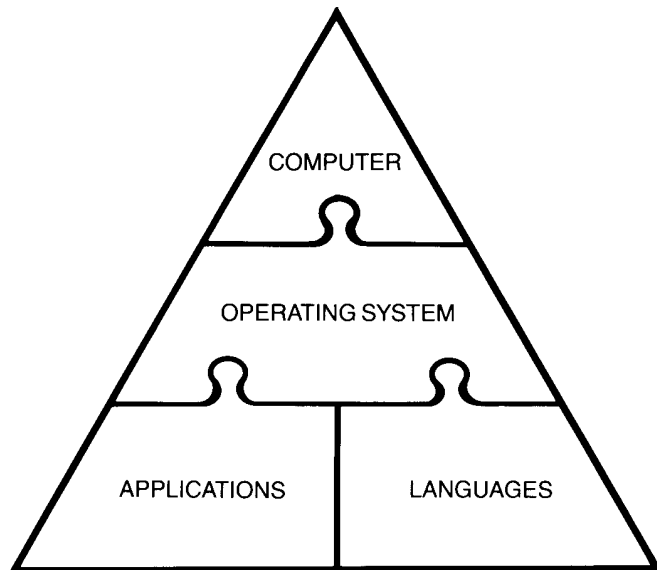
Application Programs are practical uses for the computer, such as creating and maintaining a mailing list.

Languages let you write your own application programs.

Back To OS-9

OS-9 is a versatile operating system for the 64K TRS-80 Color Computer. It is based on the UNIX operating system developed by Bell Laboratories Inc. UNIX is widely used on larger computers especially in colleges and universities.

OS-9 opens many new doors by expanding the Color Computer's capabilities. OS-9 offers sophisticated features that are normally available only in much larger computers. Among these are:

Multi-Level Filing System

Like most operating systems, OS-9 lets you store information on disk in a "file" and index these files with a directory. OS-9, however, goes one step further by letting you create a hierarchy of directories and files.

Multiuser/Multitasking Operation

Multiuser means that more than one person can use the system at the same time. The number of users is limited by the number of terminals. The TRS-80 Color Computer can have one terminal; this means that two people can use OS-9 at the same time. One person on the Color Computer and one on the terminal.

Multitasking means that two or more tasks (programs) can run at the same time. For example, with OS-9 you could print reports and enter information at the same time.

Device-independent Input/Output System

OS-9 uses a very efficient method for inputting and outputting information. It expects all input to come from the "standard input device" and all output go to the "standard output device." On the Color Computer, OS-9 expects all input to come from the keyboard/console and all output to go to the video display.

You can easily "redirect" the standard I/O devices to other devices such as printers or disks. This means that an OS-9 program needs only one output routine and one input routine. From there you can redirect them to another device. This saves time for the programmer and space on the disk because programs can be shorter.

This manual provides step-by-step instructions for getting OS-9 started on your Color Computer. It includes a sample session to help you become familiar with some of the most common OS-9 commands and features. We do not cover all OS-9 features here, but you can find detailed information in the other OS-9 manuals.

To run OS-9, you must have a 64K TRS-80 Color Computer that has at least one floppy disk drive. The OS-9 standard system disk includes modules to support the following TRS-80 Color Computer hardware:

- 64K RAM
- Keyboard
- Alphanumeric Video Display
- Color Graphics Display
- Disk Drives (1 or 2)
- Joysticks (1 or 2)
- Serial Printer*
- RS-232C Communications Port

* Optional supported hardware

We hope you enjoy your journey with OS-9!

2/BEFORE YOU START OS-9

Before you start OS-9, we suggest you run a simple test program to make sure your disk drives are still well "tuned." This test checks the speed of the drive. A disk drive speed should be about 300 RPM (rotations per minute). Even though we submit our disk drives to a rigorous quality assurance test before they are sent to you, the drive speed may begin to vary after use. This makes disks harder to read.

You can check your drive's speed at home with the following procedure:

1. Make sure that the disk system is properly connected to the computer. Turn on the TV. Next, turn on the computer and disk drives.

2. Insert the disk labeled OS-9 BOOT into Drive 0.

3. At the OK prompt, type:

   RUN "*" (ENTER)

   This starts the OS-9 Utility program.

4. The screen shows:

   b BOOT OS9
   t TEST DISK DRIVE

   Press (T) to start the disk drive test.

5. The test program prompts:

   ***SELECT DRIVE 0 - 3 OR "BREAK"**

   Enter a (0) to test Drive 0.

6. The screen shows the drive speed. The speed may change slightly during the test; this is normal. This test accepts speeds from 298.0 to 303.5 rpm. If the speed is in this range, the screen continues to display the speed.

If the speed is unacceptable, the screen changes color and displays one of these messages:

```
DISK SPEED NEEDS ADJUSTING, TOO FAST
```

or

```
DISK SPEED NEEDS ADJUSTING, TOO SLOW
```

If one of these messages appears, take your disk drive to your Radio Shack Service Center for adjustment.

To exit the test, press (BREAK) and the test prompt reappears. If you have only one drive, remove the OS-9 BOOT disk and place it in its protective sleeve.

If you have a second drive and wish to test it, remove the OS-9 BOOT disk from Drive 0 and place it in Drive 1. Press ① at the test prompt, and the screen shows the drive speed. Exit the test by pressing (BREAK) and remove the OS-9 BOOT disk.

Place the disk in its protective sleeve and store it in a safe place. You may want to run this test periodically. How often you decide to run it depends on how much you use your drives.

3 / STARTING UP OS-9

The way to start OS-9 depends on the version of your computer's disk ROM. The version number appears on the first line of the dialog when you start up your disk system:

```
DISK EXTENDED COLOR
BASIC 1.x      <---- Version Number
```

If your computer is Version 1.0, the startup procedure differs from that of Versions 1.1 and later. Versions 1.1 or later have the DOS command, which automatically boots OS-9. Version 1.0 does not and must use a special command.

Be sure your disk system is properly connected. Turn on the TV. Then turn on the disk drives and the computer.

Starting OS-9 With Version 1.0

1. Insert the disk labeled OS-9 BOOT into Drive 0.

2. At the OK prompt, type:

```
RUN "*"  (ENTER)
```

This starts the OS-9 Utility program.

3. The following appears:

```
b BOOT OS9
t TEST DISK DRIVE
```

Press ⑧ to boot OS-9.

4. The boot utility prompts:

```
INSERT OS9 DISKETTE
INTO DRIVE 0 AND PRESS A KEY
```

Remove the OS-9 BOOT disk and place the OS-9 SYSTEM MASTER disk in Drive 0. Press any key except break and OS-9 starts.

Starting Up with Version 1.1 or Later

1. Insert the disk labeled OS-9 SYSTEM MASTER into Drive 0.

2. At the OK prompt, type:

```
DOS (ENTER)
```

This starts OS-9. If the DOS command returns a syntax error (?SN ERROR), be sure you entered the command correctly. If DOS still returns the error, then you probably have version 1.0; use the previous procedure.

Entering the Date

After OS-9 displays its startup message, the time prompts displays:

```
        YY/MM/DD HH:MM:SS
TIME ?
```

Enter the date and time in the form shown. For example:

```
86/03/19 13:22 (ENTER)
```

sets the date as March 19, 1986 and the time as 1:22 pm. Enter the time in 24-hour notation; the seconds (:SS) are optional.

When you enter the date, the OS-9 prompt appears:

```
OS9:
```

OS-9 is now in control and ready to accept a command.

> **Note:** You should always keep the OS-9 System Disk in Drive 0 while running OS-9.

Turning Off the System

Before turning off your disk system, remove all disks from the drives. Turn off the printer (if connected), the disks, the computer and then the TV.

4 / FORMATTING DISKS AND MAKING BACKUPS

As you know from using the Color Disk BASIC, you need to "format" disks into a "file cabinet" orgainzation before using them. In the same manner, you must format disks before using them with OS-9.

OS-9's disk format, however, is slightly different from that of the Color Disk BASIC. Because of this, neither can read the other's disk.

This chapter shows how to format disks so that you can use them with OS-9. It also shows you how to make backup copies of them.

The OS-9 Boot disk uses Color Disk BASIC's format. To make a backup of it, use the DSKINI command to format a blank disk and the BACKUP command to make a copy of the Boot disk. Always store your original disks in a safe place.

Single-Drive Users

1. Be sure your computer and all peripherals (TV, disk drive, printer) are turned on. Start OS-9 as described earlier in the previous chapter.

2. Get a blank disk. Be sure the write-protect notch is *not* covered by a foil tab.

3. At the OS9: prompt, type:

```
format /D0 (ENTER)
```

Format displays the message:

```
COLOR COMPUTER DISK FORMATTER 1.2
FORMATTING DRIVE /D0
Y (YES) OR N (NO)
READY?
```

Remove your OS-9 SYSTEM MASTER disk and place the blank disk in Drive 0. Type Y.

4. After formatting the disk, Format prompts for a disk name. The name is not important at this point, so enter any letter. For example:

   ```
   S (ENTER)
   ```

   OS-9 begins verifying the disk. The screen shows the track number in hexadecimal notation as it is verified.

5. When the formatting is complete, the OS9: prompt appears.

   Note: You cannot make a backup on a disk with errors. If any errors occurred during the formatting, try again. Remove the blank disk from Drive 0 and replace the OS-9 SYSTEM MASTER disk. Repeat steps 2-5. If OS-9 still reports errors on the disk, use another disk.

6. Remove the formatted disk from Drive 0 and replace the OS-9 SYSTEM MASTER disk. Type:

   ```
   BACKUP /D0 #40K (ENTER)
   ```

   Backup prompts:

   ```
   READY TO BACKUP FROM /D0 TO /D0
   ?:
   ```

   Press (Y) to begin the backup.

7. The following prompt appears:

```
READY DESTINATION, HIT A KEY
```

Remove the OS-9 SYSTEM MASTER disk and place the formatted disk (destination disk) in Drive 0. Press any key. OS-9 displays the message:

```
S                        <---- (disk name)
  IS BEING SCRATCHED
OK ?:
```

Press Ⓨ and the backup begins.

8. Backup alternately prompts you to:

```
READY SOURCE, HIT A KEY
    Place the OS-9 SYSTEM MASTER disk in Drive 0
    and press any key.
    (source = master disk)
```

or

```
READY DESTINATION, HIT A KEY
    Place the formatted disk in Drive 0 and press any key.
    (destination = formatted disk)
```

9. When the backup is complete, the screen displays the number of sectors copied and and the number of sectors verified. These numbers should match.

Keep your OS-9 System backup disk in Drive 0. Store the original disk in a safe place. Use it only to make backups.

Two-Drive Users

1. Be sure your computer and all peripherals are turned on. Start OS-9 as described earlier in the previous chapter.

2. Get a blank disk. Be sure the write-protect notch is *not* covered by a foil tab. Insert the disk into Drive 1.

3. At the OS9: prompt, type:

   `format /D1` **(ENTER)**

   Format displays the message:

   ```
   COLOR COMPUTER DISK FORMATTER 1.2
   FORMATTING DRIVE /D1
   Y (YES) OR N (NO)
   READY?
   ```

   Press **(Y)**. After formatting the disk, Format prompts you for a disk name. The name is not important at this point, so enter any letter. For example:

   `S` **(ENTER)**

   OS-9 begins verifying the disk. The screen shows the track number in hexadecimal notation as it is verified.

4. When the formatting is complete, the OS9: prompt appears.

   Note: You cannot make a backup on a disk with errors. If any errors occurred during the formatting, try again. Remove the blank disk from Drive 0 and replace the OS-9 SYSTEM MASTER disk. Repeat steps 2-5. If OS-9 still reports errors on the disk, use another disk.

5. Type:

    BACKUP (ENTER)

    Backup prompts:

    READY TO BACKUP FROM /D0 TO /D1
    ?:

    Press (Y). OS-9 displays the message:

    S   (disk name)
      IS BEING SCRATCHED
    OK ?:

    Press (Y) and the backup begins.

6. When the backup is complete, the screen shows the number
    of sectors copied and the number of sectors verified. These
    numbers should match.

Remove your OS-9 SYSTEM MASTER disk from Drive 0 and
move the OS-9 System backup disk from Drive 1 to Drive 0.
Store the original disk in a safe place. Use it only to make
backups.

> **Note to Two-Drive Customers:** You can format disks to
> use in your second drive (data disks) by following steps
> 1-3. You can store information on the data disk instead of
> your OS-9 System Disk.

5 / EXPLORING THE OS-9 FILE SYSTEM

OS-9 stores information in disk "files," much as you might store a memo or other information in a file folder. The disk files, however, in addition to containing ordinary information such as memos, lists, lines of data, and the texts of documents, can also contain complete programs.

As an aid to organization, OS-9 also gives you the option of collecting groups of files in "directories," just as in an office you might organize files into categories and group them together in labeled file cabinet drawers. A directory might look like this:

```
                    ┌──────────────┐
                    │  directory   │
                    └──────┬───────┘
         ┌─────────────────┼─────────────────┐
   ┌─────┴─────┐     ┌─────┴─────┐     ┌─────┴─────┐
   │   file1   │     │  file 2   │     │   file3   │
   └───────────┘     └───────────┘     └───────────┘
```
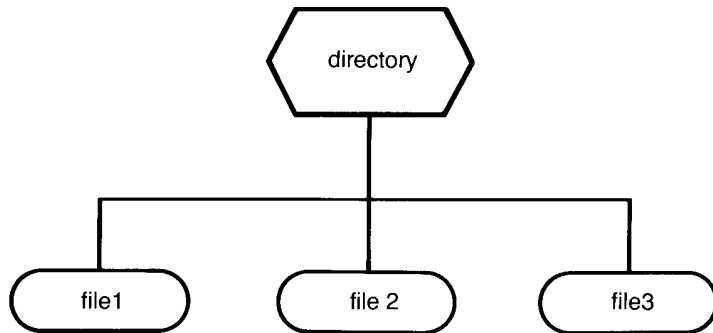
These directories are organized "hierarchically," similar to a tree or pyramid. This means that each OS-9 directory can contain another directory.

The OS-9 system starts at the "system device directory." This directory contains a directory entry for each device, such as the keyboard/display, disk drives, printer, and optional terminal. The system device directory looks like this:

```
                    ┌─────────────┐
                   ╱   System      ╲
                  ╱    Device       ╲
                  ╲    Directory    ╱
                   ╲───────────────╱
                           │
   ┌───────────┬───────────┼───────────┬───────────┐
   │           │           │           │           │
 ╱───╲       ╱────╲      ╱──────╲     ╱────╲      ╱────╲
╱  P  ╲     ╱  D0  ╲    ╱ TERM   ╲   ╱  D1  ╲    ╱  T1  ╲
╲     ╱     ╲      ╱    ╲        ╱   ╲      ╱    ╲      ╱
 ╲───╱       ╲────╱      ╲──────╱     ╲────╱      ╲────╱
```
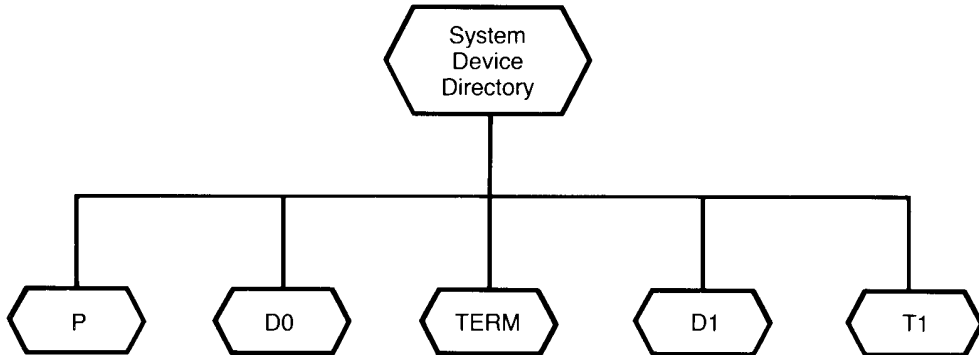
The P is the printer, D0 is the first disk drive, TERM is the keyboard/display, D1 is the second drive, and T1 is a terminal connected to the RS-232C serial port. (Even if you don't have the actual device, your directory still has the entry.)

The disk drives, (/D0 and /D1), are the only devices that can form their own "tree" by storing other directories and files. Each drive has a "root" directory which is the beginning of its tree.

The following illustration shows a typical OS-9 System with a disk in both drives:

```
                        System
                        Device
                        Directory

        P       D0              TERM            D1      T1

                D0                              D1
               ROOT                            ROOT
             DIRECTORY                       DIRECTORY

        SYS   DEFS  startup  CMDS  OS 9Boot      PAYROLL        MEMOS

             OS9Defs

  password errmssg motd  copy list dir del   HOURLY SALARIED   SALES  ACCT

                                                                chris

                                            a _ m  n _ z  a _ m  n _ z
```
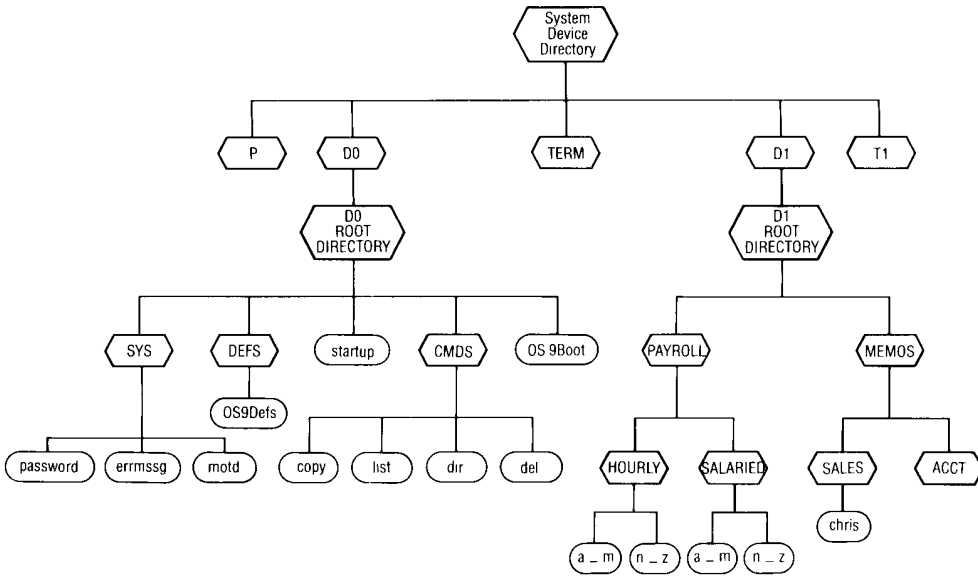
The disk in Drive 0 is the OS-9 System disk. Its root directory contains two files (startup and OS-9Boot) and three sub-directories (SYS, DEFS and CMDS). The sub-directories contain another level of files. All of these files are necessary to run OS-9.

The disk in Drive 1 is an imaginary data disk containing three levels of directories. OS-9 lets you organize your files with as many levels of directories as you need and have room for.

Notice that all device and directory names are capitalized and filenames are lower-case. This is a customary practice that lets you easily recognize directory names; however, it is not mandatory.

19

Before looking at the OS-9 directory; we suggest you type the following command to get into the upper/lower case mode. (OS-9 normally displays upper-case letters only.) At the OS-9 prompt:

```
tmode -upc
```

OS-9 now displays lower-case characters. The Color Computer displays upper-case characters as dark letters on a light background. Lower-case letters display in reverse mode, that is, light letters on a dark background.

This manual shows all commands and file names in lower-case and directory names in upper-case.

> **Note to Two-Drive Customers:** The examples in this chapter specify Drive 0. You can use Drive 1 by specifying /D1 instead of /D0 as shown.

To see the contents of the OS-9 System disk's root directory, type the following at the OS-9 prompt:

```
dir /D0 (ENTER)
```

OS-9 displays:

```
DIRECTORY OF . 00:27:37
OS9Boot    CMDS    SYS
DEFS       startup
```

This shows the first level directory on the OS-9 System disk. To see an example of a second level directory, look at the CMDS directory. Type:

```
dir /D0/CMDS (ENTER)
```

OS-9 lists the contents of the CMDS directory. The CMDS directory contains all the commands for OS-9.

OS-9 uses "pathnames" to locate files and directories. Pathnames describe the path to be taken. For example:

```
/D0/CMDS
```

20

tells OS-9 that "CMDS" is located in the D0 device directory.
Look at this pathname:

```
/D1/MYFILES/testprog
```

OS-9 looks for the file "testprog" in the directory MYFILES
which is a subdirectory of device /D1. We know that "testprog"
is a file because it is shown in lower-case letters.

You can position yourself in another directory by using the Chd
command. For example, to move to the DEFS directory, type:

```
chd /D0/DEFS (ENTER)
```

Now display the directory, using the Dir command:

```
dir (ENTER)
```

The Dir command always lists the contents of the current direc-
tory if another is not specified (for example dir /D0/CMDS).
Therefore, OS-9 displays the contents of the DEFS directory.

If you can't remember which directory you are in, you can find
out with the Pwd command (print working directory). Type:

```
pwd (ENTER)
```

OS-9 tells you that you are in the /D0/DEFS directory. Move
back to the /D0 root directory by typing:

```
chd /D0 (ENTER)
```

Creating and Deleting Your Own Directories

Now that you've seen some OS-9 files and directories, you
probably want to know how to create — and delete — your own.

Creating Directories

You use the Makdir command (make directory) to create directories. For example:

```
makdir BUSINESS (ENTER)
```

creates a directory in the current directory (/D0) called BUSINESS. You can use the Dir command and see that BUSINESS has been created. (Remember, we recommend using all uppercase letters for directory names.)

You can create directories to organize your files by projects, applications, users, and so on. Here are some sample directory names:

DEE   PAYROLL   GROSS.SALES.TEXAS
X10.PROJECT   TESTFILES   JIMS.FILES

You can also create subdirectories (directories that reside in another directory). For example:

```
makdir /D0/BUSINESS/PAYROLL (ENTER)
```

creates a directory called PAYROLL in the BUSINESS directory.

Deleting Directories

To delete a directory, use the Deldir command (delete directory). If you want to delete the PAYROLL directory, type this command:

```
deldir /D0/BUSINESS/PAYROLL (ENTER)
```

OS-9 asks:

```
LIST DIRECTORY, DELETE DIRECTORY,
OR QUIT ?
(L/D/Q)
```

Type (D) (ENTER) to delete the directory. OS-9 deletes all files in the PAYROLL directory and then deletes the PAYROLL directory.

22

Important Notes about Deleting Directories

Keep these points in mind when deleting directories:

- Before you delete a directory, be sure to move all important files to another directory or you will lose them. **The Deldir command removes all files in the directory being deleted.**

- All sub-directories of the directory being deleted will be lost.

- You cannot position yourself in a directory that is being deleted. This includes subdirectories of that directory. Position yourself in a directory higher in the hierarchal tree (use the Chd command to reposition).

Creating and Manipulating Files

Although you will normally create files with application programs, you can use the Build command to create simple files. To start the file, type:

```
build /DØ/BUSINESS/file1 (ENTER)
```

This creates a file called "file1" in the BUSINESS directory. The screen shows a question mark (?) indicating that OS-9 is waiting for you to insert information into the file. Type the following:

```
? This is the first file that
? we created,
? (ENTER)
```

The (ENTER) on the last line tells OS-9 to end the file and return the OS-9 prompt. The Dir command shows the new file:

```
dir /DØ/BUSINESS (ENTER)
```

Use the List command to display the contents of the file. For example:

```
list /DØ/BUSINESS/file1 (ENTER)
```

displays the following:

```
This is the first file that
we created,
```

The Copy command lets you duplicate a file. To duplicate /D0/BUSINESS/file1 to the file /D0/BUSINESS/file2, type:

```
copy /DØ/BUSINESS/file1 /DØ/BUSINESS/
file2 (ENTER)
```

This tells OS-9 to create a file called "file2" and copy the contents from "file1" to "file2." The first file (file1) is left untouched. To see exactly what the Copy command accomplished, list the contents of "file2." Type:

```
list /DØ/BUSINESS/file2 (ENTER)
```

OS-9 displays:

```
This is the first file that
we created,
```

Notice that "file2" is indeed a copy of "file1." You can also copy a file to another directory. For example:

```
copy /D0/BUSINESS/file1 /D0/file1 (ENTER)
```

copies the file /D0/BUSINESS/file1 to file /D0/file1. Even though the filenames are the same, the files are unique because their pathnames are different. Use the Dir command to see that "file1" exists in both directories.

OS-9 also lets you rename files:

```
rename /D0/file1 samplefile (ENTER)
```

This command changes the filename /D0/file1 to "samplefile." It is important to note that this does not duplicate the file, it simply changes the name. Therefore the file /D0/file1 does not exist after the above command is executed. Try listing /D0/file1:

```
list /D0/file1 (ENTER)
```

OS-9 returns an Error #216, which means that OS-9 could not find the file. Now list /D0/samplefile:

```
list /D0/samplefile (ENTER)
```

and OS-9 shows you its contents:

```
This is the first file that
we created,
```

Remember, if you want to duplicate a file, use the Copy command.

You can also delete files by using the Del command. To delete the file /D0/samplefile, type:

```
del /D0/samplefile (ENTER)
```

and OS-9 deletes "samplefile" from the /D0 directory.

25

6 / In Case Of Trouble . . .

OS-9 tells you if an error has occurred. Most errors are the result of making a mistake while entering the command, such as misspelling a command or omitting a parameter.

When an error occurs, OS-9 displays error messages similar to this:

    ERROR #216

The number represents a specific type of error. This type of message, however, does not explain the problem. To print a descriptive message when an error occurs you can use the OS-9 Printerr command. To activate this command, type:

    printerr (ENTER)

Now OS-9 displays errors like this:

    ERROR #216
    - PATHNAME NOT FOUND

The OS-9 error messages fall into three catagories: operator, hardware or software.

An **operator error** tells you that you are asking the computer to do something it can't do.

Perhaps the most common error is typing a command incorrectly. For example, suppose you type this Dir command:

    dirr (ENTER)

OS-9 gives you this error message:

    ERROR #216
    - PATHNAME NOT FOUND

You simply need to type the command again, spelling it correctly:

```
dir (ENTER)
```

Another common operator error is trying to store too much information on a disk. For example, if you are in the middle of storing information and run out of disk space, OS-9 warns:

```
ERROR #248
- MEDIA FULL
```

Either use another disk or delete some files and/or directories, thereby, freeing disk space.

A **hardware error** warns you of a hardware problem, usually a flawed disk or a faulty disk drive. For example, if you tell OS-9 to display the directory (Dir command), and the disk was formatted with TRSDOS, the Color Computer Disk System's Operating System, OS-9 displays the message:

```
ERROR #241
- SECTOR ERROR
```

Repeat the command, using an OS-9 formatted disk. Remember, OS-9 can't read Color Computer Disk System's disks. OS-9 can read only those disks formatted by OS-9.

Other hardware errors may occur such as write or read errors. Always try another diskette first. If the problems continue, contact your Radio Shack Service Center.

A **software error** warns you of a problem in your application program. For example, suppose your application program tries to open a file in a directory where a file by that name already exists. OS-9 displays the message:

```
ERROR #218
- FILES ALREADY EXISTS
```

Automatic Printerr Routine

You will probably find it more helpful to have OS-9 print a descriptive error message by using the Printerr command. Normally the Printerr command is only active until you reset/reboot the system.

To make it automatic on your OS-9 system, you can add the Printerr command to the "startup" file. Follow these steps:

1. Display the "startup" file, by typing:

    ```
    list /D0/startup (ENTER)
    ```

    Write down the contents. If you have a printer, you can redirect the listing to it by typing:

    ```
    list /D0/startup >/P (ENTER)
    ```

2. Rename "startup" to a temporary file:

    ```
    rename /D0/startup startup.temp (ENTER)
    ```

3. Create a new startup file:

    ```
    build startup (ENTER)
    ?
    ```

    OS-9 is now ready to build a new "startup" file. First, enter the lines that you copied from the original file. Then add these lines:

    ```
    echo (ENTER)
    printerr (ENTER)
    echo Print Error Routine Now Active
    (ENTER)
    (ENTER)
    ```

    These lines activate the Printerr routine and print a message that the Print Error Routine is now active. The (ENTER) on the line by itself tells OS-9 to end the file.

4. Press the reset button to load the new "startup" file.

Now, Printerr is a permanent part of your startup file.

29

7 / AND THERE'S MORE . . .

You've just begun to scratch the surface of OS-9. This chapter discusses some of the other ways you can use OS-9, through more commands, concepts, and programs that are becoming available.

More Commands

The following summarizes some useful OS-9 commands:

copy — single drive (-s option)

Copies a file from one disk to another using only one drive. For example:

```
copy /D0/BUSINESS/file1 /D0/newfile -s
     (ENTER)
```

Copy alternately prompts you to ready the destination or source disk and press ⓒ to continue.

date

Displays the current system date. If you specify the **t** option, the OS-9 also displays the time. For example:

```
date (ENTER)
```

displays the current system date,

```
date t (ENTER)
```

displays the current system date and time.

free

Tells you the amount of free space remaining on a disk. OS-9 displays the name of the disk and the date it was created. Next, it displays the total capacity of the disk (in sectors — a sector is equal to 256 bytes or characters), the number of unused sectors, and the largest block available. (A block is an area of contiguous sectors.) Type:

    free (ENTER)

and OS-9 displays the amount of free space on the disk in Drive 0 by default. To see how much memory is free on the disk in Drive 1, type:

    free /D1 (ENTER)

mfree

Displays the amount of available memory in your OS-9 system. The screen displays the amount as "pages" of memory. A page is equal to 256 bytes. The address is shown for each contiguous set of pages as well as a total for the pages available. To see how much memory is available in your system, type:

    mfree (ENTER)

setime

Lets you set the system time and date. Type:

    setime (ENTER)

and OS-9 prompts:

                    YY/MM/DD HH:MM:SS
        TIME ?

To set the date as March 19, 1983, at 4:15 PM, enter:

83/03/19 16:15 (ENTER)

The seconds are optional.

For more information on OS-9 commands, see *OS-9 Commands*.

Command Modifiers

OS-9 offers a way for you to customize commands for your needs. You can add modifiers to almost any command line. The following briefly explains these modifiers.

Alternate Memory Size. You can specify the amount of memory to be set aside for the command by using the number sign symbol (#). Very often this speeds up execution of a program.

I/O Redirection. These modifiers let you reroute a program's standard I/O paths to alternate files or devices. For example, a program that normally displays to the screen can easily output to the printer instead without changing the program and vice versa. The I/O redirection modifier symbols are <, >, and >>.

Concurrent Execution. You can run two or more programs at the same time. The ampersand (&) tells OS-9 to start running that program and display another prompt. At that time you can start another program to run at the same time as the first.

Pipes. Pipes transfer data between programs. The output of program1 becomes the input for program2. The exclamation point (!) symbolizes a pipe.

For more information on OS-9 assembly language and system calls see the manual, *OS-9 Technical Information*. For information on OS-9 high-level languages, check with your Radio Shack dealer.

OS-9 Languages

OS-9 lets you use a number of different languages that were previously not available for the Color Computer.

The **OS-9 Assembler** comes with your OS-9 system. It lets you write assembly language programs in 6809 code for your OS-9 system.

High Level Languages can also be used with OS-9. New languages are constantly being developed, such as BASIC and Pascal.

OS-9 System Calls can be used for easy communication between your assembly language program and OS-9.

For more information on OS-9 assembly language and system calls see the manual, *OS-9 Technical Information*. For information on OS-9 high-level languages, check with your Radio Shack dealer.

Addendum to "Getting Started with OS-9" Manual
Catalog No. 26-3030

Please note the following change in the manual:

Page 6, step 6 -- The acceptable test speeds for the disk drive test should be from 295.5 to 304.5 rpm.

OS-9 Commands

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B.   RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C.   No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D.   Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV.   RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A.   Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B.   Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C.   CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D.   CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E.   CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F.   CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G.   All copyright notices shall be retained on all copies of the Software.

V.   APPLICABILITY OF WARRANTY

A.   The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B.   The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI.   STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

OS-9 Commands

Introduction

This Manual is designed to acquaint you, with your OS-9 Operating System. OS-9 greatly expands the capabilities of your TRS-80 Color Computer.

OS-9 is based on the UNIX operating system, often acclaimed as the operating system of the future because of its versatility, unique structure, usefulness, and user-friendliness. UNIX is widely used on large computers, and now systems like OS-9 bring UNIX's clear advantages to owners of smaller computers.

OS-9 was developed by the same people who designed the 6809 microprocessor chip, the "heart" of the TRS-80 Color Computer. Unlike many other microprocessors, the 6809 is fully capable of running state-of-the-art software like OS-9. That kind of close compatibility is a real plus for the user.

The OS-9 Operating System has many advanced features, which you'll learn about in this manual. By the time you've finished reading and experimenting on your computer, you'll be familiar with, and ready to take advantage of OS-9 features like:

- Friendly user interface and environment

- Multi-user/multi-tasking realtime operating capabilities

- Extensive support for structural, modular programming

- Device-independent interrupt-driven input/output system

- Multi-level, fast random-access directory and file system

- Readily expandable and adaptable design

We suggest that you read the manual chapter by chapter, experimenting with OS-9's features and commands as you go. But if you're a computer veteran, and you feel you're ready to begin using OS-9 without any further information about it, you can jump ahead to Chapter 6, "OS-9 COMMANDS".

Whether you're a novice or a veteran you've made a wise choice with OS-9, and it won't be long before you discover for yourself its many advantages.

Contents

1/Introduction to the Shell

The "shell" is the part of OS-9 that accepts commands from the keyboard. It's designed to provide a convenient, flexible, and easy-to-use interface between you and the powerful functions of the system.

You automatically enter the shell each time you start up OS-9. When you see the "OS-9:" prompt, that means the shell is active and waiting for input — for your commands through the keyboard.

> **Note:** It doesn't matter whether you use upper-case or lower-case letters — or a combination of both — in your commands; OS-9 recognizes and handles both.

1.1 Command Structure

Commands — which are really programs for the computer to run — can include one or more words, but they always begin with the name of a program. The program can be one of a number of things, for instance:

- The name of a machine language program on disk. (Applies to experienced users, too, you'll almost always use commands from the OS-9 disk. OS-9's commands are listed and explained in Chapter 6.)

- The name of a machine language program already in memory.

- The name of an executable program compiled by a high-level language like BASIC09, Pascal, or C (See Section 3.8.)

- The name of a procedure file. (See Section 3.6.)

When OS-9 receives a command, the shell searches for the appropriate program in this sequence:

1. Memory.

2. The "execution directory", which contains OS-9's

1

command programs. (OS-9's execution directory is usually CMDS.)

3. The user's "data directory", in which the user can store program files as well as text files. When OS-9 processes a file from the data directory, it runs it as a procedure file, assuming that the file contains several commands, or procedures. OS-9 will run them sequentially, just as if the commands had been manually typed in one by one.

As soon as OS-9 locates the program, it runs it.

Command Parameters

The program specified in the command can be followed by one or more "parameters", variables which give the computer more specific instructions to follow. OS-9 automatically passes the parameters to the program called up by the shell when you enter your full command line. For example, in the command line:

```
list file1 (ENTER)
```

List is the name of a program that displays the contents of a text file, and file1 — the specified parameter — is the name of the file whose contents are to be displayed.

> **Note:** Parameters are always separated from the command line, and from each other, by spaces; therefore parameter names themselves cannot contain spaces. Chapter 6 discusses parameters for each of OS-9's commands.

Some commands have more than one parameter. For instance, the Copy command makes an exact copy of a file. It requires two parameters: the name of the file to be copied, and the name of the file that will be the copy. So if you want to copy a file called startup, and call the copy newstartup, your command line reads:

```
copy startup newstartup (ENTER)
```

Other parameters let you select built-in command options. For instance, the Dir command by itself simply shows the

name of all files in the user's current data directory — the directory in which the user is positioned when giving the command.

But if you add the e (for "entire") option as a parameter, like this:

    dir e (ENTER)

then the output includes not only the names of the files, but also complete statistics about each file — the date and time created, size, security codes, and so forth.

The Dir command can also accept as a parameter the name of a particular directory on the system. For example, the command line:

    dir SYS (ENTER)

produces a list of all files in the SYS directory. And the command line:

    dir SYS e (ENTER)

gives complete information about each file in SYS.

    dir SYS e >/P (ENTER)

gives complete information about each file in SYS and re-directs it to the printer.

> **Note:** A command line can also include one or more "modifiers" — specifications used by the shell to alter the program's standard input/output files or memory assignments. (See Chapter 3.)

1.2 Common Command Formats

This section includes examples of command formats most commonly used with OS-9, and examples of how each command might look as it's entered into the computer. Parameters in brackets are optional; others are necessary parts of the command.

FORMAT: chd *DIRECTORY NAME*

EXAMPLE: chd /DØ/SYS (ENTER)

Moves the user from the current working directory into the
directory specified as a parameter, in this case /D0/SYS.

FORMAT: makdir *DIRECTORY NAME*

EXAMPLE: makedir /DØ/EMPLOYEES (ENTER)

Creates a new directory, in this case called /D0/EM-
PLOYEES. You'll often want to follow this command with a
Chd command to make the new directory your current work-
ing directory.

FORMAT: pwd (ENTER)

EXAMPLE: /DØ/BUSINESS/PAYROLL

Shows the full path from the directory PAYROLL, to the
current working directory.

FORMAT: dir /D0/[*filename*] [e] [x]

EXAMPLE: dir PAYROLL e (ENTER)

Lists the names of all files contained in the current working
directory if you don't specify another. In this case, the
PAYROLL directory is specified. The e option gives com-
plete statistics about each file in the directory. The x option
lists files in an execution directory rather than a data
directory.

> **Note:** When you're using a command that affects direc-
> tories — and Chd, Makdir and Dir are good examples
> — make sure you specify the name of a directory and
> not a single file; otherwise, the command won't work.
> Remember: a file contains lines of text or a single pro-
> gram, and a directory is a collection of files and sub-
> ordinate directories.

FORMAT: copy *filename1 filename 2*

EXAMPLE: copy memos newmemos (ENTER)

4

Creates a new file — in this case newmemos — and then copies all data from memos into it. The original file isn't affected.

FORMAT: del *filename*

EXAMPLE: del letters (ENTER)

Deletes — destroys — the specified file.

FORMAT: free *DEVICENAME*

EXAMPLE: free /D1 (ENTER)

Shows how much free space remains on the specified device, in this case the disk on drive 1.

FORMAT: list *filename*

EXAMPLE: list junk (ENTER)

Displays on the terminal the contents — the text — of the specified file.

FORMAT: rename filename1 filename2

EXAMPLE: rename stuff miscellany (ENTER)

Changes the name of a file. In this case, the file formerly called stuff is now named miscellany.

1.3 Using the Video Display and Keyboard

OS-9 has many features which expand the capability of the Color computer's video display and keyboard. With OS-9, for instance:

- The video display has upper/lower case, screen pause, and graphics functions.

- The keyboard can generate all ASCII characters. (Appendix C lists all characters and codes you can generate from the keyboard.)

- The keyboard has a type-ahead feature that lets you enter data before it's requested by a program. But, only if the disk drives are not being accessed by a program.

- The video display and keyboard together can be dealt with as a file. OS-9 refers to them as a file called /term.

Video Display Functions

The Color Computer normally uses only upper-case letters. If you want lower-case letters (for instance, if you'd like to be able to send them to the printer), you can turn off the upper-case function with a command called tmode -upc. You then see lower-case letters represented on the screen in reverse video (try Dir as an example) — green letters on a black background instead of the usual black letters on a green background. To return to all upper-case letters, use the command tmode upc.

> Note: See the **Keyboard Shift and Control Functions** section for important information about shift lock behavior.

The display's screen pause feature stops programs after they display 16 lines. Press any key to continue program output. You can turn screen pause off and on by using, respectively, the tmode -pause and tmode pause commands.

The display system also has a complete set of commands to emulate commercial data terminals, plus a complete set of graphics commands. (They're described in detail in Appendix B.)

Keyboard Shift and Control Functions

With OS-9, several Color Computer keys have new and useful functions.

The (SHIFT) key works something like its counterpart on a typewriter, letting you select upper-case or lower-case characters. The shift lock function, which affects only letter characters, is normally on, producing upper-case letters. To obtain a

lower-case letter when the shift lock is on, use the (SHIFT) key.

To turn off the shift lock function, press (CLEAR) and (O) simultaneously. Then your keyboard generates lower-case characters, and, if you want upper-case, you use the (CLEAR) key. Again pressing (CLEAR) and (O) simultaneously turns the shift lock back on.

> **Note:** The tmode upc function affects shift lock behavior. When you're in the upc mode, upc overrides both the (SHIFT) key and the shift lock function, and you get only upper-case characters. In order for the (SHIFT) key and lock function to work, you should be in -upc.

Several key combinations, when pressed simultaneously, generate "control functions" from the keyboard. The (CLEAR) (O) combination, which reverses shift lock state, is a good example. Other control functions, and the key combinations that generate them, include:

> (CLEAR) (A) — Repeats previous input line. Displays, but doesn't process, the last line entered, and positions the cursor at the end of the line. Press (ENTER) to enter the line, or edit the line by backspacing. If you edit, you can press (CLEAR) (A) again to display the edited line.

> (CLEAR) (D) — Redisplays present input on the next line.

> (CLEAR) (W) — Temporarily halts output to the display so you can read the screen before the data scrolls off. Press any single key to resume output.

> (CLEAR) (E) — Stops the current running program. (The (BREAK) key performs the same function.)

> (SHIFT) (BREAK) — Interrupts the video display of a running program, reactivating the shell while the program runs as a background task. This function is often referred to as CONTROL C.

(CLEAR) (BREAK) — Sends an end-of-file message to programs that read input from the terminal instead of from a disk. Often referred to as the ESCAPE function, this key combination must be the first character on a line in order to be recognized.

(SHIFT) (←) — Deletes the entire current line. This function is often referred to as CONTROL X. The (←) alone, which backspaces and erases single characters, is commonly known as CONTROL H.

NOTE: The (CLEAR) key is used as a control key by OS-9. Thus, if you wanted to send a CONTROL Q function, you would press (CLEAR) (Q) simultaneously.

1.4 Sending Output to the Printer

Most commands and programs normally send their output to the Color Computer's video display. But if you want output to be printed, add this at the end of a command line:

    > / P

The ">" character tells the shell to redirect output to the printer through the Color Computer's RS-232 serial I/O port, which has the device name "/P".

If for example, you want the output from the Dir command to go to the printer, type:

    dir > / P (ENTER)

Technical Information for the RS-232 Port

The RS-232 port can be operated at all standard baud rates, from 110 baud to 2400 baud. (The default speed is 600 baud.) The character format used is 1 start bit, 8 data bits (no parity), 1 stop bit.

You can use the Xmode command to set the port's baud rate, end-of-line delay, auto line feed, and so forth. To examine the printer's current settings, type:

8

```
xmode /P (ENTER)
```

Then, if you want to make changes, use the Xmode command and information from this chart:

Baud Rate	Code
110	0
300	1
600	2
1200	3
2400	4

If you want, for instance, to set the port to 1200 baud, and the end-of-line delay (null count) to 4 character times, type:

```
xmode /P baud=3 null=4 (ENTER)
```

2/The OS-9 File System

This chapter gives you information about some of the most important elements of the OS-9 system. It acquaints you with the ways in which OS-9 deals with input and output, and with the structure and characteristics of the entire OS-9 file system.

2.1 The Unified Input/Output System

OS-9 has a unified input/output system in which data transfers to all I/O devices are performed in almost exactly the same way, regardless of the particular hardware devices involved.

It might seem that the different operational characteristics of the I/O devices would make this difficult: after all line printers and disk drives behave very differently. However, OS-9 overcomes most of these differences by defining a set of standardized logical functions for all devices and by making the devices conform to these conventions, using software routines to eliminate hardware dependencies wherever possible. The result: a much simpler and more versatile input/output system.

OS-9's unified I/O system is based on logical entities called "I/O paths". Paths are, in effect, "software channels" that can be routed from a program to a disk to any other I/O device, or even to another program.

Data transferred through paths is processed by OS-9 to conform to the hardware requirements of the specific I/O device involved. Data transfers can be either bidirectional (read/write) or unidirectional (read only or write only), depending on the device and/or how you establish the path.

Data transferred through a path is a stream of 8-bit binary bytes that have no specific type or value: what the data actually represents depends on how it's used by each program. This is important because it means that OS-9 doesn't require data to have any special format or meaning.

Some of the advantages of OS-9's unified I/O system are:

- Programs will operate correctly regardless of the particular I/O devices selected and used when the program is actually executed.

- Programs are highly portable from one computer to another, even when the computers have different kinds of I/O devices.

- I/O can be redirected to alternate files or devices when the program is run, without having to alter the program.

- New or special device-driver routines can easily be created and installed by the user.

2.2 Organization of the File System

Disks are multifile devices that store data — both text and programs — in separate logical entities called files. OS-9 handles files in a number of ways designed to help you organize information easily and well.

For instance, with OS-9, groups of files can be collected into directories, much as in an office file folders pertaining to a particular subject can be gathered into a file cabinet drawer. Directories can in turn be collected into larger directories, just as several file cabinet drawers are gathered into a single file cabinet, and so forth. With OS-9, you can have a virtually limitless number of directory levels, with each directory containing other directories and/or files.

When you're working with OS-9's files and directories — both the ones that come built in to the system and the ones you create — it's important to remember this multi-level "hierarchical" organization.

In effect, it lets you build an upside-down tree, branching out as you go down. That way, each user on your system can privately organize material without affecting anyone else's material. You can organize your own material in particularly

useful ways. And both the system itself and its users can easily locate stored material.

An OS-9 System Disk contains names of, and linkages to, all system I/O entities in the hierarchy. For example, a typical System Disk, diagrammed partially and simply, might look like this:

System Device Directory

```
P              DO           TERM              D1                    T1
               |                              |
          DO Root Directory            D1 Root Directory
  |            |            |      |            |                    |
DEFS        startup      CMDS   PAYROLL      MEMOS
  |                                 |       |            |
OS-9Defs                                  SALES      ACCOUNTING
                |     |     |    |          |
             copy  list  dir  del         chris
                          |
                  HOURLY    SALARIED
                   |    |       |
                 a _ m n  / z  a  m   n    /
```
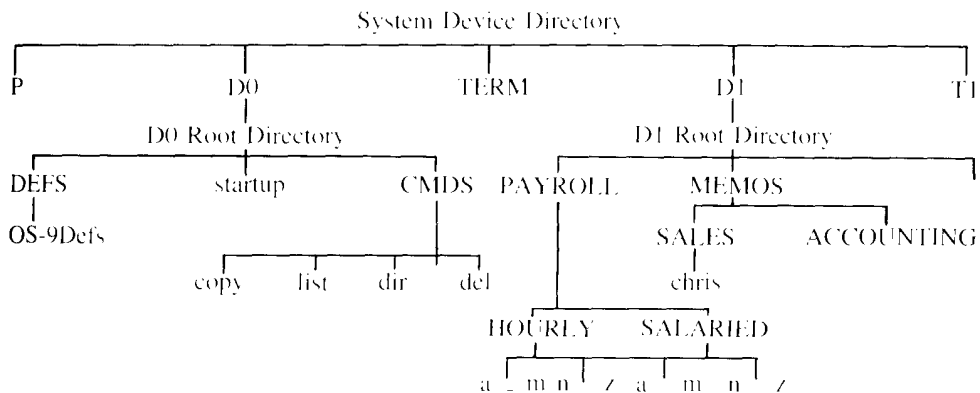
> **Note:** It's customary to capitalize directory names and to use lower-case for file names; that way, you can tell at a glance what's a directory and what's a file.

In the diagram, P is the printer; DO is the first disk drive; TERM is the keyboard and video display; D1 is a second disk drive; and T1 is the Color Computer's RS-232 serial port.

The "root directory" of DO — the "root" from which the rest of the disk's file system "grows" — contains a file called startup and two other directories, DEFS and CMDS.

Those directories, in turn, contain files — DEFS contains a file called OS-9Defs and CMDS contains four files: copy, list, dir, and del. All these files and directories — and many more — come built in to the OS-9 system.

The system diagrammed here also has a second disk, with its own root directory. (Root directories are automatically created when you initialize a disk using the format command.) Here the user has created directories called PAYROLL and MEMOS. PAYROLL, in turn, contains two other directories,

13

each of which contain two files. The MEMOS directory con-
tains two directories, one of which contains a file.

Names

Each file, directory, and physical I/O device on the system —
whether it's built in or added by you — has its own name.
When you're doing the naming, there are several things to
keep in mind.

Names can include from 1 to 29 characters, all of which are
used for matching. Each name must begin with an upper- or
lower-case letter, followed by any combination of the follow-
ing characters:

- Upper-case letters (A-Z)
- Lower-case letters (a-z)
- Decimal digits (0-9)
- Underscoring (_)
- Period (.)

Some legal names, therefore, are:

    raw.data.2
    REPORTS
    X.x
    project _ review.backup
    RJJONES
    M1O1968

File and directory names like the following ones are not legal:

19OCTOBER	(because it doesn't begin with a letter)
max*min	(because * isn't a legal character for names)
.DATA	(because it doesn't begin with a letter)
open orders	(because a name can't contain a space)

Pathlists

When you want to access anything on the system — to open
a path — you have to give OS-9 a description of the routing
of the path. You provide the information in the form of a
''pathlist'', a list of names from the root directory down to
the file you want to access.

If, for instance — again using the diagram — you want to access the chris file on the disk in Drive 1, you construct a pathlist by reading from the root directory to the file, listing the "stops along the way", and separating them with slashes, like this:

/D1/MEMOS/SALES/chris

OS-9 uses the pathlist sequentially, from left to right, to determine that the file you want is on Device D1, in the MEMOS directory, in the SALES (sub)directory; and that its name is chris.

OS-9's hierarchical organization and pathlist convention help you access what you need quickly and efficiently — and also mean that you can, if you want, have two files of the same name, as long as they're in separate directories; that way, of course, their pathlists are different.

> **Note:** Under some circumstances, you can take a "shortcut" to a file, directory, or device name. See **Working Directories** in Section 2.3.

Device Names

Each physical I/O device supported by the system has a unique name, defined when the system is set up and unchangeable while the system is running. Most of the device names used for the Color Computer are on the diagram. But in addition to P, D0, TERM, D1, and T1, the Color Computer also uses PIPE.

Device names can be used only as the first part of a pathlist, since they're at the root of the file system. In any pathlist, always precede the name of a device with a slash.

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

2.3 Directories

On OS-9, directories, which are collections of files, are in
reality themselves files -- "superfiles", but still files -- and
are processed by the same I/O functions used with regular
files.

Using Directories

To understand how directories work, assume that the disk in
drive one (" D1") is freshly formatted so that it has only a
root directory. You can use the Build command to create a
test file you call file1 on /D1. Build prints out "?" as a
prompt to indicate that it's waiting for you to enter a text
line. It places each line into the text file until you enter an
empty line with only a carriage return, like this:

```
build /D1/file1 (ENTER)
? This is the first file that (ENTER)
? we're creating. (ENTER)
? (ENTER)
```

If you use the Dir command, which lists the files in a direc-
tory, it now indicates the existence of the new file:

```
dir /D1 (ENTER)
  Directory of /D1    15:45:29
file1
```

You can use the **list** command to display the text stored in the
file:

```
list /D1/file1 (ENTER)

This is the first file
that we're creating.
```

Suppose you again use the **build** command to create two more text files:

```
build /D1/file2 (ENTER)
? This is the second file (ENTER)
? that we're creating. (ENTER)
? (ENTER)

build /D1/file3 (ENTER)
? This is another file. (ENTER)
? (ENTER)
```

Now if you use the Dir command, it shows three file names:

```
dir /D1 (ENTER)
 Directory of / D1    15:52:29
 file1     file2      file3
```

Creating Directories

To create a directory on the system, use the Makdir command. With Makdir, you can create a virtually unlimited number of directories on a disk.

Suppose that for now you want to create in the /D1 root directory a new directory called NEWDIR. You type:

```
makdir /D1/NEWDIR (ENTER)
```

and the new directory is automatically part of the D1 root directory. You can check it using Dir:

```
dir /D1 (ENTER)
 Directory of /D1        16:04:31
 file1     file2      file3
 NEWDIR
```

Now suppose you want to create a new file — a copy of file1. You want to call file1.copy, for instance — in the new directory. Use the Copy command, like this:

17

```
COPY /D1/file1/D1/NEWDIR/file1,
COPY (ENTER)
```

Note, that the second pathlist now has three names: the name of the root directory (/D1), the name of the next lower directory in the hierarchy (NEWDIR), and then the actual filename (file1.copy). Here's what the structure looks like now:

```
D1 Root Directory
     ┌──────────────┬──────────────┬──────────────┐
  NEWDIR          file1          file2          file3
     │
 file1.copy
```
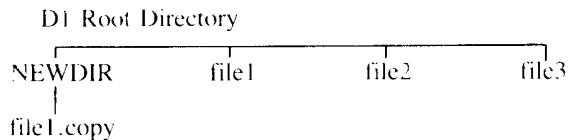
You can use Dir command to see the name of the file in the new directory:

```
dir /D1/NEWDIR (ENTER)

Directory of /D1/NEWDIR 15:29:29
file1.cop,
```

It's possible to use Makdir to create additional new directories in NEWDIR, and so on. Your only limit is disk space availability.

> **Note:** A file and the directory it's in must reside on the same device. That is, all the elements in a single pathlist must be on the same disk.

Deleting Directories

To delete a directory, it's first necessary to remove all the files it contains. If you delete a directory while it still contains files, OS-9 has no way — no path to access those files, or to return their storage to the storage pool.

Deleting a directory involves three steps:

1. Deleting all files in the directory with the **del** command.

2. Using the Attr command to reverse the directory attribute, turning the directory into a regular file. (See Section 2.4, **The File Security System.**)

3. Deleting the former directory — now a regular file — using the Del command.

It's not a difficult process, but there's an easier way: use the Deldir command to perform all three steps automatically.

Working Directories

When you're using OS-9, you're always associated with two directories: a "data directory" and an "execution directory." While you're using them, they're known as your "working" or "current" directories. The double association allows program files — executable files — to be organized separately from data files.

Immediately after startup, OS-9 sets the data directory to be the root directory of the system disk drive (usually /D0) and the execution directory to be the built-in CMDS directory on the same drive.

> **Note:** On timesharing systems, the Login command selects the user's initial data and execution directories according to specifications in the system password file.

While you're on the system, OS-9 automatically selects one or the other of your two working directories, depending on the usage of the pathlist:

- It searches the execution directory when it attempts to run or to load into memory files assumed to be executable programs. (This means that programs to be run as commands or loaded into memory must be in the execution directory.)

- It uses the data directory for all other references, such as text files.

Using Working Directories. Knowing about working directories — current directories — can let you take "shortcuts" as you write out pathlists, and lets OS-9 find what you want more quickly:

- If the command you're using relates to a file or a device **not in** your working directory, or in your working directory but above you in the hierarchy,

it's necessary to use a complete pathlist, starting at the root, for instance with /D0 or /D1.

● But if you're trying to access a file or device **within** your working directory, below you on the hierarchy, you can start your pathlist immediately below your working directory; the rest of the pathlist is implied.

For example, if your current data directory begins with /D0, and you want to reach a file called baseball in the directory whose root is /D1, you use its full pathlist, which might be:

/D1/PETE/GAMES/baseball

in your command line.

But if your current data directory is /D1/PETE/GAMES and you want to access baseball, you simply include in your command the filename:

baseball

The full pathlist is implied, and processing begins with your current data directory.

Pathlists using working directories can also specify additional lower-level directories and files. For instance, if your current directory is still /D1/PETE/GAMES, you can type:

ACTION/racing

in your command line, and this pathlist is implied:

/D1/PETE/GAMES/ACTION/racing

If your current execution directory is /D0/CMDS, and your current data directory is /D0/JONES, and you type

copy file1 file2 (ENTER)

as your command line, in its search OS-9 expands both implied pathlists to their fullest. Written out completely, they look like this:

/D0/CMDS/copy /D0/JONES/file1 /D0/JONES/file2

Changing Working Directories. You can make any directory for which you have access permission your working directory. The built-in shell commands Chd and Chx independently change the current data directory and the current execution directory, respectively.

Follow the Chd and Chx commands with a pathlist that describes the new directory to which you want to go. If your pathlist begins with a device name, OS-9 will begin its search at the device directory level — up at the root. Otherwise, its search will start at the current directory.

For instance, suppose your current data directory is /D0/JOHN and you want to make /D1/MY.DATAFILES your current data directory. Type:

```
chd /D1/MY.DATAFILES (ENTER)
```

But if you're already in /D1/MY.DATAFILES, and you want to move to a subdirectory called FEB.FILES, then you simply type:

```
chd FEB.FILES (ENTER)
```

and the full pathlist — /D1/MY.DATAFILES/FEB.FILES — is implied. Use the Chx command in exactly the same way you use Chd.

Anonymous Directory Names. Sometimes it's useful to be able to refer to your current directory, or to a higher-level directory, but you may not know the full pathlist to use. Or you may want merely to save typing time.

In either case, OS-9 makes special "name substitutes" available:

- The name "." refers to the current directory

- The name ".." refers to the "parent" of the current directory (the next highest-level directory in the path)

- The name "..." refers to the directory two levels up, and so on

You can use the names in place of pathlists and/or as the first names in pathlists. Some examples:

21

```
dir , (ENTER)
```

lists filenames in the current data directory.

```
dir ,, (ENTER)
```

lists names in the current data directory's parent directory.

```
del ,,/temP (ENTER)
```

deletes the file called temp from the current data directory's parent directory.

The substitute names refer to either the execution or data directories, depending on the context in which the names are used. For example, if "`..`" is used in a pathlist of a file which is executable, it represents the parent directory of the execution directory. Similarly, if "`.`" is used in a pathlist describing a program's input file, it represents the current data directory.

2.4 The File Security System

Every file and directory has properties called "ownership" and "attributes", which determine who may access the file and how it may be used.

OS-9 automatically stores with each file the user number associated with the process that created it. This user is considered to be the owner of the file.

Usage of security functions are based on attributes that define how and by whom the file can be accessed. There are a total of eight attributes, each of which can be turned "off" or "on" independently. When the "d" attribute is on, it indicates that the file is a directory. The "s" attribute means that the file is sharable — that more than one program can read from the file at the same time.

The other six attributes control whether the file can be read, written to, or executed, by either the owner or the "public" (all other users). Specifically, these six attributes are:

Write permission for owner: If it's on, the owner may write to the file or delete it. This permission can be used to protect important files from accidental deletion or modification.

Read permission for owner: If it's on, the owner is allowed to read from the file. This can be used to prevent "binary" files from being used as "text" files.

Execute permission for owner: If it's on, the owner can load the file into memory and execute it. The file must contain one or more valid OS-9-format memory modules in order to be loadable.

Write permission for public: If it's on, any other user may write to or delete the file.

Read permission for public: If it's on, any other user may read (and possibly copy) the file.

Execute permission for public: If it's on, any other user may execute the file.

For example, if a particular file has all permissions on except "write permit to public" and "read permit to public", the owner has unrestricted access to the file, and other users can execute it, but not 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 from the Dir **e** command used on the current data directory is:

```
Directory of . 10:20:44

Owner  Last Modified  Attributes  Sector  Bytecount    Name
-----  -------------  ----------  ------  ---------  --------
    1  83/05/29 1402  --e--e-r       47         42   file1
    0  83/10/12 0215  ---wr-wr       48         43   file2
    3  83/04/29 2335  -s----wr       51         22   file3
    1  83/01/06 1619  d--wr-wr       6D        800   NEWDIR
```

The "Attributes" column shows which attributes are currently on by the presence or absence of particular characters in this format:

```
dsewrewr
```

The character positions correspond, from left to right, to: directory, sharable, public execute, public write, public read, owner execute, owner write, owner read

Use the Attr command to examine or change the attributes of a particular file. Typing Attr followed by a filename shows you a file's current attributes, for example:

```
attr file2 (ENTER)
-s-wrewr
```

Reading the attributes from left to right, you can see that file2: is not a directory; is sharable; can't be executed by the public but can be written to and read by the public; and can be executed, written to, and read by its owner.

If you use Attr and a filename followed by a list of one or more attribute abbreviations, the file's attributes will be changed accordingly (if, of course, it's legal for you to make the changes). For instance the command:

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

enables public write and public read permissions and removes execute permission for both the owner and the public.

> **Note:** The d attribute behaves somewhat differently from the other attributes, in order to protect data stored in directories. You can't use Attr to turn on the d attribute — only Makdir can do that — and you can use Attr to turn d off only if the directory is empty.

2.5 Reading and Writing from Files

OS-9 uses single type and format for all files, each of which stores an ordered sequence of 8-bit bytes. OS-9 isn't usually sensitive to the contents of files for most functions. A given file can store a machine language program, characters of text, or almost anything else. Data is written to and read from files exactly as it's given. The file can be any size, from zero up

to the maximum capacity of the storage device, and can be expanded or shortened as desired.

When a file is created or opened, a "file pointer" is established for it. Bytes within the file are addressed like memory, and the file pointer holds the "address" of the next byte in the file to be written to or read from. The OS-9 "read" and "write" service functions always update the pointer as data transfers are performed, so that successive read or write operations perform sequential data transfers.

If you're an advanced user of the system, and particularly if you're using high-level languages, OS-9 is even more versatile than usual. There are certain functions that allow you to reposition the file pointer.

To expand a file, you can simply 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.

File Usage in OS-9

Even though there is physically only one type of file, the logical usage of files in OS-9 covers a broad spectrum. Because all OS-9 files have the same physical type, you can use commands such as Copy, Del, and so forth, with any file, regardless of its logical usage. Similarly, a particular file can be treated as having different logical usages at different times by different programs. The main usages of files discussed here are:

- Text
- Random access data
- Executable program modules
- Directories
- Miscellaneous

Text Files. These files contain variable-length sequences — lines -- or ASCII characters. Each line is terminated by a carriage return character (ASCII code 0D or decimal 13).

Text files are used for program source code, procedure files, messages, documentation, and many other purposes. The Text Editor operates on this file format.

Text files are usually read sequentially, and are supported by almost all high-level languages (such as BASIC09 read and write statements). Even though it's possible to randomly access data at any location within a text file, it's rarely done in practice because lines vary in length and it's hard to locate the beginning of each line without actually reading the data to find carriage return characters.

You can examine the content of text files by using the List command.

Random-Access Data Files. Random-access data files are created and used primarily from within high-level languages like BASIC09, Pascal, C, and Cobol. In BASIC09 and Pascal, get, put, and seek functions operate on random-access files.

Each file is organized as an ordered sequence of records. Each record is exactly the same length, so if a record's numerical index is given, the record's beginning address within the file can be computed by multiplying the record number by the number of bytes used for each record. Records can, therefore be directly accessed in any order.

In most cases, the high-level language allows each record to be subdivided into fields. Each field generally has a fixed length and usage for all records within the file. For example, the first field of a record may be defined as being 25 text characters, the next field may be two bytes long and used to hold 16-bit binary numbers, and so on.

It's important to understand that OS-9 itself doesn't directly process or deal with records other than by providing the basic file functions required by all high-level languages to create and use random-access files.

Executable Program Module Files. These files are used to hold program modules generated by the assembler or compiled by high-level languages. Each file can contain one or more program modules.

OS-9 program modules resident in memory have a standard module format that, besides the object code, includes a "module header" and a CRC check value. Program modules

stored in files contain exact binary copies of the programs as they'll exist in memory, and not one byte more. Unlike many other operating systems, OS-9 doesn't require a "load record" system because OS-9 programs are position-independent and therefore don't have to be loaded into specific memory addresses.

In order for OS-9 to load the program module(s) from a file, the file itself must have execute permission set, and each module must have a valid module header and Cylic Redundancy Checksum (CRC) value. If a program module is altered in any way, either as a file or in memory, its CRC check value is incorrect and OS-9 refuses to load the module. The verify command can check the correctness of the CRC values, and update them to corrected values if necessary.

If a file has two or more modules, they're treated as independent entities after loading and they reside at different memory regions.

If you attempt to use the List command on program files, or any other files that contain binary data, the result is a jumbled display or random characters and effects. Use the Dump command to safely examine the contents of this kind of file in hexadecimal and controlled ASCII format.

Directories. Directories — which are, in effect, "superfiles" — play a key role in the OS-9 file system. Section 2.3 of this chapter describes how they're used by various OS-9 features. Directories can be created only by the Makdir command, and can be identified by the d attribute.

Each directory is organized into 32-byte records. Each record can be an entry in the directory. The first 29 bytes of the record is a string of characters that is the file name. The last character of the name has its sign bit (most significant bit) set. If the record isn't in use, the first character position has, the value zero. The last three bytes of the record is a 24-bit binary number that's the logical sector number where the file header record is located.

The Makdir command initializes all records in a new directory to be unused entries except for the first two entries. These entries have the names "." and ".." along with the

logical sector numbers of the directory and its parent directory, respectively.

The commands Copy and List won't work with directories, Instead, use Dir. Directories also can't be deleted directly, but must first be emptied and turned into regular files.

Miscellaneous File Usage. OS-9's basic file functions are so versatile that it's possible to devise an almost unlimited number of special-purpose file formats for particular applications, formats which don't fit into any of the categories discussed here.

Examples of special file usage include COBOL Indexed Sequential (ISAM) files and some special word processor file formats which allow random access of text lines. As mentioned earlier most OS-9 utility commands work with any file format, including these special types. In general, the Dump command is the preferred method for examining the contents of unusually formatted files.

Physical File Organization

OS-9's file system implements universal logical organization for all I/O devices that effectively eliminates most hardware-related considerations for most applications. This section gives basic information about the physical file structure used by OS-9. (For more information, see the OS-9 *Technical Information* manual.)

Each OS-9 file comprises one or more "sectors", which are the physical storage units of disk systems. Each sector contains 256 data bytes. Disks are numbered sequentially starting with sector 0, track 0. This number is called a "logical sector number", or LSN. The mapping of logical sector numbers to physical track/sector numbers is done by the disk driver module.

A sector is the smallest allocatable physical unit on a disk system. However, to increase efficiency on some larger-capacity disk systems, OS-9 uses uniform-sized groups of sectors, called "clusters", as the smallest allocatable unit. Cluster sizes are always an integral power of two (2, 4, 8, and so on.) One sector of each disk is used as a bitmap

(usually LSN 1), in which each data bit corresponds to one cluster on the disk. The bits are set and cleared to indicate which clusters are in use, which are defective, and which are free for allocation to files.

The Color Computer disk system uses this format:

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

On OS-9, each file has a directory entry which includes the filename and the logical sector number of the file's "file descriptor sector", which contains a complete description of the file, including:

- attributes
- owner
- date and time created
- size
- segment list (description of data sector blocks)

Unless the file size is zero, the file will have one or more sectors/clusters used to store data. The data sectors are grouped into one or more contiguous blocks called "segments."

3/Advanced Features Of The Shell

Chapter 1 of this manual introduced basic shell functions and commands. This chapter discusses the more advanced capabilities of the shell. In addition to basic command line processing, the shell has functions that facilitate:

- Input/Output redirection, including filters

- Memory allocation

- Multitasking (concurrent execution)

- Procedure file execution

- Built-in commands

You can use these advanced capabilities in a virtually unlimited combination of ways. Of course, it's impossible to give more than a representative set of examples here — but you're encouraged to study the basic rules, use your imagination, and explore the possibilities on your own.

3.1 More About Command Line Processing

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

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

Only the verb (the program, procedure file, or command name) need be present; the other parts are optional. After the shell identifies the verb, it processes the modifiers. Any other text not yet processed is assumed to be parameters and is passed to the program being called.

If the verb is a built-in shell command (see Section 3.5), the shell simply executes it. If it's not built in, the shell searches

for the appropriate program, and, when it finds it, runs it as a new process.

Then the shell deactivates itself until the program being called terminates, at which time the shell takes the next input. The cycle continues until the shell detects an end-of-file in the input path. Then the shell terminates its own execution.

Here's a sample shell command line which calls the assembler:

```
asm sourcefile 1 -o >/P #12K  (ENTER)
```

In this example:

asm	is the verb
sourcefile 1 -o	are parameters passed to Asm
>/P	is a modifier which redirects the output (the listing) to the system's printer
#12K	is a modifier which requests that the process be assigned 12K bytes of memory instead of its (smaller) default amount

Note: The verb should always be the first entry in any line.

3.2 Execution Modifiers

Execution modifiers tailor OS-9 commands to your specifications. Type them in a command line after the verb, and either before or after any parameters you're using.

Execution modifiers are processed by the shell before the program is run. If the shell detects an error in any of the modifiers, the run is aborted and the error reported.

Characters which compose modifiers are stripped from the part(s) of the command line passed to the program as para-

meters. Therefore, the characters reserved for use as modifiers (# : ! < > &) can't be used inside parameters.

Alternate Memory Size Modifier

When the shell involves a command program it allocates the program the minimum amount of working RAM memory specified in the program's module header. (A module header is part of all executable programs and holds the program's name, size, memory requirements, and other information.)

Sometimes it's desirable to increase this default memory size. You can assign memory either in 256-byte pages by using the modifier #n where n is the decimal number of pages, or in 1024-byte increments by using the modifier #nK. The two examples below have identical results:

```
copy #8 file1 file2 (ENTER)
copy #2K file1 file2 (ENTER)
```

Each command line specifies that memory size is to be 2048 bytes. In the first command, 8*256 = 2048; in the second, 2*1024 = 2048.

I/O Redirection Modifiers

Input/Output redirection modifiers reroute a program's standard I/O paths to alternate files or devices.

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

Programs which 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: Passes data from the terminal's keyboard to the program.

- Standard output path: Outputs data from the program to the terminal's display.

- Standard error output path: Outputs routine status messages — prompts and errors, for instance — to the terminal's display. (The name "error output path" is somewhat misleading, since many kinds of messages besides errors travel the path.)

Correspondingly, OS-9 offers you three redirection modifiers:

- < redirects the standard input path

- > redirects the standard output path

- >> redirects the standard error output path

When you use a redirection modifier in a command line, follow it immediately with a pathlist describing the file or device to or from which the I/O is to be redirected.

For example, if you want to redirect the standard output of the List command to write the contents of a file called correspondence to the printer instead of to the terminal, type:

```
list correspondence >/P (ENTER)
```

Files referenced by I/O redirection modifiers are automatically opened, created, or closed (as appropriate) by the shell. In the next example, the output of the Dir command — a list of files in the directory MEMOS — is redirected to the file /D1/ savelisting:

```
DIR /D0/MEMOS >/D1/savelisting
(ENTER)
```

Then, if the list command is used on the file /D1/savelisting, the redirected output from the Dir command is displayed like this:

```
list /D1/savelisting (ENTER)
  Directory of /D0/MEMOS 10:15:00
Jackson    moeller    jones
```

Redirection modifiers can be used before and/or after the program's parameters, but each modifier can be used only once in a command. After the program specified in the command is run, the redirection modifier terminates with it; when

you run the program again, it will use its standard I/O paths unless you again specify otherwise.

> **Note:** When processes are created, they inherit their parent processes' standard I/O paths. Therefore, when the shell creates processes, they inherit its standard paths.

3.3 Command Separators

A single shell input line can request execution of more than one program. These programs can be executed sequentially or concurrently. "Sequential execution" means that one program must complete its function and terminate before the next program is allowed to begin execution. "Concurrent execution" means that several programs are allowed to begin execution and run simultaneously.

Sequential Execution

Programs entered on separate command lines are executed sequentially. But OS-9 lets you save time by specifying on a single command line several commands to be executed sequentially. You simply separate each full command from the next with a semi-colon.

For instance:

```
copy myfile /D1/newfile; dir >/P
(ENTER)
```

According to this command, the shell first executes the Copy command. Then it enters the "waiting" state until Copy terminates, at which time it executes Dir.

If an error is returned by any program, the shell doesn't execute subsequent commands on the same line, regardless of the state of the x (abort on error) option. Otherwise, ; and (ENTER) are identical command separators.

Here are two more examples of commands using the semi-colon separator:

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

dir /D1/MYFILE; list temp >/P;
del temp (ENTER)
```

Note: In a command line with semi-colon separators,
even though commands are listed in a particular se-
quence and executed in that sequence, they are in fact
all separate and equal child processes of the shell.

Concurrent Execution

The second kind of command separator is the ampersand (&)
which specifies concurrent execution. The first program you
specify is run as a separate, child process of the shell. But the
shell doesn't wait for it to finish before processing the next
command.

The concurrent execution separator is the way you specify
multiprogramming (running two or more programs simul-
taneously). With the & directing the shell to divide CPU time
equally between the processes you name in your command
line.

The number of programs that can run at the same time isn't
fixed. It depends on the amount of free memory in the system
versus the memory requirements of the programs to be run.

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. It
immediately displays both the number of the new process and
a new prompt for you, like this:

```
&007
OS9:
```

You can then enter another command, which will also be
executed while output from your command continues to go to
the printer. That's a real timesaver for you. You don't spend
unproductive time waiting for OS-9 to finish a task.

36

Note: If you have several processes running simultaneously, and want information about them, you can use the procs command.

You can, if you want, use both the concurrent and sequential command separators in a single command line, like this:

```
dir >/P& list file1& copy file1
file2; del temp (ENTER)
```

Because they're joined by & modifiers, the Dir, List, and Copy programs run concurrently. But the Del program doesn't run until the others are terminated because the command line contains a semi-colon to specify sequential execution for Del.

Pipes and Filters

The third kind of command separator is !, which is used to construct "pipelines". Pipelines consist of two or more concurrent programs whose standard input and/or output paths connect to each other using "pipes".

Pipes are the primary means by which data is transferred from process to process — they're vital to interprocess communications. Pipes are first-in, first-out buffers, "holding areas" for data.

I/O transfers using pipes are automatically buffered and synchronized. A single pipe can have several "readers" and several "writers". Multiple writers send, and multiple readers accept, data to/from the pipe on a first-come, first-served basis. An end-of-file occurs if an attempt is made to read from a pipe when there are no writers available to send data. Conversely, a write error occurs if an attempt is made to write to a pipe with no available readers.

Pipelines are created by the shell when it processes an input line with one or more ! separators. For each !, the standard output of the program named to the left of the ! is redirected through a pipe to the standard input of the program named to the right of the !. Individual pipes are created for each ! in the command line. For example:

```
update <master_file ! sort
write_report >/P (ENTER)
```

Here, the Update program has its input redirected (from its
standard input, the keyboard) to become master_file. The
standard output from that first command, because of the !,
becomes the standard input for the program sort. The output
of sort, in turn — because of another ! — becomes the stan-
dard input for the program write_report, which has its stan-
dard output redirected to the printer.

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

Programs which are specifically designed to process data us-
ing a pipeline or multiple pipelines are often called "filters".
The Tee command, which uses pipes to allow data to be
simultaneously "broadcast" from a single input path to
several output paths, is a useful filter.

Some of the most useful applications of pipelines are jobs like
character set conversion, print file formatting, data compress-
ion/decompression.

3.4 Command Grouping

Sections of shell input lines can be enclosed in parentheses.
This permits modifiers and separators to be applied to an en-
tire set of programs.

The shell processes the material in the parentheses by calling
itself recursively to execute the enclosed program list.

For example, if you want the "table of contents" of the root
directory of drive 0 and then the root directory of drive 1 to
go directly to the printer, you can type either:

```
dir /D0 >/P; dir /D1 >/P (ENTER)
```

38

or:

```
(dir /D0; dir /D1) >/P  (ENTER)
```

The results are identical. The only difference is that the printer is "kept" continuously in the second example. In the first example, another user could "steal" the printer in between the Dir commands.

You can use command grouping to cause a group of programs to be executed sequentially, but also 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 the &. However, the shell deletes the three specified files sequentially because you're using the semicolon within the parentheses.

A useful extension of this form is to construct pipelines consisting of both sequential and concurrent programs. For instance:

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

The shell first processes the output of the first Dir command and then the second. Then it sends all the Dir output together to makeuppercase; then all the output, still together, is transmitted.

3.5 Built-in Shell Commands and Option:

When processing input lines, the shell looks for several special names of commands or option switches that are built into the shell.

These commands are executed without loading a program and creating a new process, and generally affect how the shell

operates. They can be used at the beginning of a command line, or following any program separator — ;, &, or !.

Two or more adjacent built-in commands can be separated by spaces or commas.

The built-in commands and their functions are:

chd pathlist	changes the working data directory to the directory specified by the pathlist.
chx pathlist	changes the working execution directory to the directory specified by the pathlist.
ex modname	directly executes the module named. This transforms the shell process so that it ceases to exist, and a new module begins execution in its place.
w	waits for any process to terminate.
* text	allows you to make a ''comment''. Whatever text you specify isn't processed by the shell.
kill procID	aborts the process specified.
setpr procID	number changes the process's priority number.
x	causes the shell to abort on any error.
-x	causes the shell not to abort on error.
p	turns the shell prompt and messages on (default).
-p	inhibits the shell prompt and messages.
t	makes the shell copy all input lines to output.
-t	doesn't copy input lines to output (default).

The Chd and Chx commands switch the shell's working directory and, by inheritance, any subsequently created child process.

The Ex command is used where the shell is needed to initiate execution of a program without the overhead — for instance time spent and memory tied up — of a suspend shell process. The module named is processed according to standard shell operation, and modifiers can be used.

3.6 Shell Procedure Files

The shell is a reentrant program, which means that 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, all of which you can execute simply by running the one procedure file.

For example, the Deldir command is a procedure file that comes with the OS-9 system. It's actually a sequential string of commands (Del, Attr, and again Del), but you execute them all with the single command Deldir. This technique is sometimes called "batch" or "background" processing.

> **Note:** If you have occasion to enter the same command sequences repeatedly, you can build your own procedure files by using the Build command.

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" which accepts and carries out the commands in the procedure file.

When you enter any command line, if the shell can't find the specified program in memory or in the execution directory, 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.

Besides eliminating repetitive manual entry of commonly used command sequences, procedure files can allow the computer to execute a lengthy series of programs while it's unattended, or even while it's running other programs. That, of course, frees you to do other things.

To run a procedure file — for instance, one you've created and called mailsequence — type either:

shell mailsequence (ENTER)

or

mailsequence (ENTER)

Both do exactly the same thing: create a subshell which runs the commands you've built into your mailsequence procedure file.

If you want to run a procedure file in a concurrent mode, use the ampersand (&) modifier. OS-9 doesn't place any constraints on the number of procedure files you can run concurrently, as long as there's memory available.

You can even build procedure files so that they themselves cause sequential or concurrent execution of other procedure files.

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

3.7 Error Reporting

Many programs (including the shell) use OS-9's standard error reporting function, which displays an error number on

the error output path. (The standard error codes are listed in Appendix A of this manual.) If, you want you can execute the Printer command. It replaces the smaller, built-in error display routine with a larger (and slower) routine that looks up descriptive error messages from a text file called /D0/SYS/ errmsq. Once you run the printer command, it can't be turned off. Also, its effect is system-wide.

Programs called by the shell can return an error code in the "B" register (otherwise B should be cleared) on termination. This type of error, as well as errors detected by the shell itself, causes an error message to be displayed, and processing of the command line or procedure file to be terminated, unless the X (don't abort on error) built-in command has been executed.

3.8 Running Compiled Intermediate Code Programs

Before the shell executes a program, it checks the program module's language type. If it isn't 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 this command:

```
BASIC09 adventure (ENTER)
```

or you can accomplish the same thing by typing:

```
adventure (ENTER)
```

Both command lines automatically call the BASIC09 run-time system.

3.9 Editing *startup* for Timesharing Systems

Your OS-9 system has a procedure file called startup, which among other things, asks you for the date and time each time you use the system.

If you're setting up your Color Computer as a timesharing system — that is, if you're adding a terminal to it — you should alter the startup file so that whoever is using the other terminal will have appropriate access to the system.

Use the List command to look at the present contents of startup. Remember the contents. Then use the build command to create a startup procedure file, exactly like the original one except that you add this line at the end:

    tsmon /T1& (ENTER)

(You still have to press (ENTER) again to signal an end-of-file.) The tsmon command is the system's timesharing monitor, and opens standard I/O paths for the terminal, in addition to running the login sequence.

4/Multiprogramming And Memory Management

One of the most valuable capabilities of OS-9 is multiprogramming, which is sometimes called timesharing or multitasking. This lets your computer run more than one program at the same time. Multiprogramming can be a tremendous advantage in many situations. For example, you can be editing one program while another is being printed. Or you can use your Color Computer to control household automation and be able to use it at the same time for routine work and entertainment.

OS-9 uses this capability all the time for internal functions. The simple way for you to use it is by putting the & character at the end of a command line the & causes the shell to run your command as a "background", or concurrent task (see Chapter 3).

In order to allow several programs to run simultaneously and without interference, OS-9 performs many coordination and resource allocation functions. The major system resources OS-9 manages are:

- The input/output system

- CPU time

- Memory

The input/output system is discussed in Chapter 3. This chapter is designed to give you some basic information about how OS-9 uses CPU time and memory to optimize system throughput, and to make efficient multiprogramming a reality.

4.1 Processor Time Allocation and Timeslicing

CPU time is a precious resource that must be allocated wisely to maximize the computer's throughput.

Many programs by their nature spend time waiting. For example an interactive program, has to wait for the user to enter

information from the terminal. Meanwhile, the program can't accomplish anything. It can only wait.

On most systems, programs tie up CPU time while they're waiting. But multiprogramming systems like OS-9 are far more efficient than that. They assign CPU time to only those programs that can effectively use the time.

OS-9 uses a technique called "timeslicing," which allows all active processes to share CPU time. Timeslicing uses both hardware and software functions, and works this way: On OS-9, 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 timeslices.

Those timeslices – 60 every second –– are allocated to the different processes on the system. At any tick, OS-9 can suspend execution of one program and begin execution of another. (The starting and stopping of programs doesn't affect their execution.)

How frequently OS-9 gives a program timeslices depends on the program's assigned priority, relative to the assigned priority of other active processes. Process priority is expressed as a decimal number from 0 through 255, with 0 representing the highest priority and 255 the lowest.

OS-9 automatically gives the shell program a priority of 0. Since child processes inherit their parents' priorities, the shell's child processes all have priorities of 0. You can, if you want, find a process's priority number by using the Procs command. You can change the priority number by using the Setpr command.

It's not possible to compute exactly the percentage of CPU time assigned to any particular process, because of dynamic variables such as time the process spends waiting for I/O devices. But you can roughly approximate the percentage by dividing the process's priority number by the sum of the priority numbers of all active processes:

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

Note: Timeslicing happens so quickly that it looks to a human observer as if all processes are being executed simultaneously and continuously. If, however, the computer becomes overloaded with processing work, you may notice a delay in response to input from the terminal. Or you may notice that a "batch" program is taking longer than usual to run.

4.2 Process States

The CPU time allocation system automatically assigns programs/processes one of three "states" that describe their current status. Process states are also important for coordinating process execution. A process can be in only one state at any instant, although state changes may be frequent. The states are:

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

- Waiting – Applies to processes which are suspended until another process terminates. This state is used to coordinate execution of sequential programs. The shell, for example, is in the waiting state during the time a command program it's initiated is running.

- Sleeping – Applies to processes suspended by self-request for a specified time interval, or until receipt of a "signal". (Signals are internal messages used to coordinate concurrent processes.) This is the typical state of programs waiting for input/output operations.

Sleeping and waiting processes aren't assigned CPU time until they change to the active state.

Note: The Procs command gives information about process states.

4.3 Creation of New Processes

First, some terminology: when one process creates another process, the creator is called the "parent process", and the newly created process is called the "child process". The new child can itself become a parent by creating yet another process.

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

The sequence of operations required to create a new process and initially allocate resources to it is automatically performed by OS-9's "fork" function.

If for any reason any part of the sequence can't be performed, the fork is aborted and the prospective parent is passed an appropriate error code. The most frequent reason for failure is unavailablity of required resources (especially memory), or inability of the system to find the specified program.

A process can create many new processes, subject only to the limitation of the amount of unassigned memory available.

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 having the same name.

- Parameters, data specified by the parent to be passed to and used by the new process. This data is copied to part of the child process's memory area. (Parameters are frequently used to pass file names, initialization values, and other information.)

48

The new process also "inherits" copies of certain of its parent's properties. These are:

- A user number, which is used by the file security system and is used to identify all processes belonging to a specific user. (This is not the same as the "process ID", which identifies a specific process.) This number is usually obtained 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, error) used for routine input and output. Most paths can be shared simultaneously by two or more processes.

- Current (working) directories, the data directory and the execution directory.

- Process priority, which determines what proportion of CPU time the process receives.

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

- A process ID number, from 1 to 255. Each process has a unique ID number, useful for both the system and the user. Process ID numbers have no relationship with the process priority numbers. Whether an ID number is low or high makes no difference.

- Memory, enough for the new process to be able to run. In OS-9, all processes share a single address in memory. A data area, used for the program's parameters, variables and stack is allocated for each process's exclusive use. A second memory area may also be needed to load the program if it's not resident in memory.

In summary, each new process has associated with it — from one source or another:

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

4.4 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 (meaning which program modules are resident in memory at any given time). The result is that you seldom have to be bothered with the actual memory addresses of programs or data.

Within the address space, memory is assigned from higher addresses downward for program modules, and from lower addresses upward for data area, as shown below:

```
                                                      highest address
+-----------------------------------+
|                                   |
|         program modules           |
|         (RAM or ROM)              |
|                                   |
+-----------------------------------+
|                                   |
|          unused space             |
|         (RAM or empty)            |
|                                   |
+-----------------------------------+
|                                   |
|           data areas              |
|           (RAM)                   |
|                                   |
+-----------------------------------+
                                                      lowest address
```
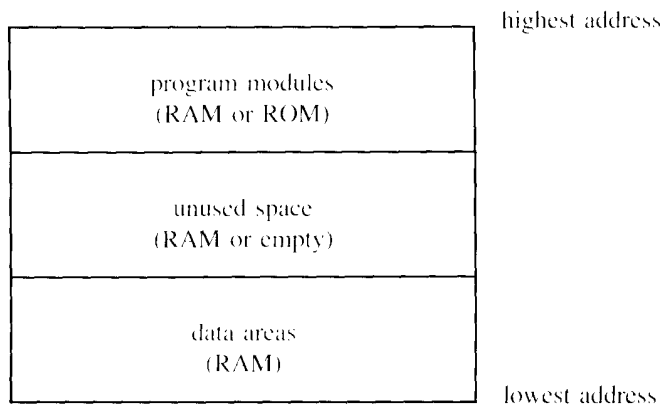
Loading Program Modules into Memory

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 "reentrant code", which means that the exact "copy" of a program can be shared by two or more different processes simultaneously without affecting each other, provided that each "incarnation" of the program has an independent memory area for its own variables.

Almost all OS-9 family software is reentrant, and can make the most efficient use of memory. For instance, BASIC09 requires 22K bytes of memory in order to be loaded. Suppose that OS-9 receives a request (from a process) to run BASIC09, but has already caused it to be loaded into memory at the request of another process. OS-9 doesn't have to cause another copy to be loaded, using another 22K of memory. Instead both processes share the same copy of BASIC09.

OS-9 automatically keeps track of how many processes are using each program module and deletes the module when all processes using the module have terminated. This frees the modules memory for other uses.

If the requested program module isn't already 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 for variable storage should be allocated to the process. The module header also includes other important descriptive information about the program, and is an essential part of OS-9 operation at the machine language level. (For detailed description of memory modules and module headers, check the *OS-9 Technical Information* manual.)

Programs can also be explicitly loaded into memory using the Load command. As with "fork", the program is actually loaded only if it isn't already in memory.

If the module isn't in memory, OS-9 copies the requested module from the file into memory and verifies the CRC. If the module isn't already in the module directory, OS-9 adds it to the directory. This process is repeated until all the modules in the file are loaded, the 64K memory limit is exceeded, or until a module with an invalid format is encountered. OS-9 always links to the first module read from the file.

If the program module is already in memory, the load process still begins in the same way: OS-9 loads the module from the file and verifies the CRC. But then when it attempts to add the module to the module directory, and notices that the module is already known there, it merely increments the known module's link count — the number of processes using the module.

You can use Load to "lock" a program into memory. That's a timesaver if you need to use the same program repeatedly. With Load the program is kept continuously in memory. OS-9 doesn't have to take the time to load it each time you use it.

The opposite of Load is the Unlink command, which decreases a program module's link count by one. When this count becomes zero, indicating that the module is no longer used by any process, the module is deleted. Its memory is deallocated and its name is removed from the module directory. The Unlink command is generally used in conjunction with the Load command. (Programs loaded by "fork" are automatically unlinked when the program terminates).

Suppose, for instance, you're planning to use the Copy command 10 times in a row. Normally the Copy program is loaded each time you enter the Copy command. But if you lock the Copy module into memory, and then enter your string of commands, you won't have to wait while Copy is loaded and unloaded repeatedly. You'll finish your work more quickly. When you're done, 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's important to use Unlink after the program is no longer needed; otherwise the program continues to occupy memory which could be used for other purposes.

> **Note:** Be very careful not to unlink modules in use by any process, or you'll cause the memory used by the module to be deallocated, and its contents destroyed. The result is a user's nightmare: all programs using the unlinked module crash.

Loading Multiple Programs:

Another important aspect of program loading is the ability to have two or more programs resident in memory at the same time. This is possible because all OS-9 program modules are written as "position-independent code", or "PIC". PIC programs don't have to be loaded into specific, predetermined memory addresses to work correctly. They can, therefore, be loaded at different memory addresses at different times.

PIC programs require special types of machine language instructions which few computers have. The ability of the 6809 microprocessor to use this type of program is one of its most powerful features, and one of the greatest aids toward multi-programming.

Since more than one program can reside in memory, you can therefore use the Load command two or more times (or have a single file contain several memory modules), and each program module is automatically loaded at different, non-

53

overlapping addresses. (Most other operating systems write over the previous program's memory whenever a new program is loaded).

This technique also means that you don't have to be directly concerned with absolute memory addresses.

> **Note:** Any number of program modules can be loaded until available system memory is full.

Memory Fragmentation

Even though PIC programs can be loaded initially at any address where free memory is available, program modules can't be relocated dynamically afterwards. That means that once a program is loaded, it must remain at the address at which it was originally loaded.

This characteristic can lead to a sometimes troublesome phenomenon called "memory fragmentation". When a program is loaded, it's assigned the first sufficiently large block of memory at the highest address possible in the address space. If several program modules are loaded, and subsequently one or more modules located in between other modules is unlinked there'll be several fragments of free memory space. The sum of the sizes of the free memory spaces may be quite large. But because they're scattered, there won't be enough free space in a single block to load a program module larger than the largest free space.

The Mfree command shows the location and size of each unused memory area, and the Mdir e command shows the address, size, and link (use) count of each module in the address space. You can use both commands to detect fragmentation. And you can usually "defragment" memory by unlinking scattered modules and reloading them. (Make certain none of the modules is in use before doing so.)

5/Use Of The System Disk

OS-9 systems use a system disk to load many parts of the operating system during system startup, and to provide files frequently used during normal system operations. Therefore, the system disk is generally kept in disk drive zero (D0) when the system is running.

Two files used during the system startup operation, OS9Boot and startup are provided and must remain in the system disk's root directory. Other files on the system disk are organized into three directories: CMDS (commands), DEFS (system-wide definitions), and SYS (other system files). Still other files and directories created by the system manager and/or users can also reside on the system disk. (These frequently include each user's initial data directory.)

5.1 The *OS9BOOT* File

The file called OS9Boot is loaded into RAM memory by the "bootstrap" routine located in the OS-9 firmware. It includes file managers, device drivers and descriptors, and any other modules which are permanently resident in memory. The OS-9 System Master Disks OS9Boot file contains these modules:

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
CCIO	Keyboard video graphics device driver
PRINTER	Printer device driver
RS232	RS-232 serial port device driver
CCDisk	Disk driver
D0, D1, D2, D3	Disk device descriptor
TERM	Terminal device descriptor
P	Printer device descriptor
T1	RS-232 serial port device descriptor
Shell	Input Output commands interface module

P	Printer (serial) device descriptor
Clock	Real-time clock module
SYSGO	System startup process

You can create new bootstrap files, which can include additional modules, by using the OS9Gen command. Any module loaded as part of the bootstrap can't be unlinked, and is stored in memory with a minimum of fragmentation. You may find it advantageous to include in the OS9Boot file any module you use constantly during normal system operation.

5.2 The SYS Directory

The directory /D0/SYS contains two important files:

- password (the system password file — see the Login command)
- errmsq (the error message file — see 3.7)

These files, and the SYS directory itself, aren't absolutely required to boot OS-9, but they're needed if you plan to use Login, Tsmon, or Printerr. You can add other system-wide files of a similar nature, if you want to.

5.3 The *startup* File

The title /D0/startup is a shell procedure file which is automatically processed immediately after system startup. You can include in startup any legal shell command line. Many people choose to include Setime to start the system clock. If this file isn't present, the system will still start correctly but you have to run Setime manually.

5.4 The CMDS Directory

The directory /D0/CMDS is the system-wide command object
code directory, normally shared by all users as their working
execution directory. The vital shell program is part of CMDS.
The system startup process "sysgo" makes CMDS the initial
execution directory.

5.5 The DEFS Directory

The directory /D0/DEFS contains assembly language source
code files. They in turn contain common system-wide sym-
bolic definitions normally included in assembly language pro-
grams by means of the OS-9 assembler "use" directive. The
presence and use of this directory is optional, but highly rec-
ommended for any system used for assembly language pro-
grams. The files commonly contained in this directory are:

- OS9Defs   Main system-wide definition file
- RBFDefs   — RBF file manager definition file
- SCFDefs   SCF file manager definition file
- SysType   — System types definition file

5.6 Changing System Disks

Most OS-9 users prefer to leave the system disk in place
while the system is running, particularly with multiuser sys-
tems. Leaving it in place guarantees that it won't be taken
away just when someone is using it.

If you do remove the disk and begin to use another one, let
OS-9 know where you want to be on the new disk by using
the Chd and Chx commands. (For directions, see Chapters 2
and 6.) Those commands set both working directory pointers
— data and execution — for the new disk.

In general, it's unwise to remove a disk and insert another while any files are open. And it's just plain dangerous to your data to make a disk exchange if any files on the first disk are open in the "write" or "update" modes.

5.7 Making New System Disks

To make a system disk, it's necessary to do these four things:

1. Format the new disk

2. Create and link the OS9Boot file by using the OS9Gen or Cobbler command

3. Create or copy the startup file

4. Copy the CMDS and SYS directories, and the files they contain

You can perform steps 2 through 4 manually, or do them automatically by using any one of these methods:

● Create and use a shell procedure file

● Use a shell procedure file generated by the Dsave command

● Use the Backup command

6/System Command Descriptions

This chapter contains alphabetical descriptions of each of the command programs supplied with OS-9. The commands are ordinarily called using the shell, but can also be called from most other programs in the OS-9 family, including BASIC09, Interactive Debugger, Macro Text Editor, and others. Unless otherwise noted, the programs described in this section are designed to run as individual processes.

6.1 Organization of Entries

Each command entry is organized to include:

- The name of the command

- A "syntax" line, which shows you what format, or "syntax" to use when you type the command

- A brief definition of what the command does

- Further details about the command and how to use it

- Information about any options available with the command

- One or more examples of command usage

6.2 Command Syntax Notations

It's important to enter the various parts of a command in the correct order, and in the correct format. The syntax line in each command description helps you do that by showing you exactly what each command requires.

The syntax line always begins with the name of the command. Occasionally, that's all you'll need (except, of course, for pressing (ENTER)). But other commands either require, or will accept, parameters - variables which give instructions to OS-9. And many commands offer you built-in options.

The syntax line gives you that information by using these notations:

Italics — Italics indicates a variable for you to supply, for instance the name of a file (*filename*), a directory (*directory name*), or a complete path to a file or directory (*pathname*). Some other variables are: devices (*devname*), memory modules (*modname*), process ID numbers (*procID*), options (*options*), a list of parameters (*paramlist*), and text (*text*). (In a command situation, *text* means a character string terminated by an end-of-line.) Another variable you will encounter is *arglist*, or argument list. Similar to *paramlist*, but is generally broader in scope, including modifiers, program specifications and so forth.

[   ] — Brackets indicate that the material within them is optional to the command.

... — An ellipsis indicates that the material immediately preceding can be repeated within the command. For instance, [*filename*][...] means that you can, specify more than one filename to the command.

The command syntax doesn't include the shell's built-in options (for instance I/O redirection), because the shell filters out its options before the command line is passed to the program being called.

6.3 System Commands

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

The following list is a summary of these commands:

Attr	Change file attributes
Backup	Make disk backup
Binex	Convert binary to s-record
Build	Build text file
Chd	Change working data directory
Chx	Change working execution directory
Cmp	File comparison utility
Cobbler	Make bootstrap file

Copy	Copy data
Date	Display system date and time
Dcheck	Check disk file structure
Del	Delete a file
Deldir	Delete all files in a directory system
Dir	Display file names in a directory
Display	Display converted characters
Dsave	Generate procedure file to copy files
Dump	Formatted file dump
Echo	Echo text to output path
Exbin	Convert s-record to binary
Format	Initialize disk media
Free	Display free space on device
Ident	Print OS-9 module identification
Kill	Abort a process
Link	Link module into memory
List	List contents of disk file
Load	Load module(s) into memory
Login	Timesharing system log-in
Makdir	Create directory file
Mdir	Display module directory
Merge	Copy and combine files
Mfree	Display free system RAM memory
OS9Gen	Build and link a bootstrap file
Printerr	Print full-text error messages
Procs	Display processes
Pwd	Print working directory
Pxd	Print execution directory
Rename	Change file name
Save	Save memory module(s) on a file
Setime	Activate and set system clock
Setpr	Set process priority
Sleep	Suspend process for period of time
Shell	OS-9 command interpreter
Tee	Copy standard input to multiple output paths
Tmode	Change terminal operating mode
Tsmon	Timesharing monitor
Unlink	Unlink memory module
Verify	Verify or update module header and CRC
Xmode	Examine or change device initialization mode

ATTR

ATTR *filename* [*permission abbreviations*]

Examines or changes the security permissions of a file.

To enter the command, type Attr followed by the name of the file whose security permissions are to be changed. Then type a list of permissions which are to be turned on or off. A permission is turned on by giving its abbreviation, or turned off by preceding its abbreviation with a minus sign. Permissions not explicitly named are not affected. If no permissions are given, the current file attributes will be printed.

You can't change the attributes of a file you don't own. User zero, can change the attributes of any file in the system.

File permission abbreviations are:

d	Directory
s	Shareable file
r	Read permit to owner
w	Write permit to owner
e	Execute permit to owner
pr	Read permit to public
pw	Write permit to public
pe	Execute permit to public

You can use the Attr command to change a directory to a file if all entries have been deleted from it. You can't change a file to a directory with this command (see Makdir).

Examples:

        attr myfile -pr -pw (ENTER)

removes read and write permissions from the public.

        attr myfile r w e pr pw pe (ENTER)

gives the file's owner, and the public, read, write and execute permissions.

```
attr datalog (ENTER)
-s-wr-wr
```

Since the command doesn't specify permissions, Attr displays
the current permissions of the Datalog file.

BACKUP

BACKUP [e] [s] [-v] [*devname*] [*devname*]

Physically copies all data from one device to another.

A physical copy is performed, sector by sector, without re-
gard to file structures. In almost all cases, the devices speci-
fied *must* have exactly, the same format (size, density, and so
forth) and must not have defective sectors.

If you omit both device names, the names /D0 and /D1 are
assumed. If you omit only the second device name, a single-
unit backup will be performed on the drive specified.

Options are:

e	Exits if any read error occurs
s	Prints single-drive prompt message
-v	Does not verify
#nK	Allow more memory (n = amount of memory), and therefore speed up the backup procedure

Examples:

```
backup /D2 /D3 (ENTER)
```

makes a backup of the diskette in Drive 2 on to the diskette in
Drive 3.

```
backup -v (ENTER)
```

assumes the names D0 and D1, and makes the appropriate
backup without verification.

```
backup (ENTER)

Ready to BACKUP from /D0 to
/D1 ?: Y (ENTER)
MYDISK
   is being scratched
OK ?: Y
Number of sectors copied: $0276
Verify pass
Number of sectors verified: $0276
```

This example shows a complete interchange between the Backup command and the user who entered it. In the example /D1, the destination disk, is named MYDISK. "Scratched" means "erased".

The following is an example of a single-drive backup. Backup reads a portion of the source disk into memory and then prompts you to remove the source disk and put the destination disk into the drive. Then Backup writes on the destination disk. Then you remove the destination disk and put the source disk back into the drive. This continues until Backup copies the entire disk. Giving Backup as much memory as possible will necessitate fewer disk exchanges.

```
backup /D0 #32K (ENTER)

Ready to BACKUP from /D0 to
D0 ?: Y
Ready DESTINATION, hit a key:
MYDISK
   is being scratched
OK ?: Y
Ready SOURCE, hit a key:
Ready DESTINATION, hit a key:
Ready SOURCE, hit a key:
Ready DESTINATION, hit a key:

(several repetitions)

Ready DESTINATION, hit a key:
Number of sectors copied: $0276
Verify pass
Number of sectors verified: $0276
```

BINEX
EXBIN

BINEX *filename1 filename2*
EXBIN *filename1 filename2*

Binex converts a binary file into an S-Record file, and Exbin converts an S-Record file into a binary file.

An S-Record file is a type of text file that contains records representing binary data in hexadecimal character form. This Motorola-standard format is often directly accepted by commercial PROM programmers, emulators, logic analyzers and similar devices that are RS-232-interfaced. It can also be useful for transmitting files over data links that can handle only character-type data; or to convert OS-9 assembler- or compiler-generated programs to load on non-OS-9 systems.

Binex converts *filename1*, an OS-9 binary format file, to a new file named *filename2* in S-Record format. If you invoke Binex on a non-binary load module file, OS-9 prints a warning message and asks you if Binex should proceed anyway. A "Y" response means yes; any other answer will terminate the program. S-Records have a header record to store the program name for informational purposes, and each data record has an absolute memory address which is not meaningful to OS-9 since it uses position-independent code. However, the S-Record format requires them, so Binex will prompt the user for a program name and starting load address. For example:

```
binex /D0/CMDS/scanner scanner.S1
  (ENTER)
Enter starting address for file:
  $100
Enter name for header record:
  scanner
```

To download the program to a device such as a PROM programmer (for example, using serial port /T1), type:

```
list scanner.S1 >/T1 (ENTER)
```

Exbin is the inverse operation; *filename1* is assumed to be an S-Record format text file which Exbin converts to pure binary form on a new file called *filename2*. The load addresses of each data record must describe contiguous data in ascending order.

Exbin doesn't generate or check for the proper OS-9 module headers or CRC check value required to actually load the binary headers or CRC check value required to actually load the binary file. You can use the Ident or Verify commands to check the validity of the modules if they're to be loaded or run.

Example:

```
exbin program.S1 CMDS/program
(ENTER)
```

BUILD

BUILD *filename*

Builds short text files by copying the standard input path into the file specified by *filename*. Build creates a file according to the *filename* parameter, then displays a "?" prompt to request an input line. Each line entered is written to the output path (the file). Entering a line consisting of only a carriage return terminates Build.

Examples:

```
build small_file (ENTER)
```

creates a new file called small _ file and puts into it whatever you type at the keyboard.

```
build /P (ENTER)
```

directs whatever you type to the printer.

```
build <mytext /T1 (ENTER)
```

Using Build, you can also transfer, or redirect, material from one file to another. Instead of the keyboard, the standard input path is the first file you name in the command. The output path is the second. In this example, the mytext file becomes the input path, and is copied to Terminal 1, the output path.

```
build newfile (ENTER)

? THE POWERS OF THE OS-9 (ENTER)
? OPERATING SYSTEM ARE TRULY (ENTER)
? FANTASTIC, (ENTER)
? (ENTER)

list newfile (ENTER)

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

This example shows an interchange between Build and the user. After building newfile, the user employs the List command to check the contents of the newly built file.

CHD
CHX

CHD *pathname* or *directory name*
CHX *pathname* or *directory name*

Chd changes the current data directory, and Chx changes the current execution directory.

Many commands in OS-9 work with user data such as text files, programs and so forth. These commands assume that a file is located in the working data directory. Other OS-9 commands assume that a file is in the working execution directory.

The Chd and Chx commands don't appear in the CMDS directory, because they're built in to the shell.

Examples:

```
chd /D1/PROGRAMS (ENTER)
```

change the current data directory the PROGRAMS data directory located on the diskette in drive 1. This example shows the use of a pathname.

```
chx .. (ENTER)
```

moves the user to the directory immediately above the current execution directory.

```
chx binary_files/text_programs
    (ENTER)
```

is another example of the use of a pathname to change another execution directory.

```
chx /D0/CMDS; chd /D1 (ENTER)
```

changes both the execution and data directories.

CMP

CMP *filename1: filename2*

Opens two files and performs a comparison of the binary values of the corresponding data bytes of the files.

If any differences are encountered, the file offset (address) and the values of the bytes from each file are displayed in hexadecimal.

The comparison ends when an end-of-file marker is encountered on either file. Cmp then displays a summary of the number of bytes compared and the number of differences found.

Examples:

```
cmp red blue (ENTER)

    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

    cmp red red (ENTER)

    Differences

        None ...

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

COBBLER

COBBLER *devname*

Creates the OS-9BOOT file required on any disk from which OS-9 is to be bootstrapped.

The boot file consists of the *same* modules which were loaded into memory during the most recent bootstrap. (To add modules to the bootstrap file use the OS9Gen command.) Cobbler also writes the OS-9 kernel on the first fifteen sectors of track 34, and excludes these sectors from the diskette allocation map. If any files are present on these sectors, Cobbler will display an error message.

The boot file must fit into one contiguous block on the diskette. For this reason, Cobbler is normally used on a freshly formatted diskette. If Cobbler is used on a diskette without a

contiguous block of storage large enough to hold the boot
file, the old boot file may be destroyed, and OS-9 won't be
able to boot from that diskette until it's reformatted.

Example:

```
cobbler /D0 (ENTER)

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

Saves current device attributes on the current system disk.

Note: This command is often used after Xmode to per-
manently change device attributes.

COPY

COPY *pathname pathname* [-s]

Copies data from the first file or device specified to the
second.

The first file or device must already exist. The second file is
automatically created if the second pathname is a file on a
diskette. Data can be of any type and is not modified in any
way as it's copied.

Copy transfers data using large block reads and writes until it
reaches an end-of-file marker on the input path. Because
block transfers are used, normal output processing of data
doesn't occur on character-oriented devices such as terminals
and printers. Therefore it's better to use List Copy when a
file consisting of text is to be sent to a terminal or printer.
With Copy, important codes (e.g. line feed) won't be added.

The -s option causes Copy to perform a single-drive copy
operation. The second pathname must be a full one if you use
-s. In a single-drive procedure, Copy reads a portion of the
source disk into memory. Then you remove the source disk
and put the destination disk into the drive, and enter a "C"

then Copy writes on the destination disk, with the process continuing until the entire file is copied.

Using the shells alternate memory size modifier to give a large memory space increases speed and reduces the number of media exchanges required for single-drive copies.

Examples:

```
copy file1 file2 #15K (ENTER)
```

copies file1 to file2 giving 15K of memory.

```
copy /D1/JOE/news /D0/PETER
messages (ENTER)
```

copies the news file on the diskette in Drive 1 to the messages file on the diskette in Drive 0.

```
copy /TERM /P (ENTER)
```

sends — copies   to the printer of anything you type into the console.

```
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 is an example of the alternating method used in a single-drive copy operation.

DATE

DATE [t]

Displays the current date, and, if you use the t option, the current system time.

Examples:

```
date t (ENTER)
```

displays the system date and time

```
date t >/P (ENTER)
```

directs the command's output to the printer.

```
setime (ENTER)

          YY/MM/DD   HH.MM.SS
TIME?  81/04/15   14.19.00

date (ENTER)

April 15, 1981

date t (ENTER)
April 15, 1981   14.20.20
```

This sequence illustrates setting a new date and time for the system by using the Setime command, and then using Date and it's t option to check system date and time.

DCHECK

DCHECK [-opts] devname

Checks disk file structure.

It's possible for sectors on a diskette to be marked as being allocated, but in fact not to be actually associated with a file or the diskette's free space. This can happen if a diskette is removed from a drive while files are still open, or if a directory which still contains files is deleted. Dcheck is a diagnostic you can use to detect this condition, as well as to check the general integrity of the directory/file linkages.

Dcheck is given the drive number of the diskette to be checked as a parameter. After verifying and printing some vital file structure parameters, Dcheck follows pointers down the diskette's file system tree to all directories and files on the

diskette. As it does so, it verifies the integrity of the file descriptor sectors, reports any discrepancies in the directory/file linkages, and builds a sector allocation map from the segment list associated with each file. If any file descriptor sectors (FDS) describes a segment with a cluster not within the file structure of the diskette, Dcheck reports a message like this:

> *** Bad FD segment ($xxxxxx-$yyyyyy) for file:
> (pathname)

This indicates that a segment starting at sector $xxxxxx$ and ending at sector $yyyyyy$ can't really be on this diskette. There's a good chance the entire FD is bad if any of its segment descriptors is bad. The allocation map is not updated for corrupt FDs.

While building the allocation map, Dcheck also makes sure that each diskette cluster appears once and only once in the file structure. If it discovers duplication, Dcheck displays a message like:

> Cluster $xxxxxx$ was previously allocated

This message indicates that Dcheck has found cluster $xxxxxx$ at least once before in the file structure. The message may be printed more than once if a cluster appears in a segment in more than one file.

Then Dcheck compares the newly created allocation map with the allocation map stored on the diskette, and reports any differences in messages like:

> 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) was found not to be part of the file system, but was marked as allocated in the diskette's allocation map. In addition to the causes mentioned in the first paragraph, some sectors may have been excluded from the allocation map by the Format program because they were defective. Or they may be

the last few sectors of the diskette, the sum of which was too small to compose a cluster.

The second message indicates that the cluster starting at sector *xxxxx* is part of the file structure but is not marked as allocated in the diskette's allocation map. It's possible that this cluster may be allocated to another file later, overwriting the contents of the cluster with data from the newly allocated file. (Any clusters that have been reported as "previously allocated" by Dcheck, as described above, surely have this problem.)

Dcheck options include:

-s	Displays count of files and directories only
-b	Suppresses listing of unused clusters
-p	Prints pathnames for questionable clusters
-w = *pathname*	Specifies path to directory for work files
-m	Saves allocation map work files
-o	Prints Dcheck's valid options

The -s option causes Dcheck to display a count of files and directories only. Only FDs are checked for validity. The -b option suppresses listing of clusters allocated but not in file structure. The -p option causes Dcheck to make a second pass through the file structure, printing the pathlists for any clusters that Dcheck finds as "already allocated" or "in file structure but not in allocation map". The -w option tells Dcheck where to locate its allocation map work file(s). The pathname specified must be a full pathname for a directory. (The directory /D0 is used if -w is not specified.) It is recommended that this pathlist not be located on the diskette being Dchecked if the diskette's file structure integrity is in doubt.

Dcheck builds its diskette allocation map in a file called *path-name*/Dcheckpp0, where *pathname* is as 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 diskette. If the -p option appears on the command line, Dcheck creates a second bitmap file (<*pathname2*>/Dcheckpp1) that has a bit set for each cluster Dcheck finds as

74

"previously allocated" or "in file structure but not in allocation map". Dcheck then makes another pass through the directory structure to determine the pathnames 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 diskette being checked. Otherwise Dcheck may be fooled if the diskette allocation map changes while it's building its bitmap file from the changing file structure. Dcheck can't process diskettes with a directory depth greater than 39 levels.

Examples:

```
dcheck /D2 (ENTER)

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)
Volume - 'System disk' 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 cluster
  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 disk' file structure is
  not intact
5 directories
25 files
```

DEL

DEL [-x] *filename* [...]

Deletes the file(s) specified.

You must have write permission for the file(s). Directories cannot be deleted unless they're changed to files or you use the Deldir command. (See the Attr command description.)

If you use the -x option, Del assumes the current execution directory.

Examples:

```
del text_program old_test_program
(ENTER)
```

deletes the two files specified.

```
del /D1/number_five (ENTER)
```

uses a complete pathname to specify the file named number _ five on the diskette in Drive 1.

```
del -x cmds.subdir/file (ENTER)
```

specifies a file called cmds.subdir/file in the current execution directory.

```
dir /D1 (ENTER)

directory of /D1 14.29.46
myfile     newfile

del newfile (ENTER)
dir /D1 (ENTER)

directory of /D1 14.30.37
myfile
```

In this interchange, the user first employs the Dir command to see what files are in the /D1 directory. Command output indicates that /D1 has two files: myfile and newfile. The user

employs the Del command to delete newfile, and then uses Dir again to make certain that newfile has been deleted.

DELDIR

DELDIR *directory name*

Deletes all files in a directory and the directory itself.

This command is a convenient alternative to manually deleting directories and the files they contain. Use it only when you want to delete everything in a directory, including the directory itself, other directories and all the subdirectories and files in them.

When Deldir runs, it prints a prompt message after the command line:

```
deldir OLDFILES (ENTER)
Deleting directory file,
List directory, delete directory,
    or quit ? (l/d/q)
```

An l response causes a Dir e command to run so you can see the files in the directory before they're deleted.

A d response initiates the deletion process.

A q response aborts the command before action is taken.

The directory to be deleted may include other directories which may themselves include other directories, and so forth. In this case, Deldir operates recursively (that is, it calls itself) so all lower-level directories are automatically deleted. The lower-level directories are processed first.

You must have correct access permission to delete all files and directories encountered. If not, Deldir will abort when it encounters the first file for which you don't have write permission.

The Deldir command automatically calls the Dir and Attr commands, so they must reside in the current execution directory.

DIR

DIR [e] [x] [*directoryname* or *pathname*]

Displays a formatted list of file names in a directory.

If no parameters are given, the current data directory is shown. If the x option is given, the current execution directory is shown. If a full pathname of a directory is given, it is shown. Results are displayed on the standard output path.

If the e option is included, each file's entire description is displayed: size, address, owner, permissions, date and time of last modification.

Examples:

    dir (ENTER)

displays the current data directory.

    dir x (ENTER)

displays the current execution directory.

    dir x e (ENTER)

displays the entire description of all files in the current execution directory.

    dir .. (ENTER)

displays the parent of the current working directory – the directory immediately above it in the hierarchy.

    dir newstuff (ENTER)

displays the newstuff directory

79

```
dir e TEXT_PROGRAMS (ENTER)
```

displays the entire description of all files in the directory
called TEST _ PROGRAMS.

DISPLAY

DISPLAY <hex> [...]

Reads one or more hexadecimal numbers given as param-
eters, converts them to ASCII characters, and writes them to
the standard output.

Display is commonly used to send special characters (such as
cursor and screen control codes) to terminals and other I/O
devices.

Examples:

```
display 0C >P (ENTER)
```

reroutes "form feed" — hex 0C — to the printer.

```
display 41 42 43 44 45 46 (ENTER)
ABCDEF
```

is an example of a command and the resulting output;
ABCDEF are ASCII characters corresponding to hex 41 42
43 44 45 46.

DSAVE

DSAVE [-opts] [devname] [directoryname or pathname]

Backs-up or copies all files in one or more directories.

Dsave is unlike most other commands in that it does not
directly affect the system. Instead, it generates a procedure
file which you execute later to actually do the work.

When you run Dsave, it writes copy commands to standard output to copy files from the current data directory on *devname* (the default is /D0) to the directory specified by *directoryname* or *pathname*. If you don't specify a directory name or pathname the copy is performed to the data directory that is the current directory at the time the Dsave procedure file is executed.

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. Since Dsave is recursive in operation, the procedure file exactly replicates all levels of the file system from the current data directory downward (such a section of the file system is called a "subtree").

If the current working directory happens to be the root directory of the disk, Dsave creates a procedure file that backs up the entire disk file by file. This is useful when it's necessary to copy many files from diskettes formatted differently, or from floppy diskettes.

Dsave options are:

-b	— Make output diskette a system diskette by using source diskette's OS9Boot file, if present
-b=*pathname*1	— Make output diskette a system diskette using *pathname* as source for the OS9Boot file
-i	— Indent for directory levels
-l	— Do not process directories below the current level
-m	— Do not include Makdir commands in procedure file
-s*integer*	— Set copy parameter to *integer* K

Examples:

```
chd /D2 (ENTER)
dsave /D2 >/D0/makecopy (ENTER)
chd /D1 (ENTER)
/D0/makecopy (ENTER)
```

The first command positions the user in /D2, the directory to be copied. Then Dsave makes a procedure file — actually a

directory — makecopy. The Chd command specifies that the
copy is to be made on the the /D1 directory, and the final
command executes the procedure file.

DUMP

DUMP [*filename* or *devname*]

Produces a formatted display of the physical data contents of
the path specified, which may be a diskette or any other I/O
device.

If you don't specify a filename, Dump uses the standard input
path — the keyboard. Dump writes output to the standard
output path — the video display. This command is commonly
used to examine the contents of non-text files.

Dump displays data 8 bytes to a line in both hexadecimal and
ASCII character format. Data bytes that have non-displayable
values are represented by periods in the character area.

The addresses displayed on the dump are relative to the be-
ginning of the file. Because memory modules are position-
independent and stored on files exactly as they exist in
memory, the addresses shown on the dump correspond to the
relative load addresses of memory-module files.

Examples:

> dump (ENTER)

displays keyboard input in hex. Output is written to the video
display.

> dump @/D1

The '@' symbol causes OS-9 to treat the entire disk as a file.

82

Sample Output:

```
dump SYS/password >/P

          0  1  2  3  4  5  6  7    0  2  4  6

Addr  8_9_A_B_C_D_E_F_  8_A_C_E_
0000  2C2C302C3132382C   ,,0,128,
0008  2F44302F434D4453   /D0/CMDS
0010  2C2E2C5348454C4C   ,,,SHELL
0018  0D55534552312C2C   .USER1,,
0020  312C3132382C2E2C   1,128,,,
0028  2E2C5348454C4C0D   ,,SHELL.
0030  55534552322C2C32   USER2,,2
0038  2C3132382C2E2C2E   ,128,,,,

          0  1  2  3  4  5  6  7    0  2  4  6

Addr  8_9_A_B_C_D_E_F_  8_A_C_E_
0040  2C5348454C4C0D55   ,SHELL.U
0048  534552332C2C332C   SER3,,3,
0050  3132382C2E2C2E2C   128,,,,,
0058  5348454C4C0D5553   SHELL.US
0060  4552342C2C342C31   ER4,,4,1
0068  32382C2E2C2E2C53   28,,,,,S
0070  48454C4C0D         HELL.
```

The first column indicates the starting address. The next eight columns (01 through EF) display data bytes in hexadecimal format. The final column (0 through E) displays data byes in ASCII format. Non-ASCII displayable bytes are shown as periods, in the ASCII character display section.

ECHO

ECHO *text*

Echoes entered text to the standard output path

Echo is typically used to generate messages in shell procedure files or to send an initialization character sequence to a ter-

minal. The text shouldn't include any of the punctuation characters used by the shell.

Examples:

```
echo HELLO, HOW'S IT GOING?
```

prints the message on the screen. This example would be useful as a background task.

```
echo >/P LISTING·OF PASSWORD FILE:
  list SYS/password >/P&
&003

eof
```

prints the message — — *listing of password file* — to the printer and lists the file SYS/password to the printer as a background task.

```
LISTING OF PASSWORD FILE
,,0,128,/D0/CMDS,,,SHELL
USER1,,1,128,,,,,SHELL
USER2,,2,128,,,,,SHELL
USER3,,3,128,,,,,SHELL
USER4,,4,128,,,,,SHELL

Here is the example run,

echo >/TERM **WARNING** DISK ABOUT
  TO BE SCRATCHED! (ENTER)
```

echoes the text to the console.

```
echo >/P LISTING OF TRANSACTION
  FILE; List Trans >/P& (ENTER)
```

combines two commands. The first echoes the entered text to the printer. The second (List) directs the contents of the trans file to the printer.

84

FORMAT

FORMAT *devname*

Physically initializes, verifies, and establishes an initial file structure on a diskette. All diskettes must be formatted before you can use them on an OS-9 system.

The diskette to be formatted must *NOT* be write protected.

The formatting process works this way:

1. Format physically initializes and sectors the diskette surface.

2. Format reads back and verifies each sector. If a sector fails to verify after several attempts, it's excluded from the initial free space on the diskette. As the verification proceeds, track numbers are displayed on the standard output device.

3. The diskette allocation map, root directory, and identification sector are written to the first few sectors of track zero. These sectors must not be defective.

Format will prompt for a diskette 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*.

FREE

FREE [*devname*]

Displays the number of unused 256-byte sectors on a device. These sectors are available for new files or for expanding existing files.

The device name you specify must be a disk drive. Free also displays the diskette's name, creation date, and cluster size. If you don't specify a device, drive 0 is assumed.

Data sectors are allocated in groups called "clusters". The number of sectors per cluster depends on the storage capacity and physical characteristics of the specific device. This means that small amounts of free space are divisible into fewer files. For example, if a given disk system uses 8 sectors per cluster, and a Free command shows 32 sectors free, a maximum of four new files could be created, even if each has only one cluster.

Examples:

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


free /D1 (ENTER)
DATA DISK created on: 83/06/16
Capacity: 630 sectors (1-sector
   clusters)
445 Free sectors, largest block
   442 sectors
```

IDENT

IDENT *filename* [*-opts*]

Displays header information from OS-9 memory modules.

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 prints 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 diskette file.

Options are:

-m   Assumes that *filename* is a module in memory

-v   Does not verify module CRC

-x   6Assumes that *filename* is in execution directory

-s   Displays on a single line module information including edition byte (first byte after module name); type/language byte; module CRC; "." if CRC verifies, "?" if it doesn't, a blank space if you use the "-v" option; and module name.

Examples:

```
ident -m ident (ENTER)

Header for:    Ident
Module size:   $06CE        #1742
Module CRC:    $6114F4      (Good)
Hdr parity:    $E0
Exec, off:     $0235        #565
Data size:     $099C        #2460
Edition:       $06          #6
```

```
        Ty/La At/Rv    $11 $81
        Prog mod, 6809 obj, re-en
```

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.

```
     ident /D0/OS9boot -s ⟨ENTER⟩
     2  $E1   $524CEB  ,  CCDisk
    82  $F1   $EDF046  ,  D0
    82  $F1   $69933D  ,  D1
    82  $F1   $6536D3  ,  D2
    82  $F1   $E155A8  ,  D3
     3  $E1   $0A6A0A  ,  CCIO
    83  $F1   $3EDF55  ,  TERM
     4  $C1   $BD0579  ,  IOMan
     6  $D1   $C06CB6  ,  RBF
     7  $D1   $04D9E6  ,  SCF
     5  $C1   $1795D0  ,  SysGo
     2  $C1   $7255DB  ,  Clock
    20  $11   $59ECC8  ,  Shell
     2  $E1   $316E57  ,  RS232
    83  $F1   $7BF6CE  ,  T1
     1  $E1   $316E57  ,  PRINTER
    83  $F1   $8080DF  ,  P
     3  $D1   $5F72A5  ,  PipeMan
     2  $E1   $5B2B56  ,  Piper
    80  $F1   $CC06AF  ,  Pipe
```

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, 1 = edition byte (first byte after name); $C0 = type/language byte; $A366DC = CRC value; . = OK CRC check; and OS9p2 = module name.

KILL

KILL *procID*

Aborts the process specified by its process ID number.

The process to be aborted must have the same user ID as the user executing the command. (Use the Procs command to obtain the process ID numbers.)

If a process is waiting for I/O, it may not die until it completes the current I/O operation. Therefore, if you Kill a process and the Procs command shows it still exists, it's probably waiting to receive a line of data from a terminal before it can die.

Since this is a built-in shell command, it doesn't appear in the CMDS directory.

Examples:

```
Kill 5 (ENTER)
```

kills the process with the ID number 5

```
Procs (ENTER)

User #  ID  Pty    state   Mem  Primary module

   0    2    0    active    2   Shell
   0    1    0    waiting   1   Sysgo
   0    3    0   sleeping  20   Copy

Kill 3 (ENTER)
Procs (ENTER)

User #  ID  Pty    state   Mem  Primary module

   0    2    0    active    2   Shell
   0    1    0    waiting   1   Sysgo
```

In this example, the user employs the Procs command to determine the ID number of the process to be killed, and finds that the number is 3. The Kill command kills the process. Then the user again employs Procs, this time to check whether the targeted process has died. Since it doesn't appear in the output, the user knows the process has been killed.

LINK

LINK *memory module name*

"Locks" a previously loaded module into memory.

The link count of the module specified is incremented by one each time it is "linked." Use the Unlink command to "unlock" the module when you no longer need it. You must use the Load command prior to using Link. Modules that are not Linked in memory will not be included in the "Cobbler" OS9Boot file, if you use the Cobbler command.

Examples:

```
link edit (ENTER)
```

locks the edit module into memory.

LIST

LIST *filename* [...]

Lists the contents of a text file.

This command copies text lines from the filename to the standard output path. The program terminates upon reaching the end-of-file of the last input path. If more than one filename is specified, the first file will be copied to standard output, the second file will be copied next, and so forth.

This command is most commonly used to examine or print text files.

Examples:

```
list /DØ/startup >/P & (ENTER)
```

Lists the contents of the Startup file, with output directed to the printer. The ampersand tells OS-9 to make the printing job a concurrently executed task.

```
list /D1/USER5/document /D0/myfile
/D0/BOB/text(ENTER)
```

Lists the contents of three files.

```
list /TERM >/P (ENTER)
```

Copies what you type at the keyboard to the printer. To go
back to the standard output path — the video display — press
(CLEAR) and (BREAK) simultaneously.

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

list animals (ENTER)
cat
cow
dog
elephant
bird
fish
```

Here the user employs Build to create a file called animals,
and enters six items into it. The List command, with the file-
name animals as a parameter, displays the contents of the
new file.

LOAD

LOAD *pathname*

Loads modules from file into memory.

The path specified is opened and one or more modules is read
from it and loaded into memory. The names of the modules
are added to the module directory. If a module is loaded that

has the same name and type as a module already in memory, the module with the highest revision level is kept.

Example:

```
mdir (ENTER)

    Module Directory at 13:36:47
    OS9             OS9p2           INIT
    Boot            CCDisk          D0
    D1              D2              D3
    CCIO            TERM            IOMan
    RBF             SCF             SysGo
    Clock           Shell           RS232
    T1              PRINTER         P
    PipeMan         Piper           Pipe
    Mdir

load edit (ENTER)
mdir (ENTER)

    Module Directory at 13:37:14
    OS9             OS9p2           INIT
    Boot            CCDisk          D0
    D1              D2              D3
    CCIO            TERM            IOMan
    RBF             SCF             SysGo
    Clock           Shell           RS232
    T1              PRINTER         P
    PipeMan         Piper           Pipe
    Mdir            Edit
```

First, the Mdir command displays the names of modules currently resident in memory. Then the Load command loads the Edit module into memory. Mdir again lists the memory modules, this time showing that Edit has successfully been added to memory.

LOGIN

LOGIN

Provides login security on timesharing systems.

Login is automatically called by the timesharing monitor Tsmon, and can also be used after initial log-in to change a terminal's user.

Login requests a user name and password, which it checks against a validation file. If the information is correct, the user's system priority, user ID, and working directories are set up according to information stored in the file, and the initial program — usually shell — specified in the password file is executed. If the user can't supply a correct user name and password after three attempts, the process is aborted.

The validation file is /D0/SYS/password. The file contains one or more variable-length text records, one for each user name. Each record has the following fields, delimited by commas:

1. User name (up to 32 characters; may include spaces). If this field is empty, any name will match.

2. Password (up to 32 characters; may include spaces). If this field is omitted, no password is required by the user whose record this is.

3. User index (ID) number (from 0 to 65535; 0 is superuser). This number is used by the file security system, and as the system-wide user ID, to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user.

4. Initial process (CPU time) priority: 1-255.

5. Pathlist of initial execution directory (usually /D0/ CMDS).

6. Pathlist of initial data directory (the specific user's directory).

7. Name of initial program to execute (usually Shell). Don't use shell command lines, such as Dir or Dcheck, as initial program names.

Here's a sample validation file:
.,0,128,/D0/CMDS,.,SHELL
USER1,,1,128,....SHELL
USER2,,2,128,....SHELL
USER3,,3,128,....SHELL
USER4,,4,128,....SHELL

To use the Login command, enter:

login (ENTER)

This will cause prompts for the user's name and (optionally) password to be displayed. If they're answered correctly, the user is logged into the system. Login initializes the user number, working execution directory, and working data directory, and executes the initial program specified by the Password file. It also displays the date, time and process number (which is *not* the same as the user ID).

If the shell from which Login was called will not be needed again, you can discard it by using the Ex command to start the Login command. For example:

ex login (ENTER)

To edit Password and add users to the system, use the OS-9 text editor.

Logging Off the system

To log off the system, terminates the initial program specified in the password file. For most programs (including shell) this involves typing an end-of-file character as the first character on a line. (CLEAR) and (BREAK), pressed simultaneously, signal end-of-file and log you off.

Displaying a "Message-of-the-Day"

A file named Motd, in the SYS directory, will cause Login to display its contents on the user's terminal after successful login. (This file isn't necessary for Login to operate.)

Example:

```
login (ENTER)
```

OS-9 Level 1 Timesharing System Verison 1.2 83/12/04
13:02:22

```
User name?: superuser (ENTER)
Password: secret (ENTER)

Process #07 logged 83/12/04
   13:03:00
Welcome!
```

To edit Motd, use the OS-9 Text Editor.

MAKDIR

MAKDIR *pathname* or *directory name*

Creates a new directory according to the pathname given. The pathname must refer to a parent directory for which the user has write permission.

The new directory is initialized and at first does not contain files except for the "." and ".." pointers to its parent directory and itself, respectively. All access permissions are enabled (except sharable).

It's customary, but not mandatory, to capitalize directory names.

Examples:

```
makdir /D1/STEVE/PROJECT (ENTER)
```

creates a directory by using its full pathname from the root.

```
makdir DATAFILES (ENTER)
```

creates a directory called DATAFILES within the current working directory.

```
makdir ../SAVEFILES (ENTER)
```

creates a directory called /SAVEFILES in the parent directory.

MDIR

MDIR [e]

Displays the present module names in the system module directory, that is all modules currently resident in memory.

If you use the e option, you'll see a full listing of the physical address, size, type, revision level, reentrant attribute, user count, and name of each module. All numbers shown are in hexadecimal.

Examples:

```
mdir (ENTER)

        Module Directory at 14:44:35

        OS9         OS9p2       INIT
        Boot        CCDisk      D0
        D1          D2          D3
        CCIO        TERM        IOMan
        RBF         SCF         Sysgo
        Clock       Shell       RS232
        T1          PRINTER     P
        PipeMan     Piper       Pipe
        Mdir
```

```
mdir e (ENTER)

        Module Directory at 10:55:04

        ADDR SIZE TY RV AT UC NAME_
        C305   2F F1  1 r       D0
        F059  7E8 C1  1 r       OS9
        F852  4F4 C1  1 r       OS9P2
        FD46   2E C0  1 r       INIT
        C363  798 E1 ·1 r    2  CCIO
        CAFB   38 F1  1 r    2  TERM
```

This is a partial listing of all the attributes of the modules in memory after executing the above command.

Warning: Many of the modules by Mdir are OS-9 system modules and executable as programs. Always check the module type code before running a module if you aren't familiar with it!

MERGE

MERGE [*filename*] [...]

Copies multiple input files specified by the parameter(s) to the standard output path.

Merge is commonly used to combine several files into a single output file. Data is copied in the order the filenames are given. Merge does no output line editing (such as automatic line feed). The standard output is generally redirected to a file or device.

Merge can be used to append or copy any type or mixture, of files to another device.

Examples:

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

merges the four files specified into a new file called com-
bined.file and sends the results directly to the new file, in-
stead of to the video display.

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

merges the two files specified and sends the output to the
printer.

MFREE

MFREE

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

Displays the address and size of each free memory block. The
size is given as the number of 256-byte pages. This informa-
tion is useful to detect and correct memory fragmentation.

Example:

```
mfree (ENTER)

_Address_   Pages
 E00-B1FF    164
 B400-B4FF     1

Total pages free = 165
Graphics Memory Not Allocated
```

OS9GEN

OS9Gen *devname*

Creates and links the OS9Boot file required on any disk from
which OS-9 is to be bootstrapped.

OS9Gen is used to add modules to an existing boot, or to create an entirely new boot file. (If you want an exact copy of the existing OS9Boot file, use the Cobbler command instead).

The name of the device on which the OS9Boot file is to be installed is passed to OS9Gen as a command line parameter. OS9Gen then creates a working file called tempboot on the device specified. Next it reads file names (pathnames) from its standard input, one pathname per line. Every file named is opened and copied to tempboot. This is repeated until an end-of-file marker or a blank line is reached on OS9Gen's standard input. All boot files must contain the OS-9 component modules listed in section 5.1.

After all input files are copied to tempboot, the old OS9Boot file, if present, is deleted. Tempboot is then renamed OS9Boot, and its starting address and size are linked in the disk's Identification Sector (LSN 0) for use by the OS-9 bootstrap firmware.

> **Note:** Any OS9Boot file must be stored in physically contiguous sectors. Therefore, OS9Gen is normally used on a freshly formatted disk. If the OS9Boot file is fragmented, OS9Gen prints a warning message indicating that the disk can't be used to bootstrap OS-9.

The list of file names given to OS9Gen can be entered from a keyboard, or OS9Gen's standard input can be redirected to a text file containing a list of file names. If you enter names manually, no prompts are given, and you enter the end-of-file marker (usually (CLEAR) (BREAK) or a blank line) after the line containing the last filename.

Examples:

To manually install a boot file on device /D1 which is an exact copy of the OS9Boot file on device /D0.

```
OS9Gen /D1 (ENTER)
/D0/OS9Boot (ENTER)
(CLEAR) (BREAK)
```

The first command line runs OS9Gen. The second enters the name of the file to be installed, and the third enters an end-of-file marker.

To manually install a boot file on device /D1 which is a copy of the OS9Boot file on device /D0 with the addition of modules stored in the files /D0/tape.driver and /D2/video.driver:

```
OS9Gen /D1 (ENTER)
/D0/OS9Boot (ENTER)
/D0/tape.driver (ENTER)
/D2/video.driver (ENTER)
(CLEAR) (BREAK)
```

The first command line runs OS9Gen. The second enters the main boot file name. The third and fourth enter the names of the two files to be added, and the fifth enters and end-of-file marker.

To do exactly what the previous example does, but to do it automatically by redirecting the standard input for OS9Gen:

```
build /D0/bootlist (ENTER)
? /D0/OS9Boot (ENTER)
? /D0/tape.driver (ENTER)
? /D2/video.driver (ENTER)
? (ENTER)
OS9Gen /D1 </D0/bootlist (ENTER)
```

The first command line uses Build to create a file called Bootlist. The next three lines enter the names of the three files within Bootlist. The fifth line terminates Build, and the sixth and final line runs OS9Gen with input redirected from the new Bootlist file.

PRINTERR

PRINTERR

Prints full-text error messages.

This command replaces OS-9 error printing routine (F$Perr), which prints only error code numbers, with a routine that

reads and displays textual error messages from the file /D0/ SYS/errmsg. Printerr's effect is system-wide.

A standard error message file is supplied with OS-9. The user or editor can replace or modify this file, which is a normal text file with variable-length lines.

Each error message line begins with the error number code (in ASCII characters), a delimiter, and the error message text. The error messages need not be in any particular order. Delimiters are spaces or any character numerically lower than $20. Any line with delimiter as its first character is considered to be a continuation of the previous line(s); this permits multi-line error messages.

Warning: Once the Printerr command has been used, it can not be undone. Once installed, DO NOT unlink the Printerr module. Printerr uses the current user's stack for an I/O buffer, so users are encouraged to reserve reasonably large stacks. *The only way to effectively Unlink Printerr is to reboot or reset the machine using OS-9.*

Example:

        printerr (ENTER)

        Note: The errmsg file must be on /D0.

PROCS

PROCS [e]

Displays a list of processes running on the system.

Normally lists only processes having the user's ID. If the e option is given, Procs lists processes of all users. The display is a "snapshot" taken at the instant the command is executed: processes can 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.

101

Example:

```
procs e (ENTER)

User  #  Id  Pty  state    Mem  Primary module
      0  2   0    active    2   Shell
      0  1   0    waiting   1   SysGo
      1  3   1    waiting   2   Tsmon
      1  4   1    waiting   4   Shell
      1  5   1    active   64   BASIC09
```

PWD
PXD

PWD
PXD

Pwd shows the path from the root directory to the current data directory. Pxd shows the path to the current execution directory.

OS-9 programs use both commands to track the actual physical location of files. People use it when they get "lost" in the file system. Both commands, show a path "home".

Examples:

```
chd /D1/STEVE/TEXTFILES/MANUALS
    (ENTER)
```

using a full pathname, Chd changes the user's current data directory, so the user is now in the MANUALS directory.

```
pwd (ENTER)
/D1/STEVE/TEXTFILES/MANUALS
```

Pwd shows the full path to the working directory.

```
chd .. (ENTER)
pwd (ENTER)
/D1/STEVE/TEXTFILES
```

The user "backs up" one level in the directory hierarchy, and then asks what the working directory is. Pwd shows that it is now TEXTFILES.

```
chd ++ (ENTER)
pwd (ENTER)
/D1/STEVE
```

The user again backs up, and sees, with Pwd, that the working directory is now STEVE.

```
pxd (ENTER)
/D0/CMDS
```

The user sees that the current execution directory is CMDS.

RENAME

RENAME *filename new filename*

Gives the file or directory specified in the pathlist a new name.

The user must have write permission for the file in order to change its name. It's not possible to change the names of devices, "." or "..".

Examples:

```
rename blue purple (ENTER)
```

gives the new name purple to the file formerly called blue.

```
rename /D3/user9/test temp (ENTER)
```

gives the new name temp to the file formerly called /test.

```
dir (ENTER)

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

```
rename animals cars (ENTER)
dir (ENTER)
```

```
Directory of , 16:23:22
myfile      cars
```

In this sequence, the user employs Dir to see the name of files in the current data directory. Then Renames changes name of file called animals to the new name cars. Another Dir command shows that the name has been changed.

SAVE

SAVE *filename modname* [...]

Creates a new file and writes a copy of the memory module(s) specified onto the file.

The module name(s) must exist in the module directory when saved. Save gives the new file access permissions for all modes except public write.

> **Note:** Save's default directory is the current data directory. Executable modules should generally be saved in the default execution directory.

Examples:

```
save D0/CMDS/workcount wcount
```

saves the wcount module into the newly created file called /workcount in the /D0/CMDS.

```
save /D1/math_pack add sub mul div
```

saves four modules (add, sub, mul and div) into the new file called /D1/math _ pack

SETIME

SETIME [yy/mm/dd/hh:mm:ss]

Sets the system date and time, then activates the real time clock.

The date and time can be entered as parameters. If no parameters are given, Setime will issue a prompt. Numbers are one- or two-decimal digits using space, colon, semicolon or slash delimiters. OS-9 system time uses the 24-hour clock, on which, for instance, 1520 is 3:20 P.M.

> **Important Note:** This command must be executed before OS-9 can perform multitasking operations.

Examples:

    setime 83,12,15,1545 (ENTER)

sets the date and time to December 15, 1983, 3:45 P.M.

    setime 83/12/15 15/45/00 (ENTER)

sets the same date in a slightly different, but equally acceptable format.

SETPR

SETPR *procID number*

Changes the CPU priority of a process.

The process priority number is a decimal number in the range 0 — the lowest — to 255. If you need information about the process ID number and current priority, use the Procs command.

You can use Setpr only on processes which have your ID number on them.

> **Note:** This command does not appear in the Cmds directory because it is built into the shell.

Examples:

    setpr 8 250 (ENTER)

sets or changes process #8 to priority 250

    procs (ENTER)

```
User__#  _ID pty state__  Mem Primary_module
      0   3    0 waiting  2    Shell <TERM
      0   2    0 waiting  1    Shell <TERM
      0   1    0 waiting  1    Sysgo <TERM

setpr 3 128 (ENTER)
procs (ENTER)

User__#  _ID pty state__  Mem Primary_module
      0   3  128 active   2    Shell <TERM
      0   2    0 waiting  2    Shell <TERM
      0   1    0 waiting  1    Sysgo <TERM
```

The Procs command displays process ID numbers and other information. The next command — Setpr 3 128 — sets process #3 to a priority of 128. The final command checks to make sure the change has been made.

SHELL

SHELL *arglist*

The shell is OS-9's command interpreter program. It reads data from its standard input path (the keyboard or a file), and interprets the data as a sequence of commands. The basic function of the shell is to initiate and control execution of other OS-9 programs.

The shell reads and interprets one text line at a time from the standard input path. After interpretation of each line, it reads another until it reaches an end-of-file marker at which time it terminates itself. A special case occurs when the shell is called from another program. In that case, it 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 single command line is processed.

The rest of this description is a technical specification of the shell syntax. Use of the shell is described fully earlier in this manual.

Shell Input Line Formal Syntax:

```
<pgm line> := <pgm> {<pgm>}
<pgm> := [<params>] [<name> [<modif>]
[pgm params>] [<modif>]
{<sep>}
```

Program Specifications:

```
<name>   := <module name>
         := <pathname>
         := ( <pgm list> )
```

Parameters:

```
<params> := <param> { <delim> <param> }
<delim>  := space or comma characters
<param>  := ex <name> [<modif>] chain to program
                                specified
         := chd <pathlist>      change working
                                directory
         := kill <procID>       send abort signal to
                                process
         := setpr<procID> <pty> change process
                                priority
         := chx <pathname>      change execution
                                directory
         := w                   wait for any process
                                to die
         := p                   turn OS9: prompting
                                on
         := -p                  turn prompting off
         := t                   echo input lines to
                                std output
         := -t                  don't echo input
                                lines
         := -x                  don't abort on error
         := x                   abort on error
         := * <text>            comment line: not
                                processed
```

`<sep>`	:= ;		sequential execution separator
	:= &		concurrent execution separator
	:= !		pipeline separator
	:= `<cr>`		end-of-line (sequential execution separator)

Modifiers:

`<modif>` := `<mod>` { `<delim>` `<mod>` }

`<mod>`	:= < `<pathname>`	redirect standard input
	:= > `<pathname>`	redirect standard output
	:= >> `<pathname>`	redirect standard error output
	:= # `<integer>`	set process memory size in pages
	:= # `<integer>` K	set program memory size in 1K increments

SLEEP

SLEEP *tick count*

Puts the process to "sleep" for a number of clock ticks.

Tick count may be any number 1 through 65535. If any number larger than 65535 is given for tick count, the number will be reduced by mod 65535. For example, 65536 would be reduced to 0; as would all multiples of 65536. A tick count of 95000 would be reduced to an actual tick count of 29464.

Sleep is generally used to generate time delays or to "break-up" CPU-intensive jobs. The duration of a tick is 16.66 milliseconds.

A tick count of 1 causes the process to "give up" its current time slice. A tick count of zero causes the process to sleep indefinitely (the process is usually awakened by a signal).

Example:

```
sleep 25 (ENTER)
```

puts the process "to sleep" for 25 ticks...416.50 milliseconds.

```
list startup SYS/motd nothing &
   sleep 0
&.004
setime </TERM

WELCOME TO COLOR COMPUTER OS-9

-004
ERROR #216

OS9:
```

The. List command starts running as a child process invoked from shell, and is run as background task. The Sleep command then puts shell to sleep indefinitely. When List eventually encounters the file nothing, which doesn't exist, it terminates, and sends a signal (the error status), which wakes up shell.

It's important to note, that if the error hadn't occurred, shell would have slept forever. (The keyboard is not read while the shell sleeps.) The only way out, would be to re-boot.

TEE

TEE *pathname* or *devname* [...]

Copies standard input to multiple outputs.

Tee is a filter that copies all text lines from its standard input path and also to any number of additional output paths whose names are given as parameters.

The example below uses a pipeline and Tee to send the output listing of the Dir command simultaneously to the terminal, the printer, and a disk file:

Examples:

```
dir e ! tee /P /D0/dir.listing
```

Here, a pipeline takes the output of the Dir e command and sends it to the terminal and Tee. Tee in turn sends the output along to the printer and to a file called /D0/dir.listing.

```
asm pgm.src l ! tee pgm.list /P
(ENTER)
```

In this example, the pipeline and Tee send the output of an assembler listing to a file (pgm.list) and to the printer.

```
ECHO WARNING SYSTEM DOWN IN
     10 MINUTES ! tee /T1 (ENTER)
```

Here, a message is broadcast to the terminal.

TMODE

TMODE [.*pathnum*] [*paramlist*] [...]

Displays or changes the operating parameters of the user's terminal.

You can specify any number of parameters from the list below, separating them by spaces or commas. If you don't specify parameters, the output will be current Tmode status.

You can also use a period and a number to specify the pathnumber to be affected. If you don't specify any, Tmode affects the standard input path.

Note: If this command is used in a shell procedure file, you must use the parameter. .pathnum to specify one of the standard output paths (0, 1, or 2) to change the terminal's operating characteristics. The change will remain in effect until the path is closed. To effect a permanent change to a device characteristic, the device descriptor must be changed.

This command can work only if a path to the file/device has already been opened. You may alter the device descriptor to set a device's initial operating parameter (see the *OS-9 Technical Information* manual).

upc	Upper-case only. Lower-case characters are automatically converted to upper-case.
-upc	Upper-case and lower case characters permitted.
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	No erase on backspace: echoes single backspace only.
bsl	Backspace over line: lines are ''deleted'' by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are ''deleted'' by printing a ''new line'' sequence (for hard-copy terminals).
echo	Input characters ''echoed'' back to terminal (default).
-echo	No echo.
lf	Auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.

pause Screen pause on: output suspended upon full screen. See **pag** parameter for definition of screen size. Resume output by typing any key.

-pause Screen pause mode off.

null = n   Set null count: number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal. Default = 0.

pag = n   Set video display page length to n (decimal) lines. Used for "pause" mode, see above.

bsp = h   Set input backspace character. Numeric value of character in hexadecimal. Default = 08.

bse = h   Set output backspace character. Numeric value of character in hexadecimal. Default = 08.

del = h   Set input delete line character. Numeric value of character in hexadecimal. Default = 18.

bell = h   Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07.

eor = h   Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D.

eof = h   Set end-of-file input character. Numeric value character in hexadecimal. Default = 1B.

type = h   ACIA initialization value: sets parity, word size, and so forth. Value in hexadecimal. Default = 00.

reprint = h   Reprint line character. Numeric value of character in hexadecimal.

dup = h   Duplicate last input line character. Numeric value of character in hexadecimal.

psc = h   Pause character. Numeric value of character in hexadecimal.

abort = h   Abort character (normally CONTROL C). Numeric value of character in hexadecimal.

quit = h   Quit character (normally CONTROL E).
             Numeric value of character in hexadecimal.

baud = d   Set baud rate for software-controllable inter-
             face. Numeric code for baud rate: 0 = 110
             1 = 300 2 = 600 3 = 1200 4 = 2400 5 = 4800
             6 = 9600 7 = 19200

Examples:

```
tmode -upc  lf  null=4 pause (ENTER)

tmode pag=24 pause bsl -echo
   bsp=8 (ENTER)
```

Note: If you use Tmode in a procedure file. It's neces-
sary to specify one of the standard output paths (.1 or
.2), since the shell's standard input path will have been
redirected to the diskette file. (Tmode can be used on
SCFMAN-type devices only.)

Example:

```
tmode .1 pag=24 (ENTER)
```

This sets line/page on standard output.

TSMON

TSMON [*devname*]

Supervises idle terminals and initiates the login sequence in
timesharing applications.

If you specify a device name, Tsmon opens standard I/O
paths for the device. When you enter a carriage return,
Tsmon automatically calls the Login command. If the login
fails because the user can't supply a valid user name or pass-
word, control returns to Tsmon.

Note: The Login command and its password file must
be present for Tsmon to work correctly (see the Login
command description).

113

Logging Off the System

Most programs will terminate when you enter an end-of-file marker ((CLEAR) (BREAK)) as the first character on a command line. This will log you off the system and return to Tsmon which will run Login again.

Examples:

```
tsmon /T18 (ENTER)
8.005
```

This will activate /T1, but must be run concurrently in order to keep /TERM active.

UNLINK

UNLINK *modname* [...]

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

OS-9 may (or may not) destroy the modules and reassign their memory depending on whether the module is in use by other processes.

It's good practice to unload modules whenever possible to make most efficient use of available memory resources. Modules that have been Loaded and Linked may have to be Unlinked twice to remove them from memory.

Warning: Never attempt to unlink a module you didn't load or link.

Examples:

```
unlink pgm1 pgm5 pgm99 (ENTER)
```

Unlinks the three modules specified.

```
mdir (ENTER)

    Module Directory at 11:26:22
    OS9         OS9P26      INIT
    Boot        CCDisk      D0
    D1          D2          D3
    CCIO        TERM        IOMan
    RBF         SCF         SysGo
    Clock       Shell       RS232
    T1          PRINTER     P
    PipeMan     Piper       Pipe
    Mdir        Edit

unlink edit (ENTER)
mdir (ENTER)

    Module Directory at 11:27:22
    OS9         OS9P2       INIT
    Boot        CCDisk      D0
    D1          D2          D3
    CCIO        TERM        IOMan
    RBF         SCF         SysGo
    Clock       Shell       RS232
    T1          PRINTER     P
    PipeMan     Piper       Pipe
    MDir
```

In this sequence, the Mdir command displays modules in
memory. The next command specifies that the edit module be
unlinked, and the output of the final command — Mdir —
shows that the unlinking has been successful: edit no longer
appears on the list.

VERIFY

VERIFY [u]

Checks whether module header parity and CRC value of one
or more modules on a file (standard input) are correct.

Module(s) are read from standard input, output is sent to standard output, and messages are sent to the standard error path.

Verify is dependent on the (<), input redirection command. If you fail to use the (<), redirection symbol, the Verify program will cause the system to lock. It is always necessary to redirect the input path. It is usually necessary to redirect the output and the error path.

If you use the u (update) option, the module(s) are copied to the standard output path with the module's header parity and CRC values replaced with verify's computed values. You see a message indicating whether the module's values match those computed by Verify. Verify, with the update option, will not set the execute flag in the file attributes. Use the Attr command to do this.

If you don't use the option, the module isn't copied to standard output. Verify simply displays a message indicating whether the module's header parity and CRC match those computed by Verify.

> **Note:** Verify does not turn on execute flag or update file. Use Attr.
>
> ```
> verify u </D0/CMDS/edit >/D0/CMDS/
>    newedit (ENTER)
> ```

because the u option is used the edit module is copied to a new module, newedit, with the header parity and CRC values replaced with verify's computed values.

> ```
> verify <EDIT   >NEWEDIT (ENTER)
> ```
>
> ```
> Module's header parity is correct.
> Calculated CRC matches module's.
> ```

The program checks the edit module, and directs program output to a file called newedit. Since the u option wasn't specified, Verify simply displays a summary message.

> ```
> verify <myprogram2 (ENTER)
> ```
>
> ```
> Module's header parity is correct.
> Calculated CRC matches module's.
> ```

Checks the myprogram2 module. Since there's no **u** in the command line, the module isn't copied to standard output. Instead, a simple message is displayed.

XMODE

XMODE *devname* [*paramlist*]

Displays or changes the initialization parameters of any SCF-type device such as the video display, printer, RS-232 port, and others.

Common uses include changing baud rates and control key definitions.

Xmode is similar to the Tmode command, but there are differences. Tmode operates only on open paths, so its effect is temporary. Xmode actually updates the device descriptor so the change persists as long as the computer is running, even if paths to the device are repeatedly opened and closed.

If Xmode is used to change parameter(s) and the Cobbler program is used to make a new system disk or re-write system tracks on the current system disk, the changed parameter is permanently reflected on the new system disk.

Xmode requires that you specify a device name. If you don't specify parameters, the present values for each parameter are displayed. You can use any number of parameters separating them by spaces or commas.

XMODE parameter names:

upc   Upper-case only. Lower-case characters are automatically converted to upper-case.

-upc   Upper-case and lower case characters permitted. (default)

bsb   Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).

-bsb	No erase on backspace: echoes single backspace only.
bsl	Backspace over line. Lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line. Lines are "deleted" by printing a "new line" sequence (for hardcopy terminals).
echo	Input characters "echoed" back to terminal (default).
-echo	No echo.
lf	Auto line feed on. Line feeds are automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.
pause	Screen pause on: Output suspended upon full screen. See **pag** parameter for definition of screen size. Resume output by typing any key.
-pause	Screen pause mode off.
null = n	Set null count. Number of null ($00) characters are transmitted after carriage returns for return delay. The number is decimal. Default = 0.
pag = n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp = h	Set input backspace character. Numeric value of character in hexadecimal. Default = 08.
bse = h	Set output backspace character. Numeric value of character in hexadecimal. Default = 08.
del = h	Set input delete line character. Numeric value of character in hexadecimal. Default = 18.

bell = h	Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07.
eor = h	Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D.
eof = h	Set end-of-file input character. Numeric value character in hexadecimal. Default = 1B.
type = h	ACIA initialization value: sets parity, word size, and so forth. Value in hexadecimal. Default = 15.
reprint = h	Reprint line character. Numeric value of character in hexadecimal.
dup = h	Duplicate last input line character. Numeric value of character in hexadecimal.
psc = h	Pause character. Numeric value of character in hexadecimal.
abort = h	Abort character (normally CONTROL C). Numeric value of character in hexadecimal.
quit = h	Quit character (normally CONTROL E). Numeric value of character in hexadecimal.
baud = d	Set baud rate for software-controllable interface. Numeric code for baud rate: 0 = 110 1 = 300 2 = 600 3 = 1200 4 = 2400 5 = 4800 6 = 9600 7 = 19200

Examples:

```
xmode /TERM -upc lf null=4 bse=1F
    pause (ENTER)

xmode /T1 pag=24 pause bsl -echo
    bsp=8 (ENTER)

xmode /P baud=3 -lf (ENTER)
```

Appendix A/Error Codes

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

OS-9 Error Codes

HEX	DEC	
$02	002	KEYBOARD INTERRUPT — The user used (BREAK) to abort a task that was currently being executed.
$03	003	KEYBOARD INTERRUPT — The user used (SHIFT) (BREAK) to cause the task to be executed as a background task with no video display, or to abort the task.
$C8	200	PATH TABLE FULL — The file can't be opened because the system path table is currently full.
$C9	201	ILLEGAL PATH NUMBER — Number too large, or for non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL
$CB	203	ILLEGAL MODE — Attempt to perform I/O function of which the device or file is incapable.
$CC	204	DEVICE TABLE FULL — Can't add another device.
$CD	205	ILLEGAL MODULE HEADER — Module not loaded because its sync code, header parity, or CRC is incorrect.
$CE	206	MODULE DIRECTORY FULL — Can't add another module.

121

$CF	207	MEMORY FULL — Not enough contiguous RAM free.
$D0	208	ILLEGAL SERVICE REQUEST — System call had an illegal code number.
$D1	209	MODULE BUSY — Non-sharable module is in use by another process.
$D2	210	BOUNDARY ERROR — Memory allocation or deallocation request not on page boundary.
$D3	211	END-OF-FILE — End-of-file encountered on read.
$D4	212	RETURNING NON-ALLOCATED MEMORY — Attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT — Device has damaged file structure.
$D6	214	NO PERMISSION — File or device attributes don't permit access requested.
$D7	215	BAD PATHNAME — Syntax error in pathlist, illegal character, for instance.
$D8	216	PATH NAME NOT FOUND — Can't find pathlist specified.
$D9	217	SEGMENT LIST FULL — File is too fragmented to be expanded further.
$DA	218	FILE ALREADY EXISTS — Filename already appears in current directory.
$DD	219	ILLEGAL BLOCK ADDRESS — Device's file structure had been damaged.
$DF	223	SUICIDE ATTEMPT — Request to return memory where your stack is located.
$E0	224	ILLEGAL PROCESS NUMBER — No such process exists.

$E2	226	NO CHILDREN — Can't wait because process has no children.
$E3	227	ILLEGAL SWI CODE — Must be 1 to 3.
$E4	228	PROCESS ABORTED — Process aborted by signal code 2.
$E5	229	PROCESS TABLE FULL — Can't fork now.
$E6	230	ILLEGAL PARAMETER AREA — High and low bounds passed in fork call are incorrect.
$E7	231	KNOWN MODULE — For internal use only.
$E8	232	INCORRECT MODULE CRC — Module has bad CRC value.
$E9	233	SIGNAL ERROR — Receiving process has previous unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE — Unable to locate module.
$EB	235	BAD NAME — Illegal name syntax.
$ED	237	RAM FULL — No free system RAM available at this time.
$EE	238	UNKNOWN PROCESS ID — Incorrect process ID number.
$EF	239	NO TASK NUMBER AVAILABLE — All task numbers in use.

Device Driver Errors

The following error codes are generated by I/O device drivers, and are somewhat hardware-dependent. Consult manufacturer's hardware manual for more details.

$F0	240	UNIT ERROR — Device unit doesn't exist.
$F1	241	SECTOR ERROR — Sector number is out of range.
$F2	242	WRITE PROTECT — Device is write-protected.
$F3	243	CRC ERROR — CRC error on read or write verify.
$F4	244	READ ERROR — Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR — Hardware error during disk write operation.
$F6	246	NOT READY — Device has "not ready" status.
$F7	247	SEEK ERROR — Physical seek to non-existent sector.
$F8	248	MEDIA FULL — Insufficient free space on media.
$F9	249	WRONG TYPE — Attempt to read incompatible media (for instance attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY — Non-sharable device is in use.
$FB	251	DISK ID CHANGE — Media was changed with files 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.

Appendix B/Display System Functions

Color Computer OS-9 lets you use the video display in alphanumeric, semigraphic, and graphic modes. There are many built-in functions to control the display. These functions are activated by various ASCII control characters. The three modes are therefore available for use by software written in any language using standard output statements (such as PRINT in BASIC). The Color Computer BASIC09 language has a Graphics Interface Module that can automatically generate these codes using BASIC09 RUN statements.

The display system has two display modes: Alphanumeric ("Alpha") mode and Graphics mode. The Alphanumeric mode also includes "semigraphic" box-graphics. The Color Computer's display system uses a separate memory area for each display mode so operations on the Alpha display do not affect the Graphics display, and vice-versa. Either display can be selected under software control. (See the Color Computer Manuals for more detailed information.)

Eight-bit characters sent to the display system are interpreted according to their numerical value, as shown in this chart:

Character Range (Hex)	Mode/Used For
00 - 0E	Alpha Mode — Cursor and screen control.
0F - 1B	Graphics Mode — Drawing and screen control.
1C - 20	Not used.
20 - 5F	Alpha Mode — Upper case characters.
60 - 7F	Alpha Mode — Lower case characters.
80 - FF	Alpha Mode — Semigraphic patterns.

The graphics and alphanumeric functions are handled by the OS-9 device driver module called CCIO.

125

Alpha Mode Display

This is the standard operational mode. It's used to display alphanumeric characters and semigraphic box graphics, and it simulates the operation of a typical computer terminal with functions for scrolling, cursor positioning, clear screen, line delete, and more.

Each 8-bit character is assumed to be an ASCII character. It is displayed if its high order bit (sign bit) is cleared. Lower case letters are displayed in reverse video. If the high order bit of the character is set, it's assumed to be a "Semigraphic 6" graphics box. See the Color Computer manuals for an explanation of semigraphics functions.

Alpha Mode Command Codes

Control Code	Name/Function
01	HOME — Return cursor to upper left hand corner of screen.
02	CURSOR XY — Move cursor to character X or line Y. The binary values minus 32 of the two characters following the control character are used as the X and Y coordinates. For example, to position the cursor at character 5 of line 10, you must give X = 37 and Y = 42.
03	ERASE LINE — Erases all characters on the cursor line.
06	CURSOR RIGHT — Move cursor right one character position.
08	CURSOR LEFT — Move cursor left one character position.
09	CURSOR UP — Move cursor up one line.
10	CURSOR DOWN — (Linefeed) move cursor down one line.

126

12	CLEAR SCREEN — Erase entire screen and home cursor.
13	RETURN — Return cursor to leftmost character of line.
14	DISPLAY ALPHA — Switch screen from graphic mode to alpha numeric mode.

Graphics Mode Display

Graphics mode is used to display high-resolution 2- or 4-color graphics. It includes commands to set color, plot and erase individual points, draw and erase lines, position the graphics cursor, and draw circles.

The "display graphics" command must be executed before any other graphics mode command is used. It causes the graphics screen to be displayed and sets a current display format and color.

The first time the "display graphics" command is given, OS-9 allocates a 6144 byte display memory. So there must be at least that much continuous free memory available. (You can use the OS-9 Mfree command to check free memory.) This memory is retained until the "end graphics" command is given, even if the program that initiated Graphics mode finishes. It's important that the "end graphics" command be used to give up the display memory when Graphics mode is no longer needed.

Graphics mode supports two basic formats: Two-color, which has 256 horizontal by 192 vertical points (G6R mode); and Four-color, which has 128-horizontal by 192 vertical points (G6C mode). Two color sets are available in either mode. Regardless of the resolution of the format selected, 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 lefthand corner of screen.

127

An invisible Graphics Cursor is used by many commands to reduce the amount of output required to generate graphics. You can explicitly set this cursor to any point by using the "set graphics cursor" command. Also, all other commands that include X, Y coordinates (such as "set point") move the graphics cursor to the specified position.

Graphics Mode Selection Codes

Code	Format
00	256 × 192 two-color graphics
01	128 × 192 four-color graphics

Color Set And Current Foreground Color Selection Codes

	Char	Two-Color Format Background	Foreground	Four-Color Format Background	Foreground
	00	Black	Black	Green	Green
Color	01	Black	Green	Green	Yellow
Set 1	02			Green	Blue
	03			Green	Red
	04	Black	Black	Buff	Buff
Color	05	Black	Buff	Buff	Cyan
Set 1	06			Buff	Magenta
	07			Buff	Orange
	08			Black	Black
Color	09			Black	Dark Green
Set 1	10			Black	Med. Green
	11			Black	Light Green
	12			Black	Black
Color	13			Black	Green
Set 1	14			Black	Red
	15			Black	Buff

Graphics Mode Control Commands

Control Code	Name/Function
15	DISPLAY GRAPHICS — Switches screen to graphics mode. This command must be given before any other graphics commands are used. The first time this command is given, a 6K byte display buffer is assigned. If 6K of contiguous memory isn't available, an error is returned. Follow this command by two characters specifying the graphics mode and current color/color set, respectively.
16	PRESET SCREEN — Presets entire screen to color code passed in next character.
17	SET COLOR — Selects foreground color (and color set) passed in next character, but doesn't change graphics mode.
18	END GRAPHICS — Disables graphics mode, returns the 6K byte graphics memory area to OS-9 for other use, and switches to alpha mode.
19	ERASE GRAPHICS — Erases all points to background color and homes graphics cursor to the desired position.
20	HOME GRAPHICS CURSOR — Moves graphics cursor to coordinates 0, 0 (lower left-hand corner).
21	SET GRAPHICS CURSOR — Moves graphics cursor to given coordinates X, Y. The binary value of two characters that immediately follow are used as the X and Y values, respectively.
22	DRAW LINE — Draws a line of the current foreground color from the current graphics cursor position to the given X, Y coordinates. The binary value of the two characters that immediately

follow are used as the X and Y values, respectively. The graphics cursor is moved to the end point of the line.

23 ERASE LINE — Same as "draw line" except that the line is "drawn" in the current background color, thus erasing the line.

24 SET POINT — Sets the pixel at point X, Y to the current foreground color. The binary values of the two characters that immediately follow are used as the X and Y values, respectively. The graphics cursor is moved to the point set.

25 ERASE POINT — Same as "draw point" except the point is "drawn" in the current background color, thus erasing the point.

26 DRAW CIRCLE — Draws a circle of the current foreground color with its center at the current graphics cursor position using a radius R which is obtained using the binary value of the next character. The graphics cursor position is not affected by this command.

Get Status Commands

The Color Computer I/O driver includes OS-9 "get status" commands that return the display status and joystick values, respectively. These are accessible via the BASIC09 Graphics Interface Module, or by the assembly language system calls listed below:

Get Display Status:

Calling Format: lda #1 (path number)
      ldb #SS.DStat (Getstat code $12)
      os9 I$GSTT Call OS-9

Passed: nothing

Returns: X = address of graphics display memory
                Y = graphics cursor address X = MSB
                  y = LSB
                A = color code of pixel at cursor address

Get Joystick Values:

Calling Format: lda #1   (path number)
                ldb $SS.Joy   (Getstat code $13)
                os9 I$GSTT   call OS-9

Passed: X = 0 for right joystick; 1 for left joystick

Returns: X = selected joystick x value (0-63)
                Y = selected joystick y value (0-63)
                A = $FF if fire button on; $00 if off

Display Control Codes Condensed Summary

1st Byte	2nd Byte	3rd Byte	Function
00			Null
01			Home alpha cursor
02	Column + 32	Row + 32	Position alpha cursor
03			Erase line
06			Cursor right
08			Cursor left
09			Cursor up
10			Cursor down
12			Clear screen
13			Carriage return
14			Select alpha mode
15	Mode	Color Code	Select graphics mode

16	Color Code		Preset screen
17	Color Code		Select color
18			Quit graphics mode
19			Erase screen
20			Home graphics cursor
21	X Coord	Y Coord	Move graphics cursor
22	X Coord	Y Coord	Draw line to X/Y
23	X Coord	Y Coord	Erase line to X/Y
24	X Coord	Y Coord	Set point at X/Y
25	X Coord	Y Coord	Clear point at X/Y
26	Radius		Draw circle

Appendix C/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 50	p 70	DLE 10
1 31	! 21	| 7C	A 41	a 61	SOH 01	Q 51	q 71	DC1 11
2 32	" 22	00	B 42	b 62	STX 02	R 52	r 72	DC2 12
3 33	# 23	~ 7E	C 43	c 63	ETX 03	S 53	s 73	DC3 13
4 34	$ 24	00	D 44	d 64	EOT 04	T 54	t 74	DC4 14
5 35	% 25	00	E 45	e 65	EMD 05	U 55	u 75	NAK 15
6 36	& 26	00	F 46	f 66	ACK 06	V 56	v 76	SYN 16
7 37	' 27	^ 5E	G 47	g 67	BEL 07	W 57	w 77	ETB 17
8 38	(28	[5B	H 48	h 68	BSP 08	X 58	x 78	CAN 18
9 39	) 29	] 5D	I 49	i 69	HT 09	Y 59	y 79	EM 19
: 3A	* 2A	00	J 4A	j 6A	LF 0A	Z 5A	z 7A	SUM 1A
; 3B	+ 2B	00	K 4B	k 6B	VT 0B			
, 2C	< 3C	{ 7B	L 4C	l 6C	FF 0C			
− 2D	= 3D	5F	M 4D	m 6D	DR 0D			
. 2E	> 3E	} 7D	N 4E	n 6E	CO 0E			
/ 2F	? 3F	\ 5C	O 4F	o 6F	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
V	0A	1A	12
∧	0C	1C	13

Appendix D/Keyboard Control Functions

Key Definitions for Special Functions and Characters

Key Combination	Control Function or Character
(CLEAR)	Used a control key (CTRL).
(BREAK)	Same as (CLEAR) (E).
(CLEAR) (—)	Generates an underline (_) character. The underline character is displayed as a left arrow (←).
(CLEAR) (,)	Generates a left brace ({) character. The left brace character is displayed as a left bracket ([) in reverse video.
(CLEAR) (.)	Generates a right brace (}) character. The right brace character is displayed as a right bracket (]) in reverse video.
(CLEAR) (#)	Generates a tilde (~) character. The tilde is displayed as a hyphen (-) in reverse video.
(CLEAR) (/)	Generates a reverse slash (\) character.
(CLEAR) (BREAK)	Generates an end-of-file (EOF). Same as (ESC) on a standard terminal.
(←)	Backspace key or CONTROL H.
(SHIFT) (←)	Deletes the entire current line. Same as CONTROL X.
(SHIFT) (BREAK)	Interrupts the video display of a running program. It reactivates the shell and then runs the program as a background task. Same as CONTROL C.
(CLEAR) (0)	Upper/lower case shift lock function.

135

(CLEAR) (1)	Generates a vertical bar (|) character. The vertical bar character is displayed as an exclamation (!) mark in reverse video.
(CLEAR) (7)	Generates an up arrow or caret (^) character.
(CLEAR) (8)	Generates a left bracket ([) character.
(CLEAR) (9)	Generates a right bracket (]) character.
(CLEAR) (A)	Repeats previous command line.
(CLEAR) (D)	Redisplays current command line on the video display.
(CLEAR) (E)	Stops the program currently being executed. Same as BREAK.
(CLEAR) (W)	Temporarily halts output to the screen display. Press any key to resume output.

INDEX

INDEX

INDEX

OS-9 Program Development

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

    A.   CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

    B.   CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

    A.   For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

    B.   RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

    C.   Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

    D.   Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

    E.   Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

    A.   EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B.   RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C.   No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D.   Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV.   RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A.   Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B.   Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C.   CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D.   CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E.   CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F.   CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G.   All copyright notices shall be retained on all copies of the Software.

V.   APPLICABILITY OF WARRANTY

A.   The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B.   The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI.   STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

OS-9 Program Development

OS-9 Program Development is in three parts:

I.   Text Editor
II.   Assembler
III.   Interactive Debugger

You use these tools in developing an OS-9 program. The *Text Editor* lets you create a source program file. The *Assembler* lets you translate the source file to a machine-language file. With the *Interactive Debugger* you run and test the source program.

Contents

OS-9 Macro Text Editor

1 / Introduction

Overview

The OS-9 Macro Text Editor is a powerful but simply learned text-preparation system. It is commonly used 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 following features of the editor facilitate and expedite the text-preparation task.

- Compact size
- Multiple read and write files open simultaneously
- All OS-9 commands usable inside the text editor
- Adjustable workspace size
- Repeatable command sequences
- Edit macros
- Multiple text buffers
- Powerful commands

The Macro Text Editor is an OS-9 executable module in position-independent, reentrant 6809 machine language. It is about 5K bytes long and requires at least 2K bytes of free RAM to run. You may use the editor on any OS-9 system that has disk storage.

Text Buffers

As you enter text, the editor places it in a temporary storage area called a text buffer. Text buffers may be thought of as scratch pads used for saving text that you wish to manipulate with various edit commands. The Macro Text Editor can use multiple text buffers, one at a time.

3

The buffer in use is the "primary buffer," and the previous primary buffer is the "secondary buffer." This manual refers to the primary buffer as the "edit buffer" or "buffer" for short. The secondary buffer is important only when you wish to use a command that moves text from one buffer to another.

Edit Pointers

In the Macro Text Editor an edit pointer identifies your position in the buffer. This is similar to holding your place with your finger when reading a newspaper.

Although the screen never shows the edit pointer, commands reposition it and display the text to which it points. Each buffer has its own edit pointer, which allows you to move from buffer to buffer without losing your place in any of them.

Entering Commands

The Macro Text Editor is an interactive editing system. You and the editor carry on a two-way conversation that goes through a cycle similar to the one below and continues until you type (Q) to quit editing.

1. EDIT shows E: on the screen, asking you to enter a command.

2. You enter (type) a line with one or more commands on it.

3. EDIT carries out the commands.

4. EDIT shows E: on the screen, asking you to enter a command.

When the screen shows E:, type one or more commands on a line and then type (ENTER) (always type (ENTER) to end a line). Enter each command by typing its name and any parameters (values) it needs.

4

If you enter more than one command on a line, separate the commands with a space. If a space is the first character on a line, the editor considers the space an insert command and not a separator.

Correct a typing error by backspacing or by deleting the entire line. You cannot correct a line after typing (ENTER).

A description of all control characters is in the *OS-9 Commands*. Listed below are some of them.

(CLEAR)(A)

repeats the previous input line.

(CLEAR)(C)

interrupts the editor and returns to command entry mode.

(CLEAR)(D)

displays the current input on the next line.

(CLEAR)(H) or (←)

backspaces, erases the previous character.

(Q)

interrupts the editor and returns to command entry mode.

(CLEAR)(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.

(CLEAR)(X), (SHIFT), OR (←)

deletes the line.

(CLEAR)(BREAK)

interrupts the editor and returns to command entry mode.

Command Parameters

With many commands you specify a value that represents a parameter, for example, the number of times to repeat a command or a phrase you wish to find. The two types of edit parameters are "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 that a command is to affect. If you do not specify a numeric parameter, the editor assumes you intend the default value of one. Specify all other numeric parameters in one of the following ways.

1. Enter a positive decimal integer from 0 to 65,535.
   Examples:
   0
   10
   5250
   65532
   31

2. Enter an asterisk (*) as a shorthand for 65,535. This is the editor's notation for infinity. The asterisk is used to specify all remaining lines, all characters, or repeat forever.

3. Use a numeric variable. (See Edit Macros, p. 30.)

String Parameters

String parameters specify a single character, group of characters, word, or phrase. Specify string parameters in either of the following ways.

1. Enclose the group of characters with delimiters — two matching characters. You may use any characters, but they must match. If one string immediately follows another, separate the two with a single delimiter that matches the others. Examples:

"string of characters"
/STRING/
: my name is Larry :
"first string"second string"
/string 1/ string 2/

2. Use a string variable. (See Edit Macros, p. 30.)

Syntax Notation

Syntax descriptions indicate what to enter and the order in which to do it. The command name is first; type this exactly as given. Following the command name are the parameters the command expects; 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
(SPACEBAR) = space character
$text$ = one or more characters terminated by typing (ENTER)

Below are examples of command syntaxes, how the command is used, and parameter requirements.

Syntax	Usage	Parameter Requirements
(↓)	(CLEAR)(7)	None
Vn	V 5	1 numeric
Ln	L*	1 numeric
S$nstr$	S"my string"	1 numeric and 1
	S4/hello/	string
	S*/search string,	
C$nstrstr$	C"this"that"	1 numeric and 2
	C3:this string:	strings
	that string:	
	C4/string//	

7

Getting Started

Start OS-9. When the screen shows OS-9, you are ready to enter the editor. To do so, type:

    EDIT (ENTER)

When the screen shows E:, enter a command. The first command to learn is how to quit (exit) the editor. Type (Q) followed by (ENTER).

The Q command terminates the editor and returns you to the OS-9 Shell, which responds with the OS-9: prompt. Learn a little about the other commands. Skim the first three sections on commands ("Displaying Text," "Manipulating the Edit Pointer," "Inserting and Deleting Lines").

Now enter the editor again and work through Sample Session 1. After you have mastered the first three sections of commands, move on to the more advanced commands and the other Sample Sessions.

If you work with text files, enter the editor with an initial input and/or output file. Although you may open additional files after entering the editor, several commands treat the initial files as special cases (it is assumed that these files are the main working files); therefore, it may be advisable to specify your working files when you start the editor.

Below is a list of ways in which the editor may be started, including the effect of each. A file that already exists is referred to as *oldfile*. A file to be created is referred to as *newfile*.

EDIT   OS-9 loads the editor and starts it. There is no initial read or write file. Perform text file operations by opening files after the editor is started.

EDIT *newfile*   OS-9 loads the editor and starts it. The editor creates a file called *newfile*, the initial write file. There is no initial read file; however, files may be read if they are opened after the editor is started.

8

EDIT *oldfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called SCRATCH; this is the initial write file. When the edit session is complete, *oldfile* is deleted, and SCRATCH is given the name *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*, *newfile*, and *file* refer to any properly constructed OS-9 pathlist.

2 / Edit Commands

Displaying Text

L*n*

lists (displays) the next *n* lines, starting at the current position of the edit pointer. The position of the edit pointer does not change. Examples:

    L (ENTER)

displays the current line.

If the edit pointer is not at the beginning of the line, only that part of the line from the edit pointer to the end of the line is displayed.

    L3 (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 *n* lines that precede the edit pointer. The position of the edit pointer does not change. Examples:

    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, nothing is displayed.

X3 (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

(CLEAR)(7)

moves the edit pointer to the beginning (first character) of the text buffer. The screen shows the up arrow when you hold down (CLEAR) and type (7). Example:

(CLEAR)(7)(ENTER)

moves the edit pointer to the beginning of the buffer.

/

moves the edit pointer to the end (last character) of the buffer. 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.

This command is useful for going through text one line at a time. For example, you may want to 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. Example:

   +0   (ENTER)

Entering a value other than zero moves the edit pointer forward *n* lines and displays the line. Examples:

   +   (ENTER)

moves the edit pointer to the next line and displays the line. This command performs the same function as (ENTER).

   +10   (ENTER)

moves the edit pointer forward 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 back (toward the top) *n* lines. Examples:

   -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. Examples:

   -   (ENTER)

moves the edit pointer back one line and displays it.

   -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. This command is used primarily to move the edit pointer to some position in the line other than the first character. Examples:

> (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. This command is used primarily to move the edit pointer to some position in a line other than the first character. Examples:

< (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

(SPACEBAR) *text*

inserts lines you enter from the keyboard. The lines are inserted before the current position of the edit pointer. The position of the edit pointer does not change. The first character you type is a space. Examples:

> **(SPACEBAR)** INSERT THIS LINE **(ENTER)**

inserts the line.

> **(SPACEBAR)** LINE ONE **(ENTER)**
> **(SPACEBAR)** LINE TWO **(ENTER)**
> **(SPACEBAR)** LINE THREE **(ENTER)**

inserts three lines.

I*n* *str*

Inserts a line of *n* copies of the string. The line is inserted before the position of the edit pointer, and the position of the edit pointer does not change. Example:

> I40/*/ **(ENTER)**

inserts a line containing 80 asterisks.

This is useful when outlining a portion of text. You can also use the "I" command to insert a line containing a single copy of the string. This is important when you want to use a macro to insert lines, since the **(SPACEBAR)** is not used within a macro. 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. Examples:

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. Examples:

K (ENTER)

deletes the character at the current position of 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. This is useful, for example, for adding comments to assembly language statements. This command extends the line, displays it, and then moves the edit pointer past it. Examples:

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 and moves the edit pointer past these lines.

U

unextends (deletes) the remainder of a line from the current position of the edit pointer. This command is commonly used to remove extensions, such as comments, from a line. Example:

U (ENTER)

deletes all the characters from the current position of the edit 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.

Searching and Substituting

S*n* string

searches for the next *n* occurrences of the *string*. When it finds the occurrence, it displays the line, and moves the edit pointer past it. If it does not find the string, the edit pointer does not move. Examples:

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" that are between the edit pointer and the end of the text.

17

Cn string1 string2

changes the next *n* occurrences of *string1* to *string2*. When it 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*, the edit pointer does not move. Examples:

        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" to "sought and found" that are between the edit pointer and the end of text.

An

sets the SEARCH/CHANGE anchor to Column *n*. To find a string that begins in Column 1 (such as an assembly language label) but that you don't want to find if it begins in any other column, set the anchor to Column 1 before using the search command to find it. Examples:

        A (ENTER)

finds a string only if it begins in Column 1.

        A50 (ENTER)

finds a string only if it begins in Column 50.

To return to the normal mode of searching, set the anchor to zero. You can now find a string regardless of the column in which it begins. Example:

A0 (ENTER)

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.

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. Examples:

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. Examples:

.SHELL DIR /D1 (ENTER)

calls the OS-9 Shell to print the directory D1.

.SHELL BASIC09 (ENTER)

starts BASIC09.

.SHELL EDIT *oldfile newfile* (ENTER)

starts another copy of 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 will use little of the available workspace, decrease the workspace size so that other OS-9 programs may use the memory that you free. Examples:

    M5000 (ENTER)

sets the workspace size to 5,000 bytes.

    M10000 (ENTER)

sets the workspace size to 10,000 bytes.

Before typing (Q) to quit editing, you may want to increase the workspace. This decreases the amount of time needed to copy the input file to the output file, since the editor is then able to read and write more of the file at one time. Memory is allocated in 256-byte pages; therefore, for the M command to have any effect, the desired workspace size must differ from the current size by at least 256 bytes. The M command does not let you return any part of the workspace that is being used for buffers or macros.

.SIZE

displays the size of the workspace and the amount that has been used. Example:

    .SIZE
        521   15328

521 is the amount of workspace used for buffers and macros. 15328 is the amount of memory available in the workspace.

Q

ends editing and returns to the OS-9 Shell.

If you specified files when you started the editor, the text in Buffer 1 is written out to the initial write file (the one you specified when you started EDIT). The remainder of the initial

input file (the one you specified when you started EDIT) is then copied to the initial write file. After the text is copied, the editor terminates and control returns to the OS-9 Shell.

V*mode*

turns the verify mode on or off. When you start the editor, the verify mode is on; therefore, the editor displays the results of all commands for which results can be displayed. If you do not want to see the results of commands, turn off the verify mode by specifying 0 (zero) for *mode*. Example:

> V0 (ENTER)

turns off the verify mode.

To return to the verify mode, specify any nonzero value for *mode*. Examples:

> V2 (ENTER)

turns on the verify mode.

> V13 (ENTER)

turns on the verify mode.

If the verify mode is on and you switch to a macro, it remains on. If you turn off the verify mode while in the macro, it is automatically restored when you return to the editor.

Manipulating Multiple Buffers

.DIR

displays the directory of the editor's buffers and macros, which is similar to the one below:

```
BUFFERS:
$         0 (secondary buffer)
*         1 (primary buffer)
         50 (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. Example:

     B5 (ENTER)

makes Buffer 5 the primary buffer; if Buffer 5 does not exist, it
is created.

P*n*

puts (moves) *n* lines into the secondary buffer. This command
removes the lines from the primary buffer, starting at the posi-
tion 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. Examples:

     P (ENTER)

moves one line to the secondary buffer.

     P5 (ENTER)

moves five lines to the secondary buffer.

     P* (ENTER)

moves to the secondary buffer all lines that are between the
current position of the edit pointer and the end of text.

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. It displays the lines that are moved. When used

with the P command, the G command moves text from one place to another. Examples:

> 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.

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. This command is used when editing a file that is too large to fit into the editor's workspace at one time. .NEW writes out all lines that precede the current line and then the editor tries to read in an equal amount of new text that is appended to the end of the buffer.

The NEW command always writes text to the initial output file (the one created when you started the editor) and always reads text from the initial input file (the one specified when you started the editor).

If you have finished editing the text currently in the buffer, you may "flush" out 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. 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 "flushes" out 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. Example:

      ,READ "myfile" (ENTER)

closes the current input file and opens "myfile" for reading.

You may specify an empty pathlist. Example:

      ,READ "" (ENTER)

closes the current input file and restores the initial input file (the one you specified when you started the editor) for reading.

An open file remains attached to the primary buffer until you close the file. You may 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 in which the file was opened.

.WRITE *str*

opens a new file for writing. The *string* specifies the pathlist for the file you wish to create. Example:

      ,WRITE "newfile" (ENTER)

closes the current write file and creates one called "newfile."

You may specify an empty pathlist. Example:

      ,WRITE "" (ENTER)

closes the current write file and restores the initial write file (the one you specified when you started the editor).

A new write file is attached to the primary buffer and remains attached until you close the file. You may have more than one write file open by using the .WRITE command to open them in different buffers. To write these files, switch to the proper buffer and then write that buffer's file. To close a file, you must be in the same buffer in which 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. Examples:

    R (ENTER)

reads one line from the input file.

    R 1 Ø (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. Examples:

    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.

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. After the fail flag is set, the editor will not execute any more commands until one of the following conditions is met:

1. The end of a command line is reached if it was typed in from the keyboard.

2. The end of the current loop is reached. Any loops that are more deeply nested will be skipped. (See the repeat command.)

3. A colon (:) command is encountered. Since loops that are nested deeper than the current level are skipped, any occurrences of : that are in a more deeply nested loop will also be skipped.

Below are commands that set the fail flag and the condition on which it is set:

$<$ 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, the fail flag is not set. This is because an asterisk usually means continue until there is nothing more to do, and the

commands succeed in doing just that. In addition to these commands that set the fail flag as a side effect, the following commands explicitly set the fail flag if some condition is not true.

.EOF

tests for end of file. This succeeds if there is no more text to read from the file; otherwise, it sets the fail flag.

.NEOF

tests for not end of file. This succeeds if there is text to read from the file; otherwise, it sets the fail flag.

.EOB

tests for end of buffer. This 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. This 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 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. This 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. This succeeds if n equals zero; otherwise, it sets the fail flag.

.STAR *n*

tests for star (asterisk). This succeeds if *n* equals 65,535 (``*``); otherwise, it sets the fail flag.

.STR *str*

tests for string match. This 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. This 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 the loop is entered from the keyboard, it must all be on one line. If it is part of a macro, however, it may span several command lines. Examples:

    [L] 5 (ENTER)

repeats the L command five times.

Note: This is not exactly the same as L5, which executes the L command only once and has 5 as its parameter.

```
[ + ] * (ENTER)
```

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 the + command until the end of the buffer is reached. Then when the command tries to move the edit pointer past the end of the buffer, the fail flag is set; this terminates the loop and clears the fail flag.

Whenever the end of the loop is reached and the fail flag is set, the current loop is terminated and the fail flag is cleared.

: *commands*

decides whether or not to execute the commands that follow it, depending on the state of the fail flag. Below are the actions taken as a result of the colon (:) being executed and the state of the fail flag.

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.

```
(CLEAR)(7)  [  ,NEOB  [  ,STR"A"  +  :
        D  ]  ]  *
```

The (↓) 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 *n* A at the beginning of the line. If there is one, the + command is executed; otherwise, the D command is executed.

Below is a command that searches the current line for ``find it.'' If the command finds ``find it,'' the screen shows the line.

Otherwise, the command line fails and the screen shows
* FAIL *.

```
[ .EOL V0 -0 V .F : .STR"find it"
  -0 .S : [>] ]*
```

.EOL V0 -0 V .F tests to determine if the edit pointer is at the
end of the line. If it is, the verify mode is turned off to prevent
the -0 from displaying the line. Then it is turned back on, and the
.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, the -0 .S commands are executed.
This 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 special-
ized 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 is made up of any number of
ordinary commands (except for (SPACEBAR) and (ENTER), any
command may be used 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 just as you would enter tett into
an edit buffer. When you are satisfied with the macro, close its
definition by typing (Q). This returns you to the normal edit
mode.

Macro Headers

A macro header must be the first line in each macro. It is made up of a name, which may be followed by 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. Examples:

    MACRO
    trim_spaces
    LIST
    EXTRA_LONG_MACRO_NAME

Although you may make a macro name any length, it is better to keep it short because you must spell it the same each time you use it. You may use upper- and lower-case letters or a mixture.

Parameter Passing

Like other commands, macros may be given parameters so that they are able to work with different strings and numbers of things. Macros are unable to use parameters directly; instead, the parameters are passed 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:

    #N
    #ABC
    #LONG_NUMBER_VARIABLE

Examples of string variables:

```
$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 declares the macro's name to be 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. Here both of the macro's parameters are passed to the S command, which does the actual searching.

```
SRCH #N $STR
S #N $STR
```

Below is an example of how to execute this macro:

```
.SRCH 15 ''string''
```

In the next example the order of the parameter is reversed. Therefore, when executing the macro, use the reverse order. Below is the macro definition:

```
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. Below is an example of how to execute this macro. The order of the parameters directly corresponds to the order of the variables in the variable list.

```
.SRCH ''string'' 15
```

! *text*

places comments inside a macro. Ignore the remainder of the line following the ! command. This allows you to include, as

part of a macro, a short description of what it does in case you forget or in case someone else wants to use the macro. Examples:

```
        !
  (↑) !   Move the edit pointer to the top of the buffer.
  L*  !   Display all lines of text.
        !
```

In the example above 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 (.). Examples:

```
    .MYMACRO
    .LIST 0
    .TRIM " "
    .MERGE " FILE_A " FILE B B"
```

.MAC str

creates a new macro or opens the definition of an existing one so that it may be edited. To create a new macro, specify an empty string. 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. Example:

> .MAC "MYMACRO" (ENTER)

opens the macro "MYMACRO" for editing.

When a macro is open, edit it or enter its definition with the same comands you use in a text buffer. After you edit the macro, type (Q) to close its definition and return to the edit mode. The

first line of the macro must begin with a name that has not already been used in order to close the definition and return to EDIT.

.SAVE *str1 str2*

saves macros on an OS-9 file. *String1* specifies a list of macros to be saved; the macro names are separated by spaces. *String2* specifies the pathlist for the file on which the macros are to be saved. Examples:

    .SAVE "MYMACRO"MYFILE"

saves the macro "MYMACRO" on the file "MYFILE."

    .SAVE "MACA MACB MACC"MFILE

saves the macros "MACA," "MACB," and "MACC" on the file "MFILE."

To save more than one macro on a file, space between their names.

.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 is loaded, EDIT verifies that no other macro exists with that name. A duplicate name does not load, and EDIT skips to the next macro on the file. EDIT displays the names of all macros it loads. Examples:

    `,LOAD "MACROFILE"`

loads the macros in the file called MACROFILE.

    `,LOAD "MYFILE"` (ENTER)

loads the macros in the file called MYFILE.

34

.DEL *str*

deletes the macro specified by the string. Examples:

> `.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 at *string1* with *string2*. The *n* option permits *n* occurrences at *string1* to be changed with *string2*.

Q

ends a macro edit session. It returns you to the normal edit mode.

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 some practice in using macro commands, turn to Sample Session 5.

3 / Sample Sessions

Sample Session 1

Clear the buffer by deleting its contents.
You Type: (CLEAR)7D* (ENTER)
Screen Shows: E:(↓)

Insert three lines into the buffer. Begin each line with a space, which is the command for inserting text.
You Type: (SPACEBAR)MY FIRST LINE (ENTER)
(SPACEBAR)MY SECOND LINE (ENTER)
(SPACEBAR)MY THIRD LINE (ENTER)
Screen Shows: E: MY FIRST LINE
E: MY SECOND LINE
E: 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:** (CLEAR) 7 always shows (↓) on the screen. Typing (-*) also moves the edit pointer to the beginning of a buffer.
You Type: (CLEAR)7 (ENTER)
Screen Shows: E: (↓)

List (display) the first line you inserted into the buffer.
You Type: L (ENTER)
Screen Shows: E:L
MY FIRST LINE

Display the first two lines you inserted into the buffer.
You Type: L2 (ENTER)
Screen Shows: E:L2
MY FIRST LINE
MY SECOND LINE

Move to the next line and display it.
You Type: (ENTER)
Screen Shows: E:
MY SECOND LINE

Move to the next line and display it.
You Type: (ENTER)
Screen Shows: E:
MY THIRD LINE

Display text beginning at the position of the edit pointer. This is the function of L.
    You Type: L (ENTER)
    Screen Shows: E:L
                        MY THIRD LINE

Insert a line into the buffer. **Note:** In the next sample you will see that the line is inserted before the current position of the edit pointer.
    You Type: (SPACEBAR) INSERT A LINE (ENTER)
    Screen Shows: E: INSERT A LINE

The following command line consists of more than one command. (CLEAR)7 (↑) 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: (CLEAR)7L* (ENTER)
    Screen Shows: E:(↑)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: E:L
                        MY FIRST LINE

Move the edit pointer forward two lines and display the lines.
    You Type: +2 (ENTER)
    Screen Shows: E:+2
                        INSERT A LINE

Display all lines from the edit pointer to the end of the buffer.
    You Type: L* (ENTER)
    Screen Shows: E:L*
                        INSERT A LINE
                        MY THIRD LINE

Move the edit pointer to the end of the buffer.
    You Type: / (ENTER)
    Screen Shows: E:/

Show that the edit pointer is at the end of text. It is, since the screen shows no lines.

You Type: L* (ENTER)
Screen Shows: E:L*

Insert two more lines.
You Type: (SPACEBAR)FIFTH LINE (ENTER)
            (SPACEBAR)LAST LINE (ENTER)
Screen Shows: E: FIFTH LINE
               E: LAST LINE

Move the edit pointer back one line and display the line.
You Type: -2 (ENTER)
Screen Shows: E:-2
                 FIFTH LINE

Move the edit pointer back two lines and display the line.
You Type: -3 (ENTER)
Screen Shows: E:-3
                 MY SECOND LINE

Move the edit pointer three characters to the right and display the remainder of the line. **Note:** Space between commands.
You Type: >3 L (ENTER)
Screen Shows: E:>3 L
                 SECOND LINE

Display the characters that precede the edit pointer on the current line.
You Type: X (ENTER)
Screen Shows: E:X
                 MY

Move the edit pointer to the end of the current line.
You Type: +0 (ENTER)
Screen Shows: E:+0

Show that the edit pointer is at the end of the line. It is, since the screen shows no lines.
You Type: L (ENTER)
Screen Shows: E:L

Display the characters that precede the edit pointer on the current line.
You Type: X (ENTER)
Screen Shows: E:X
                 MY SECOND LINE

39

Move the edit pointer back to the beginning of the current line.
    You Type: -∅ (ENTER)
    Screen Shows: E:-∅
                    MY SECOND LINE

Show that the edit pointer is at the beginning of the line. It is, since the screen shows no lines.
    You Type: X (ENTER)
    Screen Shows: E:X

Go to the beginning of the text.
    You Type: (CLEAR)7 (ENTER)
    Screen Shows: E: (↑)

Insert a line of 14 asterisks.
    You Type: I14"*" (ENTER)
    Screen Shows: E:I14"*"
                ****

Insert an empty line.
    You Type: I"" (ENTER)
    Screen Shows: E: I""

Move to the top of the text and display all lines in the buffer.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: E:(↑)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: E:+2
                    MY FIRST LINE

Extend the line with XXX.
    You Type: E" XXX" (ENTER)
    Screen Shows: E: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: E : L
                   MY SECOND LINE

Extend three lines with YYY.

    You Type: E3" (SPACEBAR) YYY" (ENTER)
    Screen Shows: E : E3" YYY"
                   MY SECOND LINE YYY
                   INSERT A LINE YYY
                   MY THIRD LINE YYY

Move back 2 lines.

    You Type: -2 (ENTER)
    Screen Shows: E : -2
                   INSERT A LINE YYY

Move the edit pointer to the end of the line; move the edit pointer back four characters; display the current line, starting at the edit pointer.

    You Type: +∅ <4 L (ENTER)
    Screen Shows: E : +∅ <4 L
                   YYY

Truncate the line at the current position of the edit pointer. This removes the YYY extension.

    You Type: U (ENTER)
    Screen Shows: E : U
                   INSERT A LINE

Go to the top of the text and display the contents of the buffer.
You Type: (CLEAR) 7L* (ENTER)
Screen Shows: E : (↑)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: E:D2
                 ****

Move the edit pointer forward two lines.
You Type: +2 (ENTER)
Screen Shows: E:+2
                         INSERT A LINE

Delete this line.
You Type: D (ENTER)
Screen Shows: E:D
                         INSERT A LINE

Display the current line.
You Type: L (ENTER)
Screen Shows: E: MY THIRD LINE YYY

Move the edit pointer to the right three characters and display
the text.
You Type: >3 L (ENTER)
Screen Shows: E:>3 L
                         THIRD LINE YYY

Kill (delete) the 11 characters that constitute THIRD LINE.
You Type: K11 (ENTER)
Screen Shows: E:K11
                         THIRD LINE

Go to the beginning of the line and display it.
You Type: -0 (ENTER)
Screen Shows: E:-0
                         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: +0 K -0 (ENTER)
Screen Shows: E: +0 K -0
                         MY YYYFIFTH LINE

Separate the two lines by inserting an end-of-line character.
You Type: >6 I// (ENTER)
Screen Shows: E:>6 I//
                           MY YYY

Note: The empty line is inserted before the current position of the edit pointer.
You Type: L (ENTER)
Screen Shows: E:L
                           FIFTH LINE

Sample Session 2

Clear the buffer by deleting its contents.
You Type: (CLEAR)7D* (ENTER)
Screen Shows: E:(↓)D*

Insert lines.
**You
Type:** (SPACEBAR)ONE TWO THREE 1.0 (ENTER)
       (SPACEBAR)ONE (ENTER)
       (SPACEBAR)(SPACEBAR)TWO (ENTER)
       (SPACEBAR)(SPACEBAR)(SPACEBAR)
          THREE (ENTER)
       (SPACEBAR)ONE TWO THREE 2.0 (ENTER)
       (SPACEBAR)ONE (ENTER)
       (SPACEBAR)(SPACEBAR)TWO (ENTER)
       (SPACEBAR)(SPACEBAR)(SPACEBAR)
          THREE (ENTER)
       (SPACEBAR)ONE TWO THREE 3.0 (ENTER)
Screen Shows: E: ONE TWO THREE 1.0
          E: ONE
          E: TWO
          E: THREE
          E: ONE TWO THREE 2.0
          E: ONE
          E: TWO
          E: THREE
          E: ONE TWO THREE 3.0

43

Go to the top of the text and display all lines in the buffer.
You Type: (CLEAR)7L* (ENTER)
Screen Shows:0E: (↑)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. The edit pointer moves past the letter O in TWO; the entire line is found and displayed.
You Type: S"TWO" (ENTER)
Screen Shows: E:S"TWO"

```
                  ONE  TWO  THREE  1.0
```

Show that the edit pointer has moved.
You Type: L (ENTER)
Screen Shows: E:L

```
                  THREE  1.0
```

Search for all occurrences of "TWO" that are between the edit pointer and the end of the buffer. When "TWO" is found, the edit pointer moves past it and the line is displayed.
You Type: .S*/TWO/ (ENTER)
Screen Shows: E: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: (CLEAR)7 C/THREE/ONE/
            (ENTER)
Screen Shows: E:(↑) C/THREE/ONE/

```
                      ONE  TWO  ONE  1.0
```

Show that the edit pointer has moved past the *string* that was changed.

    You Type: L (ENTER)
    Screen Shows: E : L
                        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. All other occurrences are skipped.

    You Type: (CLEAR)7 A2 S*/TWO/ (ENTER)
    Screen Shows: E :(↓) 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: (CLEAR)7 A1C*/ONE/XXX/ (ENTER)
    Screen Shows: E :(↑) A1C*/ONE/XXX/
                        XXX TWO ONE 1 , 0
                        XXX
                        XXX TWO THREE 2 , 0
                        XXX
                        XXX TWO THREE 3 , 0
                          THREE
                        XXX TWO THREE 2 , 0
                        XXX
                          TWO
                          THREE
                        XXX TWO THREE 3 , 0

Go to the top of the buffer and display the text.

    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: E :(↓)L*
                        XXX TWO ONE 1 , 0
                        XXX
                          TWO

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:** < . CHANGE / ONE / XXX / (ENTER)
>   **Screen Shows:** E : C / ONE / XXX /
>   XXX TWO XXX 1 . 0

Move to the beginning of the current line.
>   **You Type:** - 0 (ENTER)
>   **Screen Shows:** E : - 0
>   XXX TWO XXX 1 . 0

Change three occurrences of "XXX" to "ZZZ."
>   **You Type:** . C3 / XXX / ZZZ / (ENTER)
>   **Screen Shows:** E : C3 / XXX / ZZZ /
>   ZZZ TWO XXX 1 . 0
>   ZZZ TWO ZZZ 1 . J0
>   ZZZ

Sample Session 3

Clear the buffer by deleting its contents:
>   **You Type:** (CLEAR) 7D * (ENTER)
>   **Screen Shows:** E : (↑)D *

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. No macros are defined. This is the initial environment when EDIT is started from OS-9.
>   **You Type:** . DIR (ENTER)
>   **Screen Shows:** E : . DIR
>
>   BUFFERS :
> $ 0
>   *   1
>
>   MACROS :

Insert some lines into Buffer 1 so that later you can identify it.
>   **You Type:** (SPACEBAR) BUFFER ONE 1 . 0 (ENTER)
>   (SPACEBAR) BUFFER ONE 2 . 0 (ENTER)
>   (SPACEBAR) BUFFER ONE 3 . 0 (ENTER)
>   (SPACEBAR) BUFFER ONE 4 . 0 (ENTER)

Screen Shows: E: BUFFER ONE 1.0
E: BUFFER ONE 2.0
E: BUFFER ONE 3.0
E: BUFFER ONE 4.0

Display the text in Buffer 1.
You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↓)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: E:B0

Display the directory of buffers and macros. **Note:** The symbols identifying the buffers are now reversed.
You Type: .DIR (ENTER)
Screen Shows: E:.DIR

BUFFERS:
$ 1
*   0

MACROS:

Insert some lines into Buffer 0.
You
Type: (SPACEBAR)BUFFER ZERO 1.0 (ENTER)
(SPACEBAR)BUFFER ZERO 2.0 (ENTER)
(SPACEBAR)BUFFER ZERO 3.0 (ENTER)
(SPACEBAR)BUFFER ZERO 4.0 (ENTER)
Screen Shows: E: BUFFER ZERO 1.0
E: BUFFER ZERO 2.0
E: BUFFER ZERO 3.0
E: BUFFER ZERO 4.0

Display the text in Buffer 0.
You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↓)L*
BUFFER ZERO 1.0
BUFFER ZERO 2.0

47

```
                                        BUFFER ZERO 3.0
                                        BUFFER ZERO 4.0
```

Switch to Buffer 1.
> **You Type:** B (ENTER)
> **Screen Shows:** E:B

Display the text in Buffer 1.
> **You Type:** (CLEAR)7L* (ENTER)
> **Screen Shows:** E:(4)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:** E:+2
```
                                        BUFFER ONE 3.0
```

Switch to Buffer 0.
> **You Type:** B0 (ENTER)
> **Screen Shows:** E:B0

Display the text in Buffer 0.
> **You Type:** L* (ENTER)
> **Screen Shows:** E: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:** E:+
```
                                        BUFFER ZERO 2.0
```

Switch to Buffer 1.
> **You Type:** B (ENTER)
> **Screen Shows:** E: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)

48

Screen Shows: E:L✳
```
                        BUFFER  ONE  3.0
                        BUFFER  ONE  4.0
```

Switch to Buffer 0.
    You Type: B0 (ENTER)
    Screen Shows: E: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: E:L✳
```
                        BUFFER  ZERO  2.0
                        BUFFER  ZERO  3.0
                        BUFFER  ZERO  4.0.
```

Delete the contents of Buffer 0.
    You Type: (CLEAR)7D✳ (ENTER)
    Screen Shows: E:(↓)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: E:B

Move two lines from the primary buffer (Buffer 1) into the secondary buffer (Buffer 0).
    You Type: (CLEAR)7P2 (ENTER)
    Screen Shows: E:(↓)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 (CLEAR)7L✳ (ENTER)
    Screen Shows: E: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: E : 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: (CLEAR)7L* (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: E : 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: E : / G*

```
                    BUFFER ONE 3.0
                    BUFFER ONE 4.0
```

Show that the order of the lines is restored.

You Type: (CLEAR)7L*
Screen Shows: E : (↑)L*

```
                    BUFFER ONE 1.0
                    BUFFER ONE 2.0
                    BUFFER ONE 3.0
                    BUFFER ONE 4.0
```

50

Sample Session 4

Enter some lines of text.
    You Type: (SPACEBAR)LINE ONE (ENTER)
                (SPACEBAR)SECOND LINE OF TEXT
                    (ENTER)
                (SPACEBAR)THIRD LINE OF TEXT
                    (ENTER)
                (SPACEBAR)FOURTH LINE (ENTER)
                (SPACEBAR)FIFTH LINE (ENTER)
                (SPACEBAR)LAST LINE (ENTER)
    Screen Shows: E: LINE ONE
                E: SECOND LINE OF TEXT
                E: THIRD LINE OF TEXT
                E: FOURTH LINE
                E: FIFTH LINE
                E: LAST LINE

Open the file "oldfile" for writing.
    You Type: .WRITE"oldfile" (ENTER)
    Screen Shows: E:.WRITE"oldfile"

Write all lines to the file.
    You Type: (CLEAR)7W* (ENTER)
    Screen Shows: E:(↑)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: E:.WRITE//

Verify that the buffer is empty.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: E:(↑)L*

51

Open the file "oldfile" for reading.
  You Type: `.READ"oldfile"` (ENTER)
  Screen Shows: `E:.READ"oldfile"`

Create a new file called "newfile" for writing.
  You Type: `.WRITE"newfile"` (ENTER)
  Screen Shows: `E:.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: `E: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: `E:R*`

```
                    LINE FIVE
                    LAST LINE

          *END OF FILE*
```

Go to the top of the buffer and display the text to make sure that
it was inserted into the buffer.
  You Type: (CLEAR)`7L*` (ENTER)
  Screen Shows: `E:(↑)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: `E:W3`

```
                    LINE ONE
                    SECOND LINE OF TEXT
                    THIRD LINE OF TEXT
```

Move to the next line and display it.
You Type: + **(ENTER)**
Screen Shows: E:+
           FIFTH LINE

Show that the lines are written starting at the current line and not at the top of the buffer.
You Type: W **(ENTER)**
Screen Shows: E:W
           FIFTH LINE

Go to the top of the buffer and display the text to make sure that the lines were written to the output file.
You Type: **(CLEAR)**7L* **(ENTER)**
Screen Shows: E:**(↑)**L*
           FOURTH LINE
           LAST LINE

Clear the buffer.
You Type: **(CLEAR)**7D* **(ENTER)**
Screen Shows: E:**(↑)**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: E: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: E: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: E: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)**

53

Screen Shows: E:B R
                      SECOND LINE OF TEXT

Switch to Buffer 2 and delete its contents.
    You Type: B2 (CLEAR)7D* (ENTER)
    Screen Shows: E:B2 (↓)D*
                          LINE ONE
                          SECOND LINE OF TEXT
                          THIRD LINE OF TEXT

Insert some extra lines into the buffer.
    You Type: (SPACEBAR)EXTRA LINE ONE (ENTER)
                  (SPACEBAR)EXTRA LINE TWO (ENTER)
    Screen Shows: E: EXTRA LINE ONE
                      E: EXTRA LINE TWO

Try to write B2 buffer to file. It fails because a file has not been opened in this buffer.
    You Type: (CLEAR)7W* (ENTER)
    Screen Shows: E:(↓)W*
              FILE CLOSED

Close the file for Buffer 1 and return to Buffer 2.
    You Type: B .WRITE// B2 (ENTER)
    Screen Shows: E:B .WRITE// B2

Open the old "write" file for reading and then read it back in.
    You Type: .READ"newfile" R* (ENTER)
    Screen Shows: E:.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: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)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: (CLEAR)7D* (ENTER)
    Screen Shows: E:(↑)D*

Insert three lines.
    You Type: (SPACEBAR)LINE ONE (ENTER)
                (SPACEBAR)LINE TWO (ENTER)
                (SPACEBAR)LINE THREE (ENTER)
    Screen Shows: E: LINE ONE
                     LINE TWO
                     LINE THREE

Create a new macro using an empty string.
    You Type: ,MAC// (ENTER)
    Screen Shows: E:,MAC//

Display the contents of the macro mode, which is now open.
Note: The E prompt is now M.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: M:(↑)L*

Define the macro.
    You Type: (SPACEBAR)FIND (ENTER)
                (SPACEBAR)S"TWO" (ENTER)
    Screen Shows: M: FIND
                     S"TWO"

Display the contents of the macro.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: M:(↓)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: E:.DIR
                            BUFFERS:
                            $ 0
                            *   1

                            MACROS:
                              FIND

Display the contents of the edit buffer.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: E:(↓) L*
                            LINE ONE
                            LINE TWO
                            LINE THREE

Use the FIND macro to find the string "TWO."
    You Type: .FIND (ENTER)
    Screen Shows: E:.FIND
                            LINE TWO

Reopen the definition of the FIND macro.
    You Type: .MAC/FIND/ (ENTER)
    Screen Shows: E:.MAC/FIND/

Show that the macro is still intact.
    You Type: (CLEAR)7L* (ENTER)
    Screen Shows: M:(↓)L*
                            FIND
                            S"TWO"

56

Add the numberic parameter and the string parameter to the macro's header.

You Type: C/FIND/FIND #N $STR/ (ENTER)
Screen Shows: M:C/FIND/FIND #N $STR/
                      FIND #N $STR

Move to the second line of the macro.

You Type: + (ENTER)
Screen Shows: M:+
                      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: M: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: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)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: E:.FIND 2 /LINE/
                      LINE ONE
                      LINE TWO

Create a new macro.

You Type: .MAC// (ENTER)
Screen Shows: E:.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 the last occurrence of "STR" is found.

You Type: (SPACEBAR)FIND_LINE #N $STR (ENTER)
Screen Shows: M: FIND_LINE #N $STR

You Type: (SPACEBAR)S #N $STR (ENTER)
Screen Shows: M: S #N $STR

Turn off the verify mode.

You Type: (SPACEBAR)V0 (ENTER)
Screen Shows: M: V0

Move the edit pointer to the first character of the current line.

You Type: (SPACEBAR) -0 (ENTER)
Screen Shows: M: -0

Close the macro's definition.

You Type: Q (ENTER)
Screen Shows: M:Q
　　　　　　　　E:

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↓)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: E:.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: E:L
　　　　　　　　　LINE TWO

Create a new macro.

You Type: .MAC// (ENTER)
Screen Shows: E:.MAC//
                  M:

Use the exclamation point (!) command to comment itself. Type
the following:

```
M: CONVERT_TO_LINES #N
M: ! This is a comment
M: !
M: ! This macro converts
   the next n
M: ! space characters to new line
M: ! characters.
M: V0            ! Turn verify mode
                   off
M:               ! to prevent inter-
                   mediate results
M:               ! from being
                   displayed.
M: !
M: [             ! Begin loop
M: .SEARCH/ /    ! Search for <space>
                   character.
M: i//           ! Insert empty
                   line (new line
                   character).
M: -             ! Back up one line.
M: C/ //         ! Delete the
                   next space
                   character.
M: L +           ! Show line, move
                   past it.
M: ] #N          ! End of loop.
                   Repeat #N times.
```

Close the macro's definition.

You Type: Q (ENTER)
Screen Shows: M:Q
                  E:

Display the contents of the edit buffer.
You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)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: E:.CONVERT_TO_LINES *
                        LINE
                        LINE
                        LINE

Display the contents of the edit buffer.
You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
                        LINE
                        ONE
                        LINE
                        TWO
                        LINE
                        THREE

60

Appendices

Appendix A / Glossary

BUFFER	Holding area in memory for text.
EDIT POINTER	Internal marker the editor uses to remember your position in the edit buffer.
MACRO	New command you may define; it is composed of existing commands. Use macros to repeat the same command sequence over and over again.
PATHLIST	See the *OS-9 Commands*.
PSEUDO MACRO	Command that is part of the EDIT program and written in assembly language that is used as if it were a macro.
TEXT FILE	Place where OS-9 keeps the text that you want saved.
WORKSPACE	Available memory pool that the editor uses for buffers and macros.

Appendix B / Quick Reference Summary

EDIT	OS-9 loads the editor and starts it. There are no initial read or write files. Perform text-file operations by opening files after the editor is running.
EDIT *newfile*	If *newfile* does not exist, OS-9 loads the editor and starts it. The editor creates a file called *newfile*, and this is the initial write file. There is no initial read file; however, files may be read if they are opened after the editor is started.

EDIT *oldfile*   OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a new file called SCRATCH; this is the initial write file. When the edit session is complete, *oldfile* is deleted, and SCRATCH is given the name *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*, and this is 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.

(SPACEBAR)   Inserts 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.
(CLEAR)7	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.
An	Sets the SEARCH/CHANGE anchor to Column n, restricting searches and changes to those strings starting in the 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.
Bn	Makes buffer n the primary buffer.
Cn str1 str2	Changes the next n occurrences of string1 to string2.
Dn	Deletes n lines.
En str	Extends (adds the string to the end of) the next n lines.
Gn	Gets n lines from the secondary buffer, starting from the top. Inserts the lines before the current position in the primary buffer.
In 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 file(s) when you entered EDIT, Buffer 1 is written out 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, the line is extended with space.
U	Unextends (truncates) 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 string.
.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 an empty string is given, a new macro is created.
.NEOB	Tests for not end of buffer.
.NEOL	Tests for not end of file.
.NEW	Writes lines out to the initial output file up to the current line and then attempts to read an equal amount of text from the initial input file. The test 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 *string1* on the file specified by the pathlist in *string2*.
.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 *string* as a pathlist.
.ZERO *n*	Tests *n* to see if it is zero.
[	Starts at a macro loop; type (CLEAR)9.
]	Ends at a macro loop; TYPE (CLEAR)9.
(↑) type (CLEAR)7.	Moves edit pointer to beginning of buffer;

Appendix C / Editor Error Messages

BAD MACRO NAME	The first line in the macro does not begin 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. Usually, you omitted the variable name or inadvertently 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 typed (CONTROL)(C) or (CONTROL)(Q) to interrupt the editor. After printing the error message, the editor returns to command entry mode. **Note:** Results may not be displayed during command operation.
DUPL MACRO	You attempted to close a macro definition with the same name as another macro. 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 was never opened. 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	Close the macro definition before using the command that caused this error.
MISSING DELIM	The editor could not find a matching delimiter to complete the string you specified. You must put the string completely on one line.
NOT FOUND	The editor cannot find the specified string or macro.
UNDEFINED VAR	You used a variable that was not specified in the macro's definition parameter list. A variable parameter may be used only in the macro in which it is declared.

WHAT ??   The editor did not understand a com-
                          mand you typed. This is usually caused
                          by entering a command that does not
                          exist (misspelling its name).

WORKSPACE FULL   The buffer did not have room for the
                          text you attempted to insert. Increase
                          the workspace or remove some text.

OS-9 Assembler

1 / Introduction

The machine instructions executed by a computer are sequences of binary numbers that are difficult for people to deal with directly. Creating a machine-language program by hand is tedious, error prone, and time consuming. Assembly language bridges the gap between computers and machine-language programmers.

Assembly language uses descriptive mnemonics (abbreviations) for each machine instruction instead of numerical codes. These are much easier to learn, read, and renumber. The assembler also lets the programmer assign symbolic names to memory addresses and constant values.

This assembler is designed expressly for the modular, multi-tasking environment of the OS-9 Operating System and incorporates built-in functions for calling OS-9, generating memory modules, encouraging the creation of position-independent-code, and maintaining separate program and data sections. It is also optimized for use by OS-9 high-level language compilers such as Pascal and C.

The OS-9 assembler is extremely fast as a result of its tree-structured symbol table organization. This dramatically reduces the time for symbol table searching.

This manual describes how to use the OS-9 Assembler and explains basic programming techniques for the OS-9 environment. It is not a comprehensive course on assembly language programming or the 6809 instruction set.

If you are not familiar with these topics, consult the Motorola 6809 programming manuals and one of the many excellent assembly-language programming books available at libraries and bookstores.

Installation

The OS-9 Assembler distribution disk contains the asm file (the assembler program) and the DEFS file (a directory containing OS-9 common system-wide definition files; see chap-

ter 7). These files are OS9Defs, SysType, SCFDefs, and RBFDefs.

Copy the asm file to the CMDS directory of your system disk. Create a directory on your system disk called DEFS (if not already present) and copy the four DEFS files to it. Place the assembler distribution disk in Drive 1 and your system disk in Drive 0. Use the following commands.

```
copy/d1/asm/d0/cmds/asm #12k
makdir /d0/DEFS
copy /d1/defs/os9defs /d0/defs/os9defs #12k
copy /d1/defs/systype /d0/defs/systype #12k
copy /d1/defs/scfdefs /d0/defs/scfdefs #12k
copy /d1/defs/rbfdefs /d0/defs/rbfdefs #12k
```

Assembly Language Program Development

Writing and testing assembly language programs involves a basic edit-assemble-test cycle.

1. Create a source program file using the text editor.

2. Run the assembler to translate the source file to a machine-language file.

3. If the assembler reports errors, use the text editor to correct the source file. Go back to step 2.

4. Run and test the program, using the OS-9 Interactive Debugger.

5. If the program has bugs, use the text editor to correct the source file. Go back to step 2.

6. Document the program.

Assembler Input Files

The OS-9 Assembler reads from an input file (path) that contains variable-length lines of ASCII characters. You can correct and edit input files with the OS-9 Macro Text Editor or with any other standard text editor.

The maximum length of the input line is 120 characters. Each line contains assembler statements as explained in this manual. Terminate every line by typing (ENTER).

Running the Assembler

The assembler is a command program that can be run from the OS-9 Shell, from a shell procedure file, or from another program. The disk file and memory module names are ASM. The following is the basic format of a command line to run the assembler:

    asm file name [option(s)] [#memsize] [>listing]

Brackets enclose options; therefore, the only required items are the ASM command name and the file name, which is the source text file name (pathlist). The following is a typical command:

    asm prog5   l s   -c   #12k   >/p

In this example the source program is read from the file PROG5. The source file name can be followed by an option list, which allows you to control various factors such as whether or not to generate a listing or an object file.

The option list consists of one or more abbreviations separated by spaces or commas. An option is turned on by its presence in the list; a minus followed by an option abbreviation turns off the function. If an option is not expressly given, the assembler assumes a default condition for it.

You can override command options by OPT statements within the source program. In the example above, the options l and s are turned on, and c is turned off.

The shell processes the optional #memsize item to specify how much data area memory the assembler is assigned. If this is not specified, the assembler is assigned 4K bytes of memory in its data area. Most of this space is used to store the symbol table. Any additional memory that this option requests enlarges the symbol table.

Large programs generally use more symbols; therefore, their memory requirements are correspondingly greater. If the assembler generates the "Symbol Table Full" error message, increase the assembler's memory size. In the previous example, 12K bytes of memory are specified.

The final item, ">listing", allows the program listing generated by the assembler (on the standard output path) to be optionally redirected to another pathlist, which may be an output device such as a printer, a disk file, or a pipe to another program.

The shell handles the memory size option, and it handles output redirection, not the assembler. If you omit this item from the command line, your screen shows the output. In the above example, the listing output is directed to device p, the printer on most OS-9 systems.

Operating Modes

The OS-9 Assembler has a number of features specifically designed to conveniently develop machine-language programs for the OS-9 environment. These include special assembler directive statements for generating OS-9 memory modules, identification of 6809 addressing modes that are not usually permitted in OS-9 programs, and separate data and program address counters.

The assembler has two operating modes — normal and Motorola-compatible. In normal mode, the features mentioned above are active. In the Motorola-compatible mode,

the assembler works the same way as a standard 6809 "absolute" assembler (without separate program and data counters). This mode exists so that you can use the assembler to generate programs for 6809 computers that are not equipped with OS-9.

The assembler is in the normal mode unless you use the m option in the command line or in an OPT statement.
The -m option returns the assembler to the normal mode (you can switch modes freely to achieve special effects).

The assembler performs two "passes" (complete scans) over the source file. During each pass, it reads input lines and processes them one at a time. During the first pass, it creates the symbol table. It generates most error messages, the program listing, and the object code during the second pass.

2 / Source Statement Fields

Each input line is a text string that you terminate by typing (ENTER). The line can have from one to four fields — a label field, an operation field, an operand field (for some operations), and a comment field.

If you type an asterisk as the first character of a line, the assembler treats the entire line as a comment. It displays it in the listing but does not otherwise process it. The assembler ignores blank lines but includes them in the listing.

Label Field

The label field begins in the first character position of the line. Some statements require labels (for example, EQU and SET); others (assembler directives such as SPC, TTL) must not have them. The first character of the line must be a space if the line does not contain a label.

The label must be a legal symbolic name consisting of from one to eight upper- or lower-case characters, decimal digits, or the characters dollar sign ($), underline (_), or dot (.); however, the first character must be a letter. You must not define labels (and names in general) more than once in a program (except when used with the SET directive).

The symbol table stores label names with an associated 16-bit value, which is normally the program counter address before code is generated for the line. In other words, instructions and most constant-definition statements associate the label name with the value of the program address of the first object code byte generated for the line.

An exception to this rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field. In other words, these statements allow any value to be associated with a symbolic name.

Likewise, labels on RMB statements are given the value of the data address counter when in normal assembler mode, or the value of the program address counter when in Motorola-compatible mode.

77

Operation Field

This field specifies the machine-language instruction or assembler directive statement mnemonic name. It immediately follows and is separated from the label field by one or more spaces.

Some instructions must include a register name that is part of the operation field (for example, LDA, LDD, LDU). In these instructions the register name must be part of the name and cannot be separated by spaces as in older 6800-type assemblers. The assembler accepts instruction mnemonic names in either upper- or lower-case characters.

Instructions generate one to five bytes of object code, depending on the specific instruction and addressing mode. Some assembler directive statements (such as FCB, FCC) also generate object code.

Operand Field

The operand field follows and must be separated by at least one space from the operation field. Some instructions do not use an operand field; other instructions and assembler directives require one to specify an addressing mode, operand, address, parameters, and so on.

Comment Field

The comment field is the last field of the source statement and is used to include a descriptive comment. It is optional. The assembler does not process this field but copies it to the program listing.

3 / Symbolic Names and Expressions

Evaluation of Expressions

Operands of many instructions and assembler directives include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity, using a form similar to the algebraic notation in programming languages such as BASIC and FORTRAN.

Expressions consist of operands, which are symbolic names or constants, and operators, which specify an arithmetic or logical function. All assembler arithmetic uses two-byte (internally, 16-bit binary) signed or unsigned integers in the range of 0 to 65535 for unsigned numbers and -32768 to $+32767$ for signed numbers.

In some cases, expressions are expected to evaluate to a value that must fit in one byte (such as 8-bit register instructions) and therefore must be in the range of 0 to 255 for unsigned values and -128 to 127 for signed values. In these cases, if the result of an expression is outside this range, the screen shows an error message.

The assembler evaluates expressions from left to right, using the algebraic order of operations (that is, it multiplies and divides before it adds and subtracts). Parentheses alter the natural order of evaluation.

Expression Operands

You may use the following items as operands within an expression.

Decimal Numbers. Optional minus sign ($-$) and one to five digits. Examples:

```
100
32767
0
12
```

Hexadecimal Numbers. Dollar sign ($) followed by one to four hexadecimal characters (0-9, A-F, or a-f). Examples:

```
$EC00
$1000
$3
$0300
```

Binary Numbers. Percent sign (%) followed by one to sixteen binary digits (0 to 1). Examples:

```
%0101
%1111000011110000
%10101010
```

Character Constants. Single quote (`) followed by any printable ASCII character. Examples:

```
`X
`c
`5
`c
```

Symbolic Names. One to eight characters, upper- and lower-case alpha (A-Z or a-z), digits (0-9), and special characters _, ., or $ (underscore, period, or dollar sign). The first character cannot be a digit.

Instruction Counter. Placed at the beginning of the line, the asterisk (*) represents the program instruction counter value.

Data Counter. Placed at the beginning of the line, the period (.) represents the data storage counter value.

Operators

"Operators" specify arithmetic or logical operations to be performed within an expression. The assembler executes operators in the following order: (1) −, negative numbers; (2) & and !, logical AND and OR; (3) * and /, multiplication

and division; (4) + and −, addition and subtraction. Operators in a single expression having equal precedence, for example, + and −, are evaluated left to right. You can use parentheses, however, to override precedence.

Assembler Operators by Order of Evaluation

−	negative	^	logical NOT
&	logical AND	!	logical OR
*	multiplication	/	division
+	addition	−	subtraction

Logical operations are performed bitwise; that is, the logical function is performed bit by bit on each bit of the operands.

Division and multiplication functions assume unsigned operands, but subtraction and addition work on signed (2's complement) or unsigned numbers. The screen shows an error message if you attempt to divide by zero or multiply by a factor that results in a product larger than 65, 536.

Symbolic Names

A symbolic name consists of from one to eight upper- or lower-case characters, decimal digits, or the dollar sign ($), the underline (_), or the dot (.); however, the first character must be a letter. The following are examples of legal symbol names:

HERE
there
SPL030
VX_GH
abc.def
Q1020.1
L.123.X
t$integer

These are examples of illegal symbol names:

2move	(does not start with a letter)
main.backup	(has more than eight characters)
lbl#123	(contains #, which is not a legal name character)

You define a name the first time you use it as a label on an instruction or directive statement. You can define a name only once in the program (except SET labels). If you redefine a name (use as a label more than once), the screen shows an error message. You cannot use multiple forward references (that is, a definition using currently undefined names).

The symbol table stores symbolic names with their associated type and value. This structure uses most of the assembler's data memory space. Using the default memory size of 4K, the symbol table has room for approximately 200 names.

You can use the shell's optional memory size modifier to increase the assembler's memory space. Each entry in the table requires 15 bytes; therefore, each additional 4K of memory adds space for about 273 additional names.

For example, the command line

    asm   sourcefile   #16K

gives the symbol table enough space for a little more than a thousand names. If you select the S option, the assembler generates an alphabetical display of all symbol names, types, and values at the end of the assembly.

4 / Instruction Addressing Modes

The instruction set has a wide variety of addressing modes. Each group of similar instructions is used with specific addressing modes, which are usually specified in the assembler source statement operand field. The assembler generates an error message if an addressing mode is specified that cannot legally be used with the specific instruction.

Inherent Addressing

Certain instructions do not need operands (for example, SYNC and SWI); others implicitly specify operands (for example, MUL and ABX). In these cases no operand field is needed.

Accumulator Addressing

Some instructions have the A or B accumulators as operands. Examples:

    CLRA
    ASLB
    INCA

Immediate Addressing

In immediate addressing, the instruction uses the operand bytes as the actual value. Instructions that use 8-bit registers must have operand expressions that evaluate in the range of 0 to 255 (unsigned) or -128 to 127 (signed). If they do not, the screen shows an error message.

The syntax is:

    instr   #expression

Examples:

    LDD   #$1F00
    ldb   #bufsiz + 2
    ORCC   #$FF-CBIT

Relative Addressing

Branch-type instructions, such as BCC, BEQ, LBNE, BSR, and LBSR, use the relative addressing mode. The operand field is an expression that is the "destination" of the instruction, which is almost always a name used as a statement label somewhere in the program.

The assembler computes an 8- or 16-bit program counter offset to the destination, which is made part of the instruction. If the destination of short branch-type instructions is not in the range of -126 to $+129$ bytes of the instruction address, the screen shows the error message.

Long branch-type instructions can reference any destination. If a long branch instruction references a destination that is within the range of a smaller and faster short branch instruction, the assembler places a warning symbol (W) in the listing line's information field. All instructions using relative addressing are inherently position independent code. Examples:

    BCS   LOOP
    LBNE   LABEL5
    LBSR   START + 3
    BLT   COUNT

Extended and Extended Indirect Addressing

Extended addressing uses the second and third bytes of the instruction as the absolute address of the operand. Data section addresses of OS-9 programs are assigned when the program is actually executed; therefore, absolute memory addresses are not known before the program is run, and this

addressing mode is not normally used in OS-9 programs. The screen shows an informational warning flag (W) if this addressing mode is specified.

Extended indirect addressing is similar to extended addressing except that the address part of the machine instruction is used as the address of a memory location containing the address of the operand.

Because this mode also uses absolute addresses, it is not frequently used in OS-9, and the assembler flags it with a warning. Select this addressing mode by enclosing the address expression in brackets. Examples:

```
ADDA    $1C48      extended addressing
ADDA    [$D58A]    extended indirect addressing
LBD     START      extended addressing
stb     |end|      extended indirect addressing
```

Direct Addressing

Direct addressing uses the second byte of the instruction as the least significant byte of the operand's address. The most significant byte is obtained from the MPU's direct page register.

This addressing mode is preferred for accessing most variables in OS-9 programs because OS-9 automatically assigns unique direct pages to each task at run time and because this mode produces short, fast instructions.

The syntax for extended and direct addressing has the same form:

```
instr    -addr expr  -
```

The assembler automatically selects direct addressing mode if the high-order byte of the address matches its internal "direct page." This direct page is not the same as the run-time direct page register; it is an assembly-time value. You ordinarily set it to zero, but you can change it with the SETDP directive.

85

You can force the assembler to use direct addressing by typing the less than symbol (<) just before the address expression or to use extended addressing by typing the greater than symbol (>) just before the address expression. Examples:

lda	temp	(assembler selects mode)
LDD	>PAI + 1	(forces extended addressing)
ldx	<count	(forces direct addressing)
STD	[pointer]	(extended indirect)

Register Addressing

Some instructions operate on various MPU registers, which are referred to by a one- or two-letter name. In these instructions the operand field specifies one or more register names. The names, which can be upper- or lower-case, are:

A	accumulator A (8 bits)
B	accumulator B (8 bits)
D	accumulator A:B concatenated (16 bits)
DP	direct page register (8 bits)
CC	condition codes register (8 bits)
X	index register X (16 bits)
Y	index register Y (16 bits)
S	stack pointer register (16 bits)
U	user stack pointer register (16 bits)
PC	program counter register (16 bits)

The EXG and TFR instructions have the form:

instr reg,reg

If the registers are not the same size (either 8 or 16 bits), the screen shows the error message.

The PSHS, PSHU, PULS, and PULU instructions accept a list of one or more register names. Even though the assembler accepts register names in any order, the MPU stacks and unstacks them in a specific order.

The syntax for these instructions is:

instr reg{,reg}

Examples:

    TFR   X,Y
    EXG   A,DP
    pshs   a,b,x,dp
    PULU   d,x,pc

Indexed Addressing

The 6809 has 23 varieties of indexed addressing modes. Indexed addressing is analogous to "register indirect," meaning that an indexable register (X, Y, U, S, or PC) is used as the basic address of the instruction's operand.

The different varieties of indexed addressing use the specified register contents, which may be unchanged, temporarily modified, or permanently modified, depending on the mode.

All indexed modes must specify an index register, either X, Y, U, or SP. You can use the Register PC with the program-counter relative mode only.

To make any indexed addressing mode "indirect," enclose the operand field in brackets. The effective address generated by the addressing mode is used as the address of a pointer to the operand rather than as the address of the operand.

Constant Offset Indexed

This mode uses an optional signed (two's complement) offset that is temporarily added to the register's value to form the operand's effective address. The offset can be any number: if it is zero, the register's unaltered content is used as the effective address. The assembler automatically picks the shortest of four possible varieties that can represent the offset.

If a symbolic name used in the offset expression is not defined, the assembler generates longer code than necessary or produces phasing errors.

The syntax for constant offset indexed instructions is:

instr	,reg	zero offset
instr	offset,reg	constant offset
instr	[,reg]	zero offset indirect
instr	[offset,reg]	constant-offset indirect

Examples:

lda	,x	no offset
lda	0,x	no offset
ldx	100, x	offset of 100
LDB	COUNT,S	offset of COUNT
ldd	temp + 2,y	offset of temp + 2
leax	− 2,y	offset of − 2
clr	[PIA,X]	indirect mode

Program Counter Relative Indexed

This addressing mode is similar to constant-offset indexed except that the program counter register (PC or PCR) is used as an index register, and the assembler computes the offset differently. Instead of using the offset expression directly, the expression is assumed to refer to the address of the operand.

The assembler calculates the required offset from the current program counter location to the operand's address and uses the resulting value as the offset. One form of this instruction uses an 8-bit offset and the other uses a 16-bit offset. The assembler uses the 16-bit form unless you force the short form by typing the less than symbol (<) before the operand field.

The syntax for program-counter relative indexed is:

instr addr,PC	program counter relative
instr addr,PCR	program counter relative
instr [addr,PCR]	program counter relative indirect
instr [addr,PC]	program counter relative indirect

This addressing mode permits addresses of constants and constant tables to be accessed using position independent code as required by OS-9.

Examples:

```
ldd    temp,pcr
LDD    temp,pc           same as instruction above
leax   table,pcr
jsr    addr,pcr          same as "lbsr addr"
CLR    [control + 4,PCR]  dangerous; uses absolute
                         address at "control + 4,PCR"
                         as effective address for clear
```

Accumulator Offset Indexed

In this mode the contents of the A, B, or D accumulators are temporarily added to the specified index register to form the address of the operand. This addition is signed two's complement.

If you specify the A or B accumulators, the sign bit is "extended" to form the 16-bit value, which is added to the index register. This means that if the most significant bit of the accumulator is set, the high order byte of the offset is $FF.

Beware: This is a commonly overlooked characteristic that can produce unexpected results! Using the D register avoids this because it gives all 16 bits.

The syntax for accumulator-offset indexed is:

```
instr   A,reg
instr   B,reg
instr   D,reg
```

Examples:

```
LDX B,Y
LEAY D,X
ROL [B,U]
```

Auto-Increment and Auto-Decrement Indexed

These addressing modes use the specified index register as the effective address of the operand while permanently adding or subtracting one or two from the register. In auto-increment mode, the increment is performed after the register is used. In auto-decrement mode, the decrement is performed before the register is used. This is consistent with the way 6809 stack pointers operate in PSH and PUL instructions.

If you use indirect addressing, the decrement and increment are performed before the effective address is used as a pointer to the operand. You cannot use single auto-increment and single auto-decrement when you select indirect addressing.

Syntax for auto-increment and auto-decrement indexed addressing is:

instr	, − reg	single auto-decrement
instr	, − − reg	double auto-decrement
instr	,reg +	single auto-increment
instr	,reg + +	double auto-increment
instr	[,reg − −]	double auto-decrement indirect
instr	[,reg + +]	double auto-increment indirect

Examples:

clr ,x + +
LDX ,− − Y
lda ,s + is the same as puls a (except CCR is affected)
sta ,− s is the same as pshs a (except CCR is affected)
ldd [,s + +]

5 / Pseudo Instructions

Pseudo instructions are special assembler statements that generate object code but do not correspond to actual 6809 machine instructions. Their primary purpose is to create special sequences of constant data to be included in the program. Labels are optional on pseudo instructions.

FCB <expression> {, <expression>}

generates sequences of single byte constants (FCB) within the program. The screen shows the error message if an expression has a value of more than 255 or less than − 128 (the largest number that can be represented by a byte).

FDB <expression> {, <expression>}

generates sequences of double byte constants (FDB) within the program. If this statement evaluates an expression with an absolute value of less than 256, the high order-byte is zero.

The operand is a list of one or more expressions that are evaluated and output as constants. To generate more than one constant, separate the expressions with commas.

Examples:

    FCB 1,20,A

    fcb index/2 + 1,0,0,1

    FBD 1,10,100,1000,10000

    fdb $F900,$FA00,$FB00,$FC00

FCB 'A
FDB 'CN'
FCB '? + 80
FCC *string*
FCS *string*

generate a series of bytes corresponding to a specified *string* of one or more characters operand.

The output bytes are the literal numeric value of each ASCII character in the specified *string*. FCS is the same as FCC except the most significant bit (the sign bit) of the last character in the specified *string* is set. This is a common OS-9 programming technique to indicate the end of a text string without using additional storage.

You must enclose the characters in the specified *string* with delimiters. You can use the following characters as delimiters:

   ! " # $ % & ' ( ) * + , - . /

The delimiters must be identical, and you cannot include them in the *string* itself. Examples:

FCC	/most programmers are strange people/
FCS	,0123456789,
fcc	z

MOD size,nameoff,typelang,attrrev {,execoff,memsize}

creates a standard OS-9 module header and initializes a CRC (cyclical redundancy check) value that the assembler automatically computes as it processes the program.

OS-9 can load programs into memory only if they are in module header format. You use the MOD statement at the beginning of an OS-9 module. It must have an operand list of exactly four or exactly six expressions separated by commas. Each operand corresponds, in order, to the elements of a module header. The exact operation of the MOD statement is as follows:

1. Resets the assembler's program address counter and data address counters to zero (same as ORG 0) and initializes the internal CRC and vertical parity generators.

2. Generates the sync codes $87 and $CD as object code.

3. Evaluates and outputs as object code the first four expressions in the operand list. They are:

a. module size (two bytes)
b. module name offset (two bytes)
c. type-language byte (one byte)
d. attribute-revision byte (one byte)

4. Computes the header parity byte from the previous bytes and generates it as object code.

5. Evaluates the two optional additional operands if they are present and generates them as object code. They are:

e. execution offset
f. permanent storage size

Note: Some expressions in the operand list are one byte long, and others are two bytes.

Because the origin of the object program is zero, all labels used in the program are inherently relative to the beginning of the module. This is perfect for the module name and execution address offsets. The code in the body of the module follows. As subsequent lines are assembled, the internal CRC generator continuously updates the module's CRC value.

EMOD

terminates the module.

The EMOD statement has no operand. It outputs the correct three-byte CRC generated over the entire module.

Note: The MOD and EMOD statements do not work correctly if the assembler is in Motorola-compatible mode unless you do not use RMB or ORG statements after the MOD and before the EMOD.

The example below illustrates the basic techniques of creating a module using MOD and EMOD statements.

```
type    set PRGRM + OBJCT (these are defined in
             OS9DEFS)
```

```
revs      set REENT + 1          (this is defined in
             OS9DEFS)
          MOD pgmlen,name,type,revs,start,memsize

          * data storage declarations

temp      RMB 1
addr      RMB 2
buffer    RMB 500
stack     RMB 250
memsiz    EQU        data storage size is final ``.'' value

name      FCS /textmodule/

start     leax buffer,u get address of buffer
          clr temp
          inc temp
          ldd #500 loop count
loop      clr ,x +
          subd #1
          bne loop
          os9 F$EXIT return to OS9
          EMOD

pgmlen EQU * program size is addr of last byte + 1
```

OS9 <expression>

generates OS-9 system calls.

This statement has an operand that is a byte value to be used as the request code. The output is equivalent to the instruction sequence:

```
SW12
FCB operand
```

The OS9Defs file contains standard definitions of the symbolic names of all OS-9 service requests. You can use these names in conjunction with the OS9 statement to improve the readability, portability, and maintainability of assembly-language software.

Examples:

OS9 I$Read   (call OS-9 READ service request)

OS9 F$Exit   (call OS-9 EXIT service request)

6 / Assembler Directive Statements

Assembler directive statements give the assembler information that affects the assembly process but that does not generate code. Read the descriptions carefully because some directives require labels, labels are optional on others, and a few cannot have labels.

END

indicates the end of a program.

The use of this statement is optional since END is assumed upon an end-of-file condition on the source file. End statements may not have labels.

label EQU <*expression*>
label SET <*expression*>

assign a value to a symbolic name (the label).

The value assigned to the symbol is the value of the operand, which may be an expression, a name, or a constant.
These statements require labels.

Note: If you define symbols by EQU statements, you can define them only once in the program. If you define symbols by SET statements, you can redefine them by subsequent SET statements.

In EQU statements the label name must not have been used previously, and the operand cannot include a name that has not yet been defined (that is, it cannot contain as-yet undefined names the definitions of which also use undefined names).

In a good program all equates are at the beginning. This lets the assembler generate the most compact code by selecting direct addressing wherever possible.

You can use EQU to define program symbolic constants, especially those used in conjunction with instructions. You can use SET to define symbols that control the assembler operations, especially conditional assembly and listing control. Examples:

```
FIVE        equ     5

OFFSET      equ     address-base

TRUE        equ     $FF
FALSE       equ     0
SUBSET      set     TRUE
            ifne    SUBSET
            use     subset.defs
            else
            use     full.defs
            endc
SUBSET      set     FALSE
```

IFxx *<expression>*
<statements>
[ELSE]
<statements>
ENDC

The assembler has conditional assembly capability. It can selectively assemble or not assemble one or more parts of a program, depending on a variable or computed value. Therefore, a single source file can selectively generate multiple versions of a program.

Conditional assembly uses statements similar to the branching statements in high-level languages such as Pascal and BASIC. The generic IF statement is the basis of this capability. Its operand is a symbolic name or an expression.

The assembler compares the results. If the results are true, the assembler processes the statement following the IF statement. If the results are false, the assembler does not process the statement until it encounters an ENDC (or ELSE) statement.

Hence, the ENDC statement marks the end of a conditionally assembled program section. In the following example the IFEQ statement tests for equality of its operand with zero:

```
IFEQ SWITCH
ldd #0          assembled only if SWITCH = 0
leax 1,x
ENDC
```

The ELSE statement lets the IF statement select one of two program sections to assemble, depending on the truth of the IF statement. The assembler processes statements following the ELSE statement only if the results of the comparison are false. For example:

```
IFEQ SWITCH
ldd #0          assembled only if SWITCH = 0
leax 1,x
ELSE
ldd #1          assembled only if SWITCH is not = 0
leax -1,x
ENDC
```

You can use multiple IF statements and nest them within other IF statements. They cannot, however, have labels. Each IF statement performs a different comparison.

```
IFEQ    True if operand equals zero
IFNE    True if operand does not equal zero
IFLT    True if operand is less than zero
IFLE    True if operand is less than or equal to zero
IFGT    True if operand is greater than zero
IFGE    True if operand is greater than or equal to zero
IFP1    True only during Pass 1 (no operand)
```

You can use the IF statements that test for less than or greater to test the relative value of two symbols if they are subtracted in the operand expression. For example,

```
IFLE    MAX-MIN
```

is true if MIN is greater then MAX.

Note: The logic is reversed, because this statement literally means

    IF MAX-MIN < = 0

The IFP1 statement causes subsequent statements to be processed during Pass 1, but skipped during Pass 2. It is useful because it allows program sections that contain only symbolic definitions to be processed only once during the assembly. Pass 1 is the only pass during which they are actually processed because they do not generate actual object code output.

The OS9Defs file is a rather large section of such definitions. For example, many source files have the following statement at the beginning.

    IFP1
    use   /d0/defs/OS9Defs
    ENDC

NAM *string*
TTL *string*

define or redefine a program name and listing title line that is printed on the first line of each listing page's header.

These statements cannot have label or comment fields.

These statements display the program name on the left side of the second line of each listing page; a dash and the title line follow. You may change the name and title as often as you wish.

    Examples:

        nam DATAC
        ttl Data Acquisition System

    Generates:

        Datac - Data Acquisition System

OPT *<option>*

sets or resets any of several assembler control options.

The operand of the OPT statement is one of the characters that represents the various options. If a minus sign (−) precedes the option name, the option is turned off; otherwise, it is turned on. Two exceptions are the D and W options, which must be followed by a number. This statement must not have label or comment fields.

Option Default (initial) State

C	Conditionals On — displays conditional assembly statements in the listing. (C)
D*num*	Page Depth — sets the number of lines per listing page, including heading and blank line. (D66)
E	Error Messages On — displays error messages in listing. When this option is off, an E appears in a statement's informational field if an error is present. (E)
F	Use Form Feed — uses a form feed for page eject instead of line feeds. (− F)
G	Generate All Constant Lines — displays all lines of code generated by pseudo instructions. Otherwise, it displays only the first line. (− G)
L	Listing On — generates formatted assembly listing. If off, the assembler displays only error messages. (− L)
M	Mode On — turns on Motorola-compatible mode. (− M)
N	Narrow Listing — generates listing in a non-columnized, compressed format for better presentation on narrow video display devices. (− N)
O	| − file name | generates object code file: (O)

If you do not specify a file name, the assembler creates an object file having the same name as the input file in the current execution directory.

If you specify a single name, the assembler creates an object file having that name but still in the current execution directory.

If you specify a full pathlist, the assembler uses it as the name specification of the device, directory, and file to create.

S   Generate Symbol Table — displays the entire contents of the symbol table at the end of the assembly. Displays each name, its value, and a type code character:

> D = data variable (RMB definitions)
> E = equate label (EQU)
> L = program label
> S = set label
> U = undefined name

The table is displayed across the page in alphabetical order. (− S)

W*num*   Set Page Width — defines the maximum length of each listing line. Lines are truncated if they exceed this number. The comment field starts at column 50; therefore, a number smaller than this may cause important parts of the listing to be lost. (W80)

Examples:

opt l
opt w72
opt s

ORG <*expression*>

changes the value of the assembler's data location counter (normal mode) or the instruction location counter (Motorola-compatible mode).

It evaluates the expression and sets the appropriate counter to the value of the result. ORG statements cannot have labels.

Note: OS-9 does not use load records that specify absolute addresses of the generated object code. The object code is assumed to be a contiguous memory module. Therefore, programs assembled using the Motorola-compatible mode that alter the instruction address counter do not load correctly.

Examples:

    ORG   DATAMEN

    ORG   . + 200

PAG[E]

begins a new page of the listing. The alternate form of PAG is PAGE for Motorola compatibility.

SPC <*expression*>

puts blank lines in the listing.

The value of the operand, which can be an expression, constant, or name, determines the number of blank lines to be generated. If you use no operand, a single blank line is generated.

The above two statements improve the readability of program listings. They are not themselves displayed and cannot have labels.

SETDP <*expression*>

assigns a value to the assembler's internal direct page counter, which is used to automatically select direct versus extended addressing.

The direct page counter does not necessarily correspond to the program's actual direct page register during execution. The default value of the counter is zero and should not be changed in OS-9 programs; this statement is intended for use with the Motorola-compatible mode only. SETDP statements cannot have labels.

USE *pathlist*

temporarily stops the assembler from reading the current input file. It then requests OS-9 to open another file or device specified by the pathlist, from which it reads input lines until an end-of-file occurs. At that point, the assembler closes the latest file and resumes reading the previous file from the statement following the USE statement.

You can nest USE statements (for example, a file being read as the result of a USE statement can also perform USE statements) up to the number of simultaneously open files the operating system allows (usually 13, not including the standard I/O paths). Some useful applications of the USE statement are to accept interactive input from the keyboard during assembly of a disk file (as in USE/TERM) and to include library definitions or subroutines into other programs. USE statements cannot have labels.

7 / DEFS Files: Fact or Fiction

OS9DEFS

Most programmers use the OS9Defs file with assembly-language programs and add their own definitions to this file. To include the OS9Defs file with your source code when assembling the file, use the following statements:

    IFP1
    USE /D0/DEFS/OS9DEFS
    ENDC

This speeds up assembly and prevents the OS9Defs file from being displayed every time.

OS9Defs contains the following groups of defined symbols:

    System Service Request Code
    Signal Codes
    Status Codes For Getstat/Putstat
    Direct Page Variables
    Table Size
    Module Format and Offsets
    Module Field Definitions
    Module Type/Language Masks and Definitions
    Process Descriptor
    Process Status Flags
    OS-9 System Entry Vectors
    Path Descriptor Offsets
    File Access Modes
    Pathlist Special Symbols
    File Manager Entry Offsets
    Device Driver Entry Offsets
    Device Table Format
    Device Static Storage Offsets
    Interrupt Polling Table Format
    Register Offsets on Stack
    Condition Code Bits
    System Error Codes
    I/O Error Codes

This chapter is a reference source, not a guide to the structure and workings of the operating system. For more extensive information, see *OS-9 Technical Information*.

System Service Request Codes

a group of labels that define all OS-9 system calls and list their associated values. These labels let you use the system request name in an OS-9 call.

Signal Codes

a group of labels that define the four OS-9 signals and their associated values. This is an appropriate place for your own user defined signals.

Status Codes for GetStt and SetStt

a group of labels that list and give the values for the predefined status call functions of the system calls I$GetStt and I$SetStt that are supported by OS-9 file managers and device drivers. These labels are then available for loading the B register before the call is made. This is an appropriate place for your own user defined status codes.

Direct Page Variables

a group of labels that define the offsets into page 0 of OS-9 system variables. OS-9 uses page 0 variables for interrupt vectors, table addresses, process queues, and internal memory information. We strongly recommend that you not use page 0 variables in your programs, unless you write special drivers and interrupt handlers or debug systems. Using these variables improperly can cause unexpected and perhaps fatal system operation.

Table Size

a group of equates that define the size of the table that the OS-9 operating system uses internally.

Module Format and Offsets

a group of labels that define the offsets into a module header of all OS-9 compatible modules. You can use module offsets to find information in a module, for example, the module's size, name, type, or language. In this group are the Universal Module offsets and the offsets for specific module types. Descriptors, drivers, programs, and file managers have a different module format.

Module Field Definitions
Module Type/Language Masks and Offsets
Module Attributes/Revision Masks and Offsets

a group of symbols that define the bits of information that go into a module header. You can use this section to decode a module header and modify one. Since the OS-9 Interactive Assembler generates a module header with the MOD and EMOD, use these symbols to read a header. This group defines masks according to type, language, attribute, and revision bytes and lists the values for each field.

Process Descriptor

contains the table of information describing a process.

Process Status Flags

define the flags that OS-9 uses to mark a process for different states, for example, dead and sleeping.

OS-9 System Entry Vectors

a group of symbols that define OS-9 system entry points. These are the vector addresses for the various interrupts. They are pseudo vectors, not actual hardware vector points.

Path Descriptor Offsets

a group of symbols that define the offsets for OS-9 path descriptors. OS-9 creates a path descriptor offset for every path opened in the system.

File Access Modes

a group of symbols that define the file access modes under OS-9. You can use these definitions for I$Create and I$Open system calls that require the file attributes to be set at the time of the call.

Pathlist Special Symbols

a group of symbols that define the special pathlist characters. You can use them to parse a pathlist. This is an appropriate place to insert special characters that you use frequently.

File Manager Entry Offsets

a group of symbols that define the entry offsets of all file managers on an OS-9 system. If you write your own file manager, you must provide these entry offsets.

Device Driver Entry Offsets

a group of symbols that define all entry offsets of device drivers in the OS-9 system. You must provide these offsets at the beginning of your drivers.

Device Table Format

a group of symbols that define the form of the table that contains an entry for every active device in the OS-9 system. The operating system uses this information internally.

Device Static Storage Offsets

a group of symbols that define the variables within a device static storage area. This area contains information about the device and is filled in when the device is activated. The actual

filling in of the parameters is done by three sources: IOMAN, the file manager, and the device driver. The offsets listed in this area are filled in by IOMAN.

Interrupt Polling Table Format

a group of symbols that define the structure of the entries for the polling table. The format contains all the information the interrupt service routine needs to handle interrupts generated by active devices. OS-9 uses this information internally.

Register Offsets on Stack

a group of symbols that define the offset to the registers that are pushed on the stack whenever the 6809 CPU gets an interrupt of the form NMI, IRQ, SWI, SWI2 (an OS-9 system call), or SWI3. You can use these symbols when writing drivers to get the I$GetStt or I$SetStt codes. You can also use them to pass parameters on the stack to different procedures in a program.

Condition Code Bits

a group of symbols that define the values for each condition code. Use these masks to set or reset the bits. It is good programming practice to use these labels in your code.

System Error Codes
I/O Error Codes

a group of labels that define all errors returned by OS-9 and the I/O handlers. If your programs have any form of error trapping, you must compare the error to a known error definition in order to determine what should occur.

SCFDEFS

In this file are the definitions pertaining to the sequential file manager and sequential file devices. It contains the following groups of defined symbols:

> Static Storage Requirements
> Character Definitions
> File Descriptor Offsets

You can use this file when writing drivers for sequential devices and managers. This is an appropriate place for your own SCF definitions. For more information on any group of defined symbols, see the *OS-9 Technical Information*.

Static Storage Requirements

a group of symbols that define the offsets to the static storage required by SCF devices. This area continues from V.USER defined in OS9DEFS. SCF devices must reserve this space for the SCF manager. The driver determines the storage reserved after this group.

Character Definitions

a group of symbols that are defined so that certain SCF devices can filter special characters. This is an appropriate place for adding your own SCF special characters.

File Descriptor Format

a group of symbols that describe the file manager's parameters. The actual total storage is declared in OS9Defs under the entry called Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PD.FST and PD.OPT fields. This is where SCF's definitions are located.

RBFDEFS

In this file are the definitions pertaining to random block file managers and random file devices. It contains the following groups of defined symbols:

> Random Block Path Descriptor Format
> State Flags
> Device Descriptor Format
> File Descriptor Format
> Segment List Entry Format
> Directory Entry Format
> Static Storage

You can use this file when writing random block file managers and devices. This is the appropriate place for adding your own RBF definitions.

Random Block Path Descriptor Format

a group of symbols that define the file descriptor offsets for RBF devices. The actual total storage is declared in OS9DEFS under Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PS.FST and PD.OPT fields. This is where RBF's definitions are located.

State Flags

a group of symbols that define the flags that OS-9 uses internally to mark the state of the disk buffer.

Device Descriptor Format

a group of symbols that define the format of the contents of sector zero of an RBF device. RBF uses this format to find the actual physical information on the device. OS-9 uses this information to fill in the drive table. This differs from SCF type devices in that the actual device information is kept on the media. The device descriptor in memory is then mainly used by the format program.

File Descriptor Format

a group of symbols that defines a format that is kept on disk and contains information on the file, for example, its size, segment list, and owner. RBF reads in this information and uses it when accessing a file. Whenever you modify the file, this sector is modified.

Segment List Entry Format

a group of symbols that define an entry in a file segment list. The actual list is composed of the beginning sector and the size (in number of sectors) of each segment of the file. Files that have extensions also have an additional segment for the segment list.

Directory Entry Format

two symbols that define a directory entry, the file name and the file descriptor sector address.

Static Storage

a group of symbols that define the size and format of the drive tables allocated by the driver. The drive tables begin at DRVBEG and continue to DRVMEM. Also V.NDRV is allocated before the tables and defines the number of drives and therefore the number of tables used by a driver. The rest of the static storage is defined in OS9DEFS and in the driver. You can use this information when writing your RBF device driver.

SYSTYPE

This file contains descriptions of the physical parameters of the various OS-9 systems. Some of those parameters are:

CPU Type Definitions
CPU Speed Definitions
Disk Controller Definitions
Clock Module Definitions
PIA Type Definitions
System Type Definitions
Disk Port Address
Disk Definition
Disk Parameters
Clock Port
I/O Port

You can use this file when writing or modifying drivers. All the necessary information is supplied on the file.

8 / Assembly-Language Programming Techniques

For your program to run correctly in the OS-9 environment, it must be position-independent, and all memory locations modified by the program (variables and data structures) must be in an area that OS-9 assigns at run time.

You have no control over which addresses OS-9 assigns at a load area or over where OS-9 assigns the program's variables. Because of the powerful 6809 instruction set and addressing modes, these rules do not force you into writing tricky or complex programs; rather, they require you to write programs in a specific way.

Your programs usually fall into one of three categories:

1. A subroutine or subroutine package. You must write your subroutines in position independent code. Data sections are usually a matter of coordination with the calling program, and OS-9 normally plays no direct role in this.

2. A program to be executed as an individual process (commands are of this type). You must use position independent code and receive data area parameters that delineate the assigned memory space.

3. Programs to be run on another, non-OS-9 computer.

Program Sections and Data Sections

If you run your program as a process (by means of the OS-9 Shell, fork system call, or execute system call), OS-9 assigns two separate and distinct memory areas. The program object code loads into one memory space in the form of a memory module. Variables and data structures load into the other space. The program's module header specifies the minimum permissible size for each area.

The distinction between these two spaces is extremely important. The data address counter is for the data area, and the instruction address counter is for the program area. The values of both counters are never absolute addresses. They are relative to the beginning of an OS-9-assigned address.

Program Area

The program area is a simple, continuously allocated memory space where OS-9 loads the program. For OS-9 to load the program, it must be in memory module format. *OS-9 Technical Information* contains a detailed description of memory modules and how they work. This manual assumes you are familiar with them.

The assembler generates programs that can consist of one or more memory modules. It writes them to the same file, and OS-9 loads them together. In assembly-language source programs, modules usually begin with a MOD pseudo-instruction and end with an EMOD pseudo-instruction. These take care of the header and module CRC generation for you.

Never modify the program area by the program itself, especially if the program is to be reentrant and/or placed in ROM. It can (and should) contain constants and constant tables, as long as they are not altered by the program.

Position Independent Mode

You do not know the absolute address of anything in the program until it is run. The 6809 position-independent addressing modes are based on "program counter relative addressing." All branch and long-branch instructions use this addressing mode.

  —>   Use BRA and LBRA instead of JMP; use BSR and LBSR instead of JSR extended or direct mode instructions (indexed is OK).

All load, store, arithmetic, and logical instructions can use the program-counter-relative (PCR) indexed addressing mode.
-> Do not use immediate addressing to load a register with an absolute address (instruction label name). Use PCR indexed addressing instead.

Many well-written programs use constant tables of addresses (often called dispatch tables or pointer tables). For the program to be position independent, these tables cannot contain absolute addresses. You must create tables of addresses that are relative to an arbitrary location. The routines that use the tables read the table entries and then add them to the absolute address of the arbitrary location. The sum is the run-time absolute address.

You can determine the absolute address of the arbitrary location by using PCR instructions (typically LEA). The choice of the common address is arbitrary, but two places may have specific advantages: the beginning address of the table (an index register probably will contain this address anyway); and the first byte of the module.

Making table entries relative to the start of the module is especially handy because the value of the assembler's instruction address counter is also relative to the beginning address of the module.

In the following example, a routine jumps to one of several subroutines, the relative addresses of which are contained in a table. The instructions pass the routine to a number in the B accumulator, which uses it as an index to select the routine.

```
begin       mod a,b,c,d,e,f start of module

   (various instructions)

dispat      leax table,pcr           get the absolute
                                      address of the table
            aslb                      multiply index by 2
                                      (two bytes/entry)
            ldd  b,x                  get contents of table
                                      entry
            leax    begin,pcr         get beginning
                                      address of module
            jmp     d,x               add relative address
                                      and go...
```

```
table          fdb      routine1
               fdb      routine2
               fdb      routine3
               fdb      routine4
```

In the following example the entries are relative to the beginning of the table instead of to the beginning of the module.

```
dispat         leax     table,pcr        get the absolute
                                         address of the table
               aslb                      multiply index by 2
               ldd      b,x              get routine offset
               jmp      d,x              add and go...

table          fdb                       routine1-table
               fdb                       routine2-table
               fdb                       routine3-table
               fdb                       routine4-table
```

The above example contains fewer instructions. It is also faster because the register already contained the reference address in a register; therefore, you can eliminate a LEAX instruction. This technique is also useful for accessing character strings, constants, complex data types, and so on.

Accessing the Data Area

The "minimum permanent storage size" entry of the module header specifies the size of the data area. A program may, however, occupy more than this minimum. OS-9 allocates memory in multiples of 256-byte pages, and it allocates all processes at least one page. The data area must be large enough for all the program's variables and data structures, plus a stack (at least 250 bytes) and space to receive parameters.

When OS-9 calls the process, it passes the bounds of the data area to the process in the Registers MPU. U contains the beginning address, and Y contains the ending address. It sets the Register SP to the ending address + 1, unless parameters were passed. It sets the direct page register to the page number of the beginning page.

In the assembly-language source program, you can assign storage in the data area with the RMB pseudo instruction, which uses the separate data address counter. You need to declare all variables and structures at the beginning of the program. Declare smaller, frequently used variables first. They usually fit in the first page, and you can access them with short, fast direct-page addressing instructions. Follow with larger items. You can address them in two ways:

1. If you maintain a Register throughout the program, use constant-offset-indexed addressing.

2. Part of the program's initialization routine can compute the actual addresses of the data structures and store them in pointer locations in the direct page. Obtain the addresses later with direct-page addressing mode instructions.

Note: You cannot use program-counter relative addressing to obtain addresses of objects in the data section, because the memory addresses assigned to the program section and the address section are not a fixed distance apart. Of course, immediate and extended addressing are not generally usable.

The following example illustrates the U relative technique.

```
                * declare variables

temp1           rmb 2
temp2           rmb 2
buf1            rmb 400
buf2            rmb 400
buf3            rmb 400
```

```
                         * clear each 400-byte buffer

        leax buf1,u              get address of buf1
        bsr clrbuf
        leax buf2,u              get address of buf2
        bsr clrbuf
        leax buf3,u              get address of buf3
        bsr clrbuf

                         * clear buffer subroutine
                         * X = address of buffer

        clrbuf   ldd #400        D = byte count
        cloop    clr ,x+          clear byte and advance pointer
                 subd #1         decrement count
                 bne cloop       loop if not done yet
                 rts
```

9 / Assembler Error Reporting

When the assembler detects an error, it displays an error message just before the line containing the error. If a statement has two or more errors, the assembler displays each error on a different line preceding the erroneous line.

If the -L option inhibits the assembler, the assembler still displays error messages and erroneous lines. The statistical summary displayed at the end of the assembly contains the total numbers of errors and warnings. The assembler writes the error messages, erroneous source lines, and the assembly summary to the assembler task's error and status path, which the shell may redirect. The assembler writes the listing to the output path, which it may redirect independently of the error messages. This is useful when a procedure file calls the assembler. Example:

    asm sourcefile − l 0 >same.listing >>save.errs

You can perform a quick assembly just to check for errors by calling the assembler with the listing and object code generation both disabled by the − L − O options. In this way you can find and correct many errors before displaying a lengthy list. Example:

    asm sourcefile − l − 0

The − E option turns off error-message display, but you can still detect lines containing errors by the presence of an E in the informational column of the listing line. You may want to use this option to generate a ''cleaner'' listing of a program known to have many errors.

Sometimes the assembler stops processing an erroneous line, and therefore you may not be able to detect additional errors on the same line; so make corrections carefully.

Explanation of Error Messages

Error messages consist of brief phrases that describe the error.

Syntax and Grammar Errors

Improperly constructing source statements can cause the following errors of syntax and grammar.

ADDRESS MODE. The addressing mode specified is not legal for the instruction.

BAD INSTR. The assembler does not recognize the instruction given in the source statement.

BAD LABEL. The statement's label contains an illegal character or does not begin with an alphabetical character.

] MISSING. A closing bracket is missing.

CONST DEF. The instruction requires a constant or an expression that is missing or in error.

INDEX REG. The instruction requires the name of an index register but none was found.

LABEL NOT ALLOWED. This type of statement cannot have a label.

NEEDS LABEL. The statement must have a label.

OUT OF RANGE. The destination (label) of the branch is too far to use a short-branch instruction.

REG NAM. The required register name is missing or misspelled.

REG SIZES. The registers specified in a TFR or EXG instruction are of different lengths.

Arithmetic Errors

Improper arithmetic or the use of improper arithmetic expressions can result in the following errors.

DIV BY 0. A division with a zero divisor occurred.

EXPR SYNTAX. The arithmetic instruction is illegally constructed or is missing an operand following an operator.

IN NUMBER. A constant number (decimal, hexadecimal, or binary) is too large or contains an illegal character.

MULT OVERFL. The result of a multiplication is more than 65,535 (two bytes).

PARENS. The expression contains an unequal number of right and left parentheses.

RESULT>255. The result of the expression is too large to fit in the 1-byte value used by the instruction.

Symbolic Name Errors

When symbolic names are improperly used, defined, or redefined, the following errors can occur.

PHASING. The statement's label had a different address during the first assembly pass. This usually happens when an instruction changes addressing modes, and thus its length, after the first pass because its operand becomes defined after the source line is processed. Usually the error occurs on all labels following the offending source line.

REDEFINED NAME. The label was defined previously in the program.

UNDEFINED NAME. The symbolic name was never defined in the program.

Assembler Operational Errors

Using the assembler incorrectly can cause the following errors.

CAN'T OPEN PATH. The file cannot be opened (source file) or created (object file).

INPUT PATH. The input path contains a read error.

MEMORY FULL. The symbol table is full. More memory is required to assemble the program.

OBJECT PATH. The object file contains a write error.

OPT LIST. The assembler command line or an OPT statement contains an illegal option or is missing an option.

Appendix A / Sample Command Lines

asm disk_crash

assembles the file disk_crash.
This command does not create a listing or an object file. It reports error to the standard error path and establishes 4K memory for symbols (asm default).

asm work.rec o #16k

assembles the file work.rec.
This command does not create a listing. It does create an object file with the name work.rec in the current commands directory. It reports errors to the standard output path and establishes 16K memory for symbols.

asm tyco 0 = /d0/cmds/tyco.obj 1 #16k

assembles the file tyco.
This command creates a listing directed at the standard output and an object file with the name tyco in the /d0/cmds directory. It reports errors to the listing path and establishes 16K memory for symbols.

asm it_works o,1 #16k >/p

assembles the file it_works.
This command creates a listing directed at /p and an object file with the name it_works in the current commands directory. It reports errors to the listing path and establishes 16K memory for symbols.

asm test_util 1,s,w72,d25 #10k

assembles the file test_util.
This command creates a listing directed at the standard output. The listing has 25-line pages and 72-column lines. It does not create an object file. It establishes 10K memory for symbols and creates a symbol table. It reports errors to the listing path.

asm /term i 1 o = d0/progs/woof

assembles input from the terminal.

This command creates a listing directed at the standard output and an object file with the name woof in the /d0 progs directory. It reports errors to the listing path and establishes 4K memory for symbols (asm default).

Appendix B / Error Messages Abridged

The assembler displays an error message for each error it detects. It displays the messages before the line in which the error occurs. You can suppress the display of error messages by using the − E command.

ADDRESS MODE. The specified addressing mode is not legal for the instruction.

BAD INSTR. The assembler does not recognize the instruction given in the source statement.

BAD LABEL. The statement's label contains an illegal character or does not begin with a letter.

] MISSING. A closing bracket is missing.

CAN'T OPEN PATH. The file cannot be opened.

CONST DEF. A constant expression is missing or in error.

DIV BY 0. A division with a zero divisor occurred.

EXPR SYNTAX. The arithmetic instruction is illegally constructed or is missing an operand following an operator.

INDEX REG. The name of an index register is required but none was found.

IN NUMBER. A constant number (decimal, hexadecimal, or binary) is too large or contains an illegal character.

INPUT PATH. The input path contains a read error.

LABEL NOT ALLOWED. The statement cannot have a label.

MEMORY FULL. The symbol table is full; create more memory to assemble the program.

MULT OVERFL. The result of a multiplication is more than 65,535.

NEEDS LABEL. The statement requires a label.

OBJECT PATH. The object file path contains a write error.

OPT LIST. An option is illegal or missing.

OUT OF RANGE. The destination of the branch is too far to use a short-branch instruction.

PARENS. The expression contains an unequal number of right and left parentheses.

PHASING. The value of the instruction address or data address counter was different during Pass 1; that is, an instruction changed addressing modes and length during Pass 2.

REDEFINED NAME. The label was defined previously in the program.

REG NAM. A register name is missing or misspelled.

REG SIZES. The registers specified in a TFR or EXG instruction are of different sizes.

RESULT>255. The result of the expression is too large to fit in the required byte.

Appendix C / Assembly Language Programming Examples

The following pages contain three assembly language programming examples. They are:

UpDn   -Program to convert input case to upper or lower.

P   -Parallel interface descriptor.

These programs are provided to give an example of what an assembly language program should be in the way of structure and form. They also provide the programmer with a guide to three of the main program types.

UPDN

UpDn — Assembly Language Programming Example

```
         *
         * this is a prosram to convert characters from
         *    lower to upper case (by usins the u option)
         * the method of passins the paramters throush
         *    os9 is used here (system calls)
         * to use type
         * "updn u(opt for lower to upper) <'input' > 'output'"
         *
                          nam   UpDn
         * file include in assembly
                          ifp1
                          use   /D0/defs/os9defs
                          endc
         *
         * OS-9 System Definition File Included
         *
                          opt   l
                          ttl   Assembly Lansuase Example
         *
         * module header macro
         *
   0000 87CD005D          mod   UDSIZ,UDNAM,TYPE,REVS,START,SIZE
   000D 757064EE   UDNAM  fcs   /updn/      module name for memory
   0011             TYPE   set   PRGRM+OBJCT mod type
   0081             REVS   set   REENT+1     mod revision
         *
         * storase area for variables
         *
 D 0000             TEMP   rmb   1           temp storase for read
 D 0001             UPRBND rmb   1           storase for upperbound
 D 0002             LWRBND rmb   1           storase for lowerbound
 D 0003                    rmb   250         storase for stack
 D 00FD                    rmb   200         storase for parameters
 D 01C5             SIZE   equ   .           end of data area
         *
         * actual code starts here
         * x resister is pointins to start of parameter area
         * y resister is pointins to end of parameter area
         * this is how to set a parameter that is passed on
         *    the command line and where to look for it
         *
```

```
0011                START  equ  *          start of executable
0011  A680          SRCH   lda  ,x+        search parameter area

0013  84DF                 anda #$df       make upper case
0015  8155                 cmpa #'U        see if a U was input
0017  270E                 beq  UPPER      branch to set uppercase
0019  810D                 cmpa #$0d       see if a carriage return
001B  26F4                 bne  SRCH       go get another char
      *
      * fall through to set upper to lower bounds
      *
001D  8641                 lda  #'A        set lower bound
001F  9702                 sta  LWRBND     set it in storage area
0021  865A                 lda  #'Z        get upper bound
0023  9701                 sta  UPRBND     set it in storage area
0025  2008                 bra  START1     go to start of code
      *
      * set lower to upper bounds
      *
0027  8661          UPPER  lda  #'a        get lower bound
0029  9702                 sta  LWRBND     set it in storage
002B  867A                 lda  #'z        get upper bound
002D  9701                 sta  UPRBND     set it in storage
      *
      * converting code
      * this part uses the I$READ and
      *   the I$WRIT system calls
      * read the systems programmers manual
      *   for information relating to them
002F  30C4          START1 leax temp,u     get storage address
0031  8600                 lda  #0         standard input
0033  108E0001             ldy  ##01       number of characters
0037  103F89        LOOP   os9  I$READ     do the read
003A  2515                 bcs  EXIT       exit if error
003C  D600                 ldb  TEMP       get character read
003E  D102                 cmpb WRBND      test char bound
0040  2506                 blo  WRITE      branch if out
0042  D101                 cmpb UPRBND     test char bound
0044  2202                 bhi  WRITE      branch if out
0046  C820                 eorb #$20       flip case bit
0048  D700          WRITE  stb  TEMP       put it in storage
004A  4C                   inca            reg 'a' stand output
004B  103F8A               os9  I$WRITE    write the character
004E  4A                   deca            return to stand input
004F  24E6                 bcc  LOOP       get char if no error
0051  C1D3          EXIT   cmpb #E$EOF     is it an EOF error
0053  2601                 bne  EXIT1      not eof, leave carry
0055  5F                   clrb            clear carry, no error
0056  103F06        EXIT1  os9  F$EXIT     error returned, exit
0059  2604D9               emod            last command
005C                UDSIZ  equ  *          size of program
                    END
```

P - Device Descriptor for "P"

```
                              nam  P

                              ifp1
                              endc

                              ttl  Device Descriptor for "P"
        *****************
        *   PRINTER device module
        *
0000 87CD0035              mod  PRTEND,PRTNAM,DEVIC+OBJCT,
                                REENT+1,PRTMGR,PRTDRV
000D 02                    fcb  WRITE     mode
000E FF                    fcb  $FF
000F E040                  fcb  A,P       port address
0011 18                    fcb  PRTNAM-*-1option byte count
0012 00                    fcb  DT.SCF    Device Type: SCF

        * Default Path options

0013 00                    fcb  0         case=UPPER and lower
0014 00                    fcb  0         backspace=BS char only
0015 01                    fcb  1         delete=CRLF
0016 00                    fcb  0         no auto echo
0017 01                    fcb  1         auto line feed on
0018 00                    fcb  0         no nulls after CR
0019 00                    fcb  0         no page pause
001A 42                    fcb  66        lines per page
001B 08                    fcb  C$BSP     backspace char
001C 18                    fcb  C$DEL     delete line char
001D 0D                    fcb  C$CR      end of record char
001E 00                    fcb  0         no end of file char
001F 04                    fcb  C$RPRT    reprint line char
0020 01                    fcb  C$RPET    dup last line char
0021 17                    fcb  C$PAUS    pause char
0022 00                    fcb  0         no abort character
0023 00                    fcb  0         no interrupt character
0024 5F                    fcb  '_        backspace echo char
0025 07                    fcb  C#BELL    line overflow char
0026 01                    fcb  PIASID    Printer Type
0027 00                    fcb  0         undefined baud rate
```

```
0028 0000               fcb    0          no echo device
002A D0        PRTNAM   fcs    "P"        device name
002B B0                 fcs    "0"        room for name patching
002C 5348C6    PRTMGR   fcs    "SCF"      file manager
002F 5049C     PRTDRV   fcs    "PIA"      driver

0032 A9B118             emod

0035           PRTEND   EQU    *
                        END
```

Appendix D / 6809 Instructions And Addressing Modes

	DIRECT	EXTEND	INDEX	IMMED	ACCUM	INHER	RELAT	REGIS
ABX						X		
ADCA	X	X	X	X				
ADCB	X	X	X	X				
ADDA	X	X	X	X				
ADDB	X	X	X	X				
ADDD	X	X	X	X				
ANDA	X	X	X	X				
ANDB	X	X	X	X				
ANDCC				X				
ASL	X	X	X					
ASLA						X		
ASLB						X		
ASR	X	X	X					
ASRA						X		
ASRB						X		
(L)BCC							X	
(L)BCS							X	
(L)BEQ							X	
(L)BGE							X	
(L)BGT							X	
(L)BGI							X	
(L)BHS							X	
BITA	X	X	X	X				
BITB	X	X	X	X				
(L)BLE							X	
(L)BLO							X	
(L)BLS							X	
(L)BLT							X	
(L)BMI							X	
(L)BNE							X	
(L)BPL							X	
(L)BRA							X	
(L)BRN							X	
(L)BSR							X	
(L)BVC							X	
(L)BVS							X	
CLR	X	X	X		X			
CMPA	X	X	X	X				
CMPB	X	X	X	X				
CMPD	X	X	X	X				
CMPS	X	X	X	X				
CMPU	X	X	X	X				
CMPX	X	X	X	X				
CMPY	X	X	X	X				
COM	X	X	X		X			
CWAI				X				
DAA						X		
DEC	X	X	X	X	X			
EORA	X	X	X	X				
EORB	X	X	X	X				

	DIRECT	EXTEND	INDEX	IMMED	ACCUM	INHER	RELAT	REGIS
EXG								X
INC	X	X	X		X			
JMP	X	X	X					
JSR	X	X	X					
LDA	X	X	X	X				
LDB	X	X	X	X				
LDD	X	X	X	X				
LDS	X	X	X	X				
LDU	X	X	X	X				
LDX	X	X	X	X				
LDY	X	X	X	X				
LEAS			X					
LEAU			X					
LEAX			X					
LEAY			X					
LSL	X	X	X		X			
LSR	X	X	X		X			
MUL						X		
NEG	X	X	X		X			
NOP						X		
ORA	X	X	X	X				
ORB	X	X	X	X				
ORCC				X				
PSHS								X
PSHU								X
PULS								X
PULU								X
ROL	X	X	X		X			
ROR	X	X	X		X			
RTI						X		
RTS						X		
SBCA	X	X	X	X				
SBCB	X	X	X	X				
SEX						X		
STA	X	X	X					
STB	X	X	X					
STS	X	X	X					
STU	X	X	X					
STX	X	X	X					
STY	X	X	X					
SUBA	X	X	X	X				
SUBB	X	X	X	X				
SUB	X	X	X	X				
SWI						X		
SWI2						X		
SWI3						X		
SYNC						X		
TFR								X
TST	X	X	X		X			

Appendix E / ASCII Character Set

SYMBOL	HEX VALUE	SYMBOL	HEX VALUE	SYMBOL	HEX VALUE
NUL	00	+	2B	V	56
SOH	01	'	2C	W	57
STX	02	–	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	3	]	5D
BS	08	3	33	^	5E
HT	09	4	34	—	5F
LF	0A	5	35	`	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
SO	0E	9	39	d	64
SI	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC(XON)	11	<	3C	g	67
DC2	12	=	3D	h	68
DC3(XOFF)	13	>	3E	i	69
DC4	14	?	3F	j	6A
NAK	15	(a	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
SP	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B

SYMBOL	HEX VALUE	SYMBOL	HEX VALUE	SYMBOL	HEX VALUE
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	30	U	55		

138

OS-9 Interactive Debugger

1 / Introduction

DEBUG is an interactive debugger that aids in diagnosing systems and testing 6809 machine-language programs. You can also use it to gain direct access to the computer's memory. DEBUG's calculator mode can simplify address computation, radix conversion, and other mathematical problems.

Calling DEBUG

DEBUG is supplied on your OS-9 system disk. When the screen shows the OS-9 prompt, call DEBUG by typing:

DEBUG (ENTER)

Basic Concepts

DEBUG responds to 1-line commands entered from the keyboard. The screen shows the DB: prompt when DEBUG expects a command.

Terminate each line by typing (ENTER). Correct a typing error by using the backspace (left arrow) key, or delete the entire line by typing (X) while pressing (CLEAR).

Each command starts with a single character, which may be followed by *text* or by one or two arithmetic *expressions*, depending on the command. You may use upper- or lower-case letters or a mixture. When you use the (SPACEBAR) to insert a space before a specific *expression*, the screen shows the results in hexadecimal and decimal notation. Example:

In the calculator mode, obtain hexadecimal and decimal notation for the hexadecimal expression A + 2:
You Type: (SPACEBAR) (A)(+)(2)
Screen Shows: DB: A+2
$000C #00012

Note: In the examples in this manual, general instructions are followed by specific typing instructions and then by what the screen shows. In some cases, examples will follow the explanation of more than one command. Be sure to execute these examples in the exact order in which they are given so that you will obtain the specified display on your screen.

2 / Expressions

DEBUG's integral expression interpreter lets you type simple or complex expressions wherever a command calls for an input value. DEBUG expressions are similar to those used with high-level languages such as BASIC, except that some extra operators and operands are unique to DEBUG.

Numbers in expressions are 16-bit unsigned integers, which are the 6809's "native" arithmetic representation. The allowable range of numbers is 0 to 65535. Two's complement addition and subtraction is performed correctly, but will print out as large positive numbers in decimal form.

Some commands require byte values, and the screen shows an error message if the result of an expression is too large to be stored in a byte, that is, if the result is greater than 255. Some operands, such as individual memory locations and some registers, are only one byte long, and they are automatically converted to 16-bit "words" without sign extension.

Spaces, other than a space at the beginning of a command, do not affect evaluation; use them as necessary between operators and operands to improve readability.

Constants

Constants can be in base 2 (binary), base 10 (decimal), or base 16 (hexadecimal). Binary constants require the prefix %; decimal constants require the prefix #. All other numbers are assumed to be hexadecimal and may have the prefix $. Examples:

Decimal	Hexadecimal	Binary
#100	64	%1100110
#255	FF	%11111111
#6000	1770	%1011101110000
#65535	FFFF	%1111111111111111

143

You may also use character constants. Use a single quote (') for 1-character constants and a double quote (") for 2- character constants. These produce the numerical value of the ASCII codes for the character(s) that follow. Examples:

'A   = $0041
'0   = $0030
"AB   = $4142
"99   = $3939

Special Names

Dot (.) is DEBUG's current working address in memory. You can examine it, change it, update it, use it in expressions, and recall it. Dot eliminates a tremendous amount of memory address typing.

Dot-Dot (..) is the value of Dot before the last time it was changed. Use Dot-Dot to restore Dot from an incorrect value or use it as a second memory address.

Register Names

Specify Registers MPU with a colon (:) followed by the mnemonic name of the register. Examples:

:A   Accumulator A
:B   Accumulator B
:D   Accumulator D
:X   X Register
:Y   Y Register
:U   U Register
:DP   Direct Page Register
:SP   Stack Pointer
:PC   Program Counter
:CC   Condition Codes Register

The values returned are the test program's registers, which are "stacked" when DEBUG is activve. One-byte registers are promoted to a word when used in expressions.

Note: When a breakpoint interrupts a program, the Register SP points at the bottom of the Register MPU stack.

Operators

"Operators" specify arithmetic or logical operations to be performed within an expression. DEBUG executes operators in the following order: (1) − , negative numbers; (2) & and !, logical AND and OR; (3) * and /, multiplication and division; (4) + and − , addition and subtraction. Operators in a single expression having equal precedence, for example, + and − , are evaluated left to right. You can use parentheses, however, to override precedence.

Forming Expressions

An *expression* is composed of any combination of constants, register names, special names, and operators. The following are valid expressions:

$$\#1024 + \#128$$

$$:X - :Y - 2$$

$$. + 20$$

$$:Y*(:X + :A)$$

$$:U\ \&\ FFFE$$

Indirect Addressing

Indirect addressing returns the data at the memory address using a value (expression, constant, special name, and so on) as the memory address. The two DEBUG indirect addressing modes are:

<expression>

returns the value of a memory byte using *expression* as an address.

[expression]

returns the value of a 16-bit word using *expression* as an address.

Examples:

    <200>

returns the value of the byte at Address 200.

    [:X]

returns the value of the word pointed to by Register X.

    [. + 10]

returns the value of the word at Address Dot plus 10.

3 / Debug Commands

Calculator Commands

(SPACEBAR)<*expression*> (ENTER)

evaluates the expression and displays the results in both hexa-decimal and decimal. Examples:

> **You Type:** (SPACEBAR)5000+200 (ENTER)
> **Screen Shows:** DB: 5000+200
> $5200 #20992

> **You Type:** (SPACEBAR)8800/2 (ENTER)
> **Screen Shows:** DB: 8800/2
> $4400 #17408

> **You Type:** (SPACEBAR)#100+#12 (ENTER)
> **Screen Shows:** DB: #100+#12
> $0070 #00112

These commands also convert values from one representation to another. Examples:

Convert a binary expression to hexadecimal and decimal:
> **You Type:** (SPACEBAR)%11110000 (ENTER)
> **Screen Shows:** DB: %11110000
> $00F0 #00240

Convert a 1-character constant to hexadecimal ASCII and decimal ASCII:
> **You Type:** (SPACEBAR)'A (ENTER)
> **Screen Shows:** DB: 'A
> $0041 #00065

Convert a decimal expression to hexadecimal and decimal:
> **You Type:** (SPACEBAR)#100 (ENTER)
> **Screen Shows:** DB: #100
> $00C4 #00100

You can also use indirect addressing to look at memory without changing Dot. Example:
> **You Type:** (SPACEBAR). (ENTER)
> **Screen Shows:** DB: .
> $004F #00079

147

In addition, you can use indirect addressing to simulate 6809 indexed or indexed indirect instructions. The following example is the same as the assembly language syntax [D,Y].

    You Type: (SPACEBAR)[: D + : Y] (ENTER)
    Screen Shows: DB : [: D + : Y]
                         $0110   *00272

Dot and Memory Examine and Change Commands

displays the current value of Dot (the current working memory address) and its contents. Example:

    You Type: . (ENTER)
    Screen Shows: DB : .
                        2201 B0

The present value of Dot is 2201, and B0 is the contents of memory location 2201.

(ENTER)

increments Dot and displays its new value and contents. Example:

"Step through" sequential memory locations:

    You Type: (ENTER)
    Screen Shows: DB :
                        2202 05
    You Type: (ENTER)
    Screen Shows: DB :
                        2203 C2
    You Type: (ENTER)
    Screen Shows: DB :
                        2204 82

--

backs up Dot one address and displays its value and contents. Example:

Display the current value of Dot:
    You Type: ↓ (ENTER)
    Screen Shows: DB : ↓
                2204 82

Back up one address and display its value and contents:
    You Type: - (ENTER)
    Screen Shows: DB : -
                2203 C2

Back up another address and display its value and contents:
    You Type: - (ENTER)
    Screen Shows: DB : -
                2202 05

. expression

changes the value of Dot. This command evaluates the specified *expression*, which becomes the new value for Dot. Example:

    You Type: ↓ 500 (ENTER)
    Screen Shows: DB : ↓ 500
                0500 12

..

restores the last value of Dot. Example:

Display the current value of Dot and its contents:
    You Type: ↓ (ENTER)
    Screen Shows: DB : ↓
                1000 23

Change the value of Dot:
    You Type: ↓ 2000 (ENTER)
    Screen Shows: DB : ↓ 2000
                2000 9C

Restore the last value of Dot:
> **You Type:** , , (ENTER)
> **Screen Shows:** DB : , , (ENTER)
>   1000 23

= *expression*

changes the contents of Dot. This command evaluates the *expression* and stores the results at Dot. It then increments Dot and displays the next address and contents.

This command also checks Dot after the new value is stored to make sure it changed to the correct value. If it did not, the screen shows an error message. This happens when you attempt to alter non-RAM memory. In particular, the registers of many 6800-family interface devices (such as PIAs and ACIAs) do not read the same as when written to.
Example:

Display the current value of Dot and its contents:
> **You Type:** , (ENTER)
> **Screen Shows:** DB : ,
>   2203 C2

Change the contents of Dot:
> **You Type:** =FF (ENTER)
> **Screen Shows:** DB : =FF
>   2204 01

Show that the contents of Dot have changed:
> **You Type:** - (ENTER)
> **Screen Shows:** DB : -
>   2203 FF

Warning: This command can change any memory location. You can destroy DEBUG, the program under test, or OS-9 if you incorrectly change any of their memory areas.

Register Examine and Change Commands

You can use any of several forms of the colon (:) register command to examine one or all registers or to change a specific register's contents.

The "registers" affected by these commands are actually "images" of the register values of the program under test, which are stored on a stack when the program is not running. Although a "dummy" stack is established automatically when you start DEBUG, use the E command to give the register images valid data before using the G command to run the program. The "registers" are valid after breakpoints are encountered and are passed back to the program upon the next G command.

Note:
1. If you change the Register SP, you move your stack and the other register contents change.
2. Bit 7 of Register CC (the E flag) must always be set for the G command to work. If it is not set, DEBUG does not return to the program correctly.

: *register*

displays the contents of a specific *register*. The contents are in hexadecimal. Examples:

You Type: : PC **(ENTER)**
Screen Shows: DB : :PC
                          C499

You Type: : B **(ENTER)**
Screen Shows: DB : :B
                          007E

You Type: : SP **(ENTER)**
Screen Shows: DB : :SP
                          42FD

151

:

displays all *registers* and their contents. Example:

You Type: : (ENTER)
Screen Shows: DB : :

               PC=B265 A=01 B=0B CC=80
               DP=0C
               SP=0CF4 X=FF0D Y=000B
               U=00AE

:*<register>* *<expression>*

assigns a new value to a *register*. DEBUG evaluates the *expression* and stores it in the specified *register*. When you name 8-bit registers, the value of the *expression* must fit in a single byte. If it does not, the screen shows an error message, and the register does not change. Examples:

You Type: : X #4096 (ENTER)
Screen Shows: DB : :X #4096

Breakpoint Commands

The breakpoint capabilities of DEBUG let you specify addresses where you wish to suspend execution of the program under test and reenter DEBUG. When you encounter a breakpoint, the screen shows the values of the Registers MPU and the DB: prompt.

After a breakpoint is reached, you can examine or change registers, alter memory, and resume program execution. You may insert breakpoints at up to 12 addresses.

You can insert breakpoints by using the 6809 SWI instruction, which interrupts the program and saves its complete state on the stack. DEBUG automatically inserts and removes SWI instructions at the right times; so you do not "see" them in memory.

Because the SWIs operate by temporarily replacing an instruction OP code, there are three restrictions on their use:

1. You cannot use breakpoints in programs in ROM.

2. You must locate breakpoints in the first byte (OP code) of the instruction.

3. You cannot utilize the SWI instruction in user programs for other purposes. (You can use SWI2 and SWI3.)

When you encounter the breakpoint during execution of the program under test, reenter DEBUG by typing <:><*register name*>. The screen shows the program's register contents.

B

displays all present breakpoint addresses.

B *<expression>*

inserts a breakpoint at a specified expression.

Examples:

Insert a breakpoint at the specified expression:
  You Type: B 1C00 (ENTER)
  Screen Shows: DB: B 1C00

Insert another breakpoint at the specified expression:
  You Type: B 4FD3 (ENTER)
  Screen Shows: DB: B 4FD3

Display the current value of Dot and its contents:
  You Type: . (ENTER)
  Screen Shows: DB: .
         1277 39

Insert the breakpoints at Dot:
  You Type: B . (ENTER)
  Screen Shows: DB: B .

Display all present breakpoint addresses:
You Type: B (ENTER)
Screen Shows: DB: B
                1C00 4FD3 1277

K

kills (removes) all breakpoints.

K *<expression>*

kills a breakpoint at the specified *expression*. Examples:

Display all present breakpoint addresses:
You Type: B (ENTER)
Screen Shows: DB: B
                1C00 4FD3 1277

Kill a breakpoint at the address specified by the *expression*:
You Type: K 4FD3 (ENTER)
Screen Shows: DB: K 4FD3

Display all present breakpoint addresses:
You Type: B (ENTER)
Screen Shows: DB: B
                1C00 1277

Kill all breakpoints:
You Type: K (ENTER)
Screen Shows: DB: K

Display all present breakpoint addresses:
You Type: B (ENTER)
Screen Shows: DB: B

Program Setup and Run Commands

E *module name*

prepares DEBUG for testing a specific program module.

This command's function is similar to that of the OS-9 Shell in starting a program. It does not, however, redirect I/O or override (#) memory size. The E command sets up a stack, parameters, registers, and data memory area in preparation for executing the program to be tested. The G command starts the program.

Note: This command allocates program and data area memory as appropriate. The new program uses DEBUG's current standard I/O paths, but can open other paths as necessary. In effect, DEBUG and the program become coroutines.

This command is acknowledged by a register dump showing the program's initial register values. The G command begins program execution. The E command sets up the Registers MPU as if you had just performed an F$CHAIN service request as shown below:

DP,U   low

direct page
data area
parameter area

X,S   high
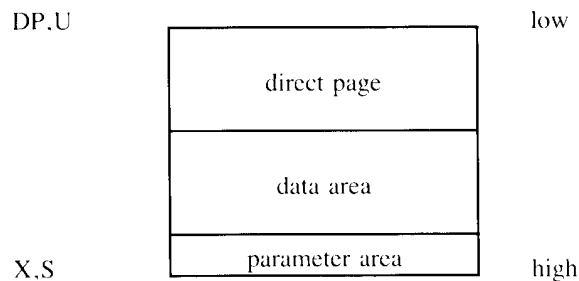
$$
\begin{aligned}
D &= \text{Parameter area size} \\
PC &= \text{Module entry point absolute address} \\
CC &= (F=0),\ (I=0)\ \text{Interrupts disabled}
\end{aligned}
$$

Example:

Display the program's initial register values:
> **You Type:** E myprogram (ENTER)
> **Screen Shows:** DB: E myprogram
>
>   SP   CC   A   B DP
>   X   Y   PC
>   ØCF3 C8   ØØ Ø1 ØC
>   ØCFF ØDØØ 9214

G

goes to (resumes) program execution after a breakpoint. If a breakpoint exists at the present program counter address, that breakpoint is not inserted so that it is not immediately reexecuted. A loop must contain at least two breakpoints if execution is to be suspended each time through the loop.

Note: The E command is usually used before the first G command to set up the program to be tested. DEBUG initially sets up a default stack; so G *expression* can be used to start a program using the results of the *expression* as a starting address. Examples:

> DB: G 4C00
> DB: G :PC + 100
> DB: G [.]

L *module name*

links to the module. If successful, it sets Dot to the address of the first byte of the program and displays it. You can use L to find the starting address of an OS-9 memory module.

Example:

Link to the module FPMATH:
> **You Type:** L FPMATH (ENTER)
> **Screen Shows:** DB: LFPMATH
>   ECØØ 87

Utility Commands

C <*expression1*> <*expression2*>

performs a "walking bit" memory test and clears all memory between the two evaluated addresses. *Expression1* gives the starting address, and *expression2* gives the ending address, which must be higher. If any bytes fail the test, this command displays their address. Of course, you can test and clear only RAM memory.

Warning: This command can be dangerous. Be sure which memory address you are clearing.

Examples:

Clear all memory between Addresses 2000 and 15FF:
>   **You Type:** C 15FF 2000 (ENTER)
>   **Screen Shows:** DB: C 15FF 2000
>   17E4
>   17E7

The screen's display of 17E4 and 17E7 indicates bad memory at those addresses.

Clear all memory between the last value of Dot and Address FF.
>   **You Type:** C . .+FF (ENTER)
>   **Screen Shows:** DB: C . .+FF

The screen shows a blank line following the command line, which indicates good memory.

M <*expression1*> <*expression2*>

produces a screen-sized tabular display of memory contents in both hexadecimal and ASCII form.

The starting address of each line is on the left, followed by the contents of the subsequent memory locations. On the far right is the ASCII representation of the same memory locations.

Periods are substituted for nondisplayable characters. The high order bit is ignored for the display of the ASCII character.

157

S *<expression1>* *<expression2>*

searches an area of memory for a 1- or 2-byte pattern, beginning at the present Dot address. *Expression1* is the ending address of the search, and *expression2* is the data for which to search. If *expression2* is less than 256, a 1-byte comparison is used; if it is greater than 256, a 2-byte comparison is used. If a matching pattern is found, Dot is set to its address, which is displayed. If a matching pattern is not found, the screen shows the DB: prompt.

$ (ENTER)

calls the OS-9 Shell, which responds with prompts for one or more command lines.

$ *Shell Command*

executes the command and returns to DEBUG.

Also use $ to call the system utility programs and the Interactive Assembler from within DEBUG. Examples:

> **You Type:** $DIR (ENTER)
> **Screen Shows:** DB: $DIR
>   DIRECTORY OF . 00:00:21
> OS9 BOOT   CMDS   SYS
>   DEFS   STARTUP   OLDFILE
> NEWFILE   BUSINESS   FILE1

Q

quits (leaves) DEBUG and returns to the OS-9 Shell. Example:

> **You Type:** Q (ENTER)
> **Screen Shows:** DB: Q
>   OS9:

4 / Using Debug

You use DEBUG primarily to test system memory and I/O devices, to "patch" the operating system or other programs, and to test hand-written or compiler-generated programs.

Sample Program

The simple assembly-language program shown below illustrates the use of DEBUG commands. This program prints "HELLO WORLD" and then waits for a line of input.

```
                        NAM            EXAMPLE
                        USE    /D0/DEFS/OS9DEFS
              * Data Section
0000                    ORG    0
0000          LINLEN    RMB    2            LINE LENGTH
0002          INPBUF    RMB    80           LINE INPUT BUFFER
0052                    RMB    50           HARDWARE STACK
00E7          STACK     EQU    .-1
00E8          DATMEM    EQU    .            DATA AREA MEMORY SIZE

              * Program Section
000  87CD0047           MOD    ENDPGM,NAME,$11,$81,ENTRY,DATMEM
000D 4558414D  NAME     FCS    /EXAMPLE/    MODULE NAME STRING

0014          ENTRY     EQU    *            MODULE ENTRY POINT
0014 308D0020           LEAX   OUTSTR,PCR   OUTPUT STRING ADDRESS
0018 108E000C           LDY    #STRLEN      GET STRING LENGTH
001C 8601               LDA    #1           STANDARD OUTPUT PATH
001E 103F8C             OS9    I$WRITLN     WRITE THE LINE
0021 2512               BCS    ERROR        BRA IF ANY ERRORS
0023 3042               LEAX   INPBUF,U     ADDR OF INPUT BUFFER
0025 108E0050           LDY    #80          MAX OF 80 CHARACTERS
0029 8600               LDA    #0           STANDARD INPUT PATH
002B 103F8B             OS9    I$READLN     READ THE LINE
002E 2505               BCS    ERROR        BRA IF ANY I/O ERRORS
0030 09F00              STY    LINLEN       SAVE THE LINE LENGTH
0033 C600               LDB    #0           RETURN WITH NO ERRORS
0035 103F06   ERROR     OS9    F$EXIT       TERMINATE THE PROCESS
0038 48454C4C OUTSTR    FCC    /HELLO WORLD/ OUTPUT STRING

0043 0D                 FCB    $0D          END OF LINE CHARACTER
000C          STRLEN    EQU    *-OUTSTR     STRING LENGTH
0044 268A06             EMOD                END OF MODULE
0047          ENDPGM    EQU    *            END OF PROGRAM
                        END
```

A Session With DEBUG

The following example illustrates how to use DEBUG with the program on the previous page. (The actual RAM address may vary depending on your computer's installation of OS-9.)

OS9:DEBUG #2K

```
Interactive Debugger
DB: $LOAD /D1/EXAMPLE
DB: L EXAMPLE
A900  87

DB:
     A900 87
DB:  M . . +44          (dump program on display)
A900  87CD0047000D1181 ...G....
A908  6F00140084455841 O....EXA
A910  4D504CC5308D0020 MPL.0..
A918  108E000C8601103F .......?
A920  8C25123042108E00 .%.0B...
A928  508600103F8B2505 P...?.%.
A930  109F00C600103F06 ......?.
A938  48454C4C4F20574F HELLO WO
A940  524C440DDB72DDFF RLD..R..

DB:  E EXAMPLE          (prepare to run program)
     SP   CC  A  B  DP  X     Y     U     PC
     0DF3 C8  00 01 0D  0DFF  0E00  0D00  9214
DB:
     A900  87
DB:  B . +2E            (set breakpoint at address A92E)
DB:  G                  (run program)
HELLO WORLD
hello computer
(ENTER)

     BKPT:              (breakpoint encountered)
     SP   CC  A  B  DP  X     Y     U     PC
     0DF3 C0  00 01 0D  0D02  0D00  0D00  922E
DB:  M :0  :U +20       (display register area)
0A00  00010D020000000C ........
0A08  0CF400004C000000 ....L...
```

```
0A10  00000087CD002400 .....$.
0A18  0D11810C001201C2 .......
0A20  000000000FF00FF

DB:    . :U+2
       0A02  68
DB:
       0A03  65
DB:
       0A04  6C
DB:
       0A05  6C
DB:    Q

OS9:
```

Patching Programs

To "patch" a program (to change its object code), follow these five steps:

1. Load the program into memory using OS-9's LOAD command.

2. Link to and change the program in memory using DEBUG's L and = commands.

3. Save the new, patched version of the program on a disk file using OS-9's SAVE command.

4. Update the program module's CRC check value using OS-9's VERIFY command. Be sure to use the U option.

5. Set the module's execute status using OS-9's ATTR command.

Step 4 is unique to OS-9 (as compared with other operating systems) and often overlooked. However, it is essential because OS-9 refuses to load the patched program into memory until its CRC check value is updated and correct.

The example that follows shows how the program on page 00 is "patched." In this case the LDY #80 instruction is changed to LDY #32.

```
OS9: DEBUG                    (call DEBUG)
Interactive Debugger
DB: $LOAD EXAMPLE             (call OS-9 to load program)
DB: L EXAMPLE                 (set dot to beg addr of program)
    2000  87                  (actual address will vary)
DB: . . + 28                  (add offset of byte to change*)
    2028  50                  (current value is 00)
DB  = #32                     (change to decimal 12)
    2028  10                  (change confirmed)
DB: $SAVE TEMP EXAMPLE        (save on file called "TEMP")
DB: $VERIFY U <TEMP>          (update CRC and copy to "NEWEX")
    NEWEX
DB: $ATTR NEWEX E PE          (set execution status)
DB: $DEL TEMP                 (delete temporary file)
DB: q                         (exit DEBUG)
```

Patching OS-9 Component Modules

Patching modules that are part of OS-9 (modules contained in the OS-9 Boot file) is a bit trickier than patching a regular program because you must use the COBBLER and OS-9GEN programs to create a new OS-9 Boot file. The example below shows how an OS-9 "device descriptor" module is permanently patched, in this case to change the upper-case lock of the device /TERM from on to off. This example assumes that a blank freshly formatted diskette is in Drive 1 (/D1).

Caution: Always use a copy of your OS-9 System Disk when patching, in case something goes wrong.

```
OS9: DEBUG (ENTER)            (call DEBUG)
Interactive Debugger
DB: L TERM (ENTER)            (set dot to addr of TERM module)
    CA82  87                  (actual address will vary)
DB: . . + 13 (ENTER)          (add offset of byte to change*)
    CA95 01                   (current value is 01)
DB: = 1 (ENTER)               (change value to 01 for "OFF")
    CA96 01
DB: - (ENTER)                 (move back one byte)
CA95  00                      (change confirmed)
DB: Q (ENTER)                 (exit DEBUG)
OS9: COBBLER /D1 (ENTER)      (write new bootfile on /D1)
```

OS9: VERIFY </D1/OS9BOOT >/D01/TEMP U (ENTER)
                                      (update CRC value)
OS9:DEL /D1/OS9BOOT (ENTER)   (delete old boot file)
OS9:COPY /D0/TEMP/D1/OS9BOOT
                                      (install updated boot file)

Then you can use the Dsave command to build a new systems
disk.

Appendix / Debug Command Summary

(SPACEBAR) *expression*	Evaluate; display in hexadecimal and decimal.

Dot Commands

.	Display Dot address and contents.
..	Restore last DOT, display address and contents.
. *expression*	Set Dot to result, display address and contents.
= *expression*	Set memory at Dot to result.
—	Decrement Dot, display address and contents.
(ENTER)	Increment Dot, display address and contents.

Register Commands

:	Display all register contents.
:*register*	Display specific register contents.
:*register* *expression*	Set register to result.

Program Setup and Run Commands

E *module name*	Prepare for execution.
G	Go to program.
G *expression*	Go to program at result address.
L *module name*	Link to module named, display address.

Breakpoint Commands

B	Display all breakpoints.
B *expression*	Set breakpoint at result address.
K	Kill all breakpoints.
K *expression*	Kill breakpoint at result address.

Utility Commands

M *expression1*	Display memory dump in tabular form. *expression2*
C *expression1* *expression2*	Clear and test memory
S *expression1* *expression2*	Search memory for pattern
$ *text*	Call OS-9 Shell
Q	Quit (exit)DEBUG

Error Codes

DEBUG detects several types of errors and displays a corresponding error message and code number in decimal notation. The various codes and descriptions are listed below. Error codes other than those listed are standard OS-9 error codes returned by various system calls.

0   ILLEGAL CONSTANT: The expression included a constant that had an illegal character or that was greater than 65,535.

1   DIVIDE BY ZERO: A division was attempted using a divisor of zero.

2   MULTIPLICATION OVERFLOW: The product of the multiplication was greater than 65,535.

3   OPERAND MISSING: An operator was not followed by a legal operand.

4   RIGHT PARENTHESIS MISSING: Parentheses were misnested.

5   RIGHT BRACKET MISSING: Brackets were misnested.

6   RIGHT ANGLE BRACKET MISSING: A byte-indirect was misnested.

7   INCORRECT REGISTER: A misspelled, missing, or illegal register name followed the colon.

8   BYTE OVERFLOW: An attempt was made to store a value greater than 255 in a byte-sized destination.

9   COMMAND ERROR: A command was misspelled, missing, or illegal.

10   NO CHANGE: The memory location did not match the value assigned to it.

11   BREAKPOINT TABLE FULL: The maximum number of 12 breakpoints already exist.

12   BREAKPOINT NOT FOUND: No breakpoint exists at the address given.

13   ILLEGAL SWI: An SWI instruction was encountered in the user program at an address other than a breakpoint.

TEXT EDITOR INDEX

ASSEMBLER INDEX

S

INTERACTIVE DEBUGGER INDEX

R

U

OS-9 Technical Information

LIMITED WARRANTY

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B.   RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C.   No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D.   Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV.   RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A.   Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B.   Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C.   CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D.   CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E.   CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F.   CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G.   All copyright notices shall be retained on all copies of the Software.

V.   APPLICABILITY OF WARRANTY

A.   The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B.   The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI.   STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

OS-9 Technical Information

Introduction

OS-9 is a versatile multiprogramming/multitasking operating system for the TRS-80 Color Computer. It is well-suited for a wide range of applications. In addition to multiprogramming, its main features are:

- Comprehensive management of all system resources: memory, input/output, and CPU time

- Efficient operation in typical microcomputer configurations

- An expandable, device-independent unified I/O system

History and Design Philosophy

In the few years since its introduction, OS-9 has developed a worldwide user base of thousands. It is especially popular in the engineering, research, education, and scientific applications.

Microware and Motorola modeled OS-9 upon Bell Telephone Laboratories' UNIX operating system, which is becoming widely recognized as a standard for mini and micro multiprogramming operating systems. Although its implementation is different, OS-9 retains the overall concept and user interface of UNIX. If you are experienced with UNIX, you should feel quite comfortable with OS-9.

In addition, OS-9 introduces some important features to make the most of the 6809 microprocessor. Perhaps the most innovative of these is OS-9's "memory module" management system. This system provides extensive support for modular software techniques, which encourage simplified and more reliable programming.

The OS-9 modules are introduced in *Chapter 1* of this manual and are discussed in detail throughout.

About This Manual

This manual provides all the information necessary to install, maintain, expand, or write assembly-language programs for OS-9 systems. It assumes you are familiar with the 6809 architecture, instruction set, and assembly language.

Special Notations

For your convenience, the following special notations are used in this manual.

lower-case italics
represent words, letters, characters, or values that you supply.

$nn
specifies that *nn* is a hexadecimal number. All other numbers in the text of this manual are in decimal (base 10) form, unless otherwise noted.

Contents

Chapter 4 / OS-9's Unified Input/Output System 25

Chapter 5 / Random Block File Manager . . . 33

Chapter 6 / Sequential Character File Manager 53

Chapter 7 / Assembly Language Programming Techniques. 67

Chapter 8 / System Calls 75

Appendices 149

Table of Diagrams and Charts

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

1 / System Organization

OS-9 is composed of a group of modules, each of which has specific functions. The illustration below shows the major modules. Actual module names are capitalized.

Color Computer OS-9 Modules

```
┌──────┐   ┌─────────────────────┐   ┌───────┐
│ INIT │   │   OS9 and OS9P2      │   │ CLOCK │
│      │   │   OS-9 KERNEL        │   │       │
└──────┘   └─────────────────────┘   └───────┘
                     │
              ┌──────────────┐
              │    IOMAN     │
              └──────────────┘
         ┌───────────┼───────────┐
  ┌──────────┐  ┌──────────┐  ┌──────────┐
  │ RBFMAN   │  │ SCFMAN   │  │ PIPEMAN  │
  │ Disk     │  │ Char.    │  │ Pipe     │
  │ File Man.│  │ File Man.│  │ File Man.│
  └──────────┘  └──────────┘  └──────────┘
```

| CCDISK Disk Driver | CCIO Video/ Keybrd Driver | PRINTER Printer Driver | RS232 Serial Port Driver | PIPES Dummy Driver |

| D0 | D1 | TERM | P | RS232 | PIPE |

Device Descriptors

1
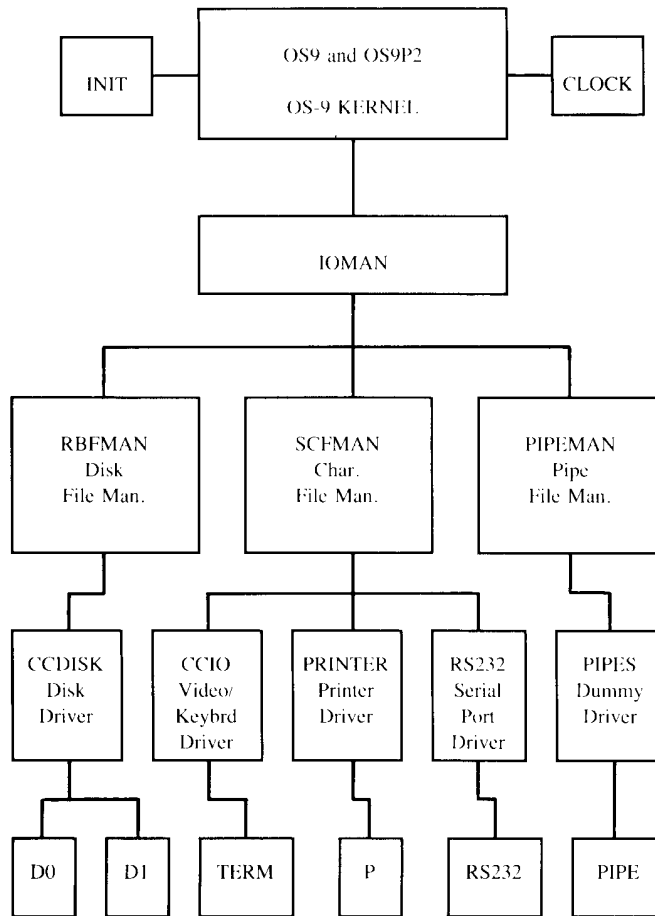
Kernel, Clock Module, and INIT

The first level contains the "kernel," "clock module," and "INIT."

The kernel provides basic system services, such as multitasking and memory management. It links all other system modules.

The clock module is a software handler for the specific real-time-clock hardware.

INIT is an initialization table used by the kernel during system startup. It loads initial tasks and specifies initial table sizes and initial system device names.

Input/Output Modules

The remaining modules make up OS-9's unified I/O system. They are defined briefly here and are discussed in detail in *Chapter 4*.

IOMAN

The second level (the level below the kernel) contains the "input/output" manager (IOMAN). IOMAN provides common processing of all I/O operations. It is required if any OS-supported I/O is to be performed.

File Managers

The third level contains the "file managers." File managers perform I/O request processing for similar classes of I/O devices.

The Random Block File Manager (RBFMAN) processes all disk I/O operations. The Sequential Character File Manager (SCFMAN) handles all non-disk I/O operations that basically operate a character at a time, such as terminal and printer operations. The Pipe File Manager (PIPEMAN) handles pipes,

which are memory buffers that act like files for inter-process data transfers.

Device Drivers

The fourth level contains the "device drivers." Device drivers handle basic physical I/O functions for specific I/O controller hardware. You may add your own customized drivers, or you may receive new drivers with accessory hardware devices.

Device Descriptors

The fifth level contains the "device descriptors." These modules are small tables that associate each I/O port with its logical name, device driver, and file manager. They also contain the port's initialization data and physical address.

When the device descriptors are used, only one copy of each driver is required for each type of I/O controller, regardless of the number of controllers the system uses.

Shell

The "shell" (not shown) is the command interpreter. It is technically a program and not part of the operating system itself, and is described fully in the *OS-9 Commands* manual.

Boot

All modules are loaded into RAM during system startup by the disk bootstrap module, "Boot." Boot (not shown) is initially loaded into memory by the Color BASIC DOS command.

2 / The Kernel

The "nucleus" of OS-9 is the kernel, which supervises the system and manages system resources. It is about 3K bytes long and is contained in the OS9 and OS9P2 modules, with Boot, at memory addresses $F000 - $FEFF.

The kernel's main functions are:

- System initialization after reset
- System call processing
- Memory management
- Multiprogramming (6809 management)
- Interrupt processing

Note: Input/output functions are not included in the list because the kernel does not directly process them. Instead, it passes I/O system calls to IOMAN for processing.

System Initialization

After a hardware reset, the kernel initializes the system. This involves the following: locating modules loaded in memory from the OS-9 Boot file, allocating memory for internal tables, and running the system startup task (SYSGO). The INIT module is used during startup to specify initial table sizes and system device names.

The SYSGO program does the following:

1. Calls the shell and initializes the high-level system.

2. Starts the first user process.

3. Remains in the Wait state to make sure that all uses do not terminate and thus halt the system. SYSGO can keep the system going by restarting the first user process.

System Call Processing

"System calls" are used to communicate between OS-9 and assembly-language programs for such functions as memory allocation and process creation. In addition to I/O and memory management functions, system calls have other functions. These include interprocess control and timekeeping.

System calls use the SWI2 instruction followed by a constant byte representing the code. Parameters for system calls are usually passed in the 6809 register.

OS9Defs and Symbolic Names

A system-wide assembly-language equate file called OS9Defs defines symbolic names for all system calls. This file is included when assembling hand-written or compiler-generated code. The OS-9 assembler has a built-in macro to generate system calls. For example:

    OS9 I$read

is recognized and assembled as the equivalent to:

    SWI2
    FCB I$read

Types of System Calls

System calls are divided into two categories, "I/O" calls and "function" calls.

I/O calls perform various input/output functions. Calls of this type are passed by the kernel to IOMAN for processing. The symbolic names for I/O calls begin with I$. For example, the Read system call is called I$read.

Function calls perform memory management, multi-programming, and other functions. Most are processed by the kernel. The symbolic names for function calls begin with F$. For example, the Link system call is called F$link.

The function calls include "user" calls and privileged "system mode" calls. (See *Chapter 8* for more information.)

Memory Management

Memory management is an important operating system function. Using memory modules, OS-9 manages the logical contents of memory and the physical assignment of memory to programs.

All programs that are loaded must be in the memory module format. This format allows OS-9 to maintain a module directory. The directory contains information about the module, including its name and address and the number of processes using it.

When a module no longer is needed, OS-9 deallocates its part of memory and removes its name from the module directory (except ROM, which is discussed later).

The memory modules are the foundation of OS-9's modular software environment. These are some advantages of memory management:

- Automatic run-time "linking" of programs to libraries of utility modules

- Automatic "sharing" of reentrant programs

- Replacement of small sections of large programs for update or correction (even when in ROM).

Memory Use

OS-9 reserves some space at the top and bottom of RAM for its own use. The amount depends on the sizes of system tables that are specified in the INIT module.

All other RAM is pooled into a "free memory" space. As memory is allocated or deallocated, it is dynamically taken from or returned to this pool.

The basic unit of memory allocation is the 256-byte "page." OS-9 always allocates memory in whole numbers of pages.

The data structure used to keep track of memory allocation is a 32-byte "bit map" located at $0200 - $021F. Each bit in this table is associated with a specific page of memory.

Cleared bits indicate that the page is free and available for assignment. Set bits indicate that the page is in use or that no RAM memory is present at that address.

Memory is allocated automatically when any of the following occurs:

- Program modules are loaded into RAM
- Processes are created
- Process request additional RAM
- OS-9 needs I/O buffers or larger tables

Each of these functions usually has an inverse function that causes memory to be deallocated and returned to free memory.

In general, memory for program modules and buffers is allocated from high addresses downward. Memory for process data areas is allocated from lower addresses upward.

On the next page is a map of a "typical" system. Actual memory sizes and addresses may vary depending on the exact system configuration.

Color Computer OS-9 Typical Memory Map

```
                                    ◄─────  $FFFF
┌─────────────────────────────┐
│      I O Device Addresses    │     ◄─────  $FF00
├─────────────────────────────┤            $FEFF (End of RAM)
│                              │
│        OS-9 Kernel           │     ◄─────  $F000
├─────────────────────────────┤
│       File Managers,         │
│     Device Drivers, and      │
│           so on              │
│         Shell (1K)           │
├─────────────────────────────┤     ◄─────  $C000 (may vary)
│    OS-9 Data Structures      │
│     (approximately 1K)       │
├─────────────────────────────┤
│      Free Memory for         │
│       General Use            │
├─────────────────────────────┤     ◄─────  $0500 (may vary)
│    OS-9 Data Structures      │
│     and Direct Page          │
└─────────────────────────────┘     ◄─────  $0000 Beginning of RAM
```
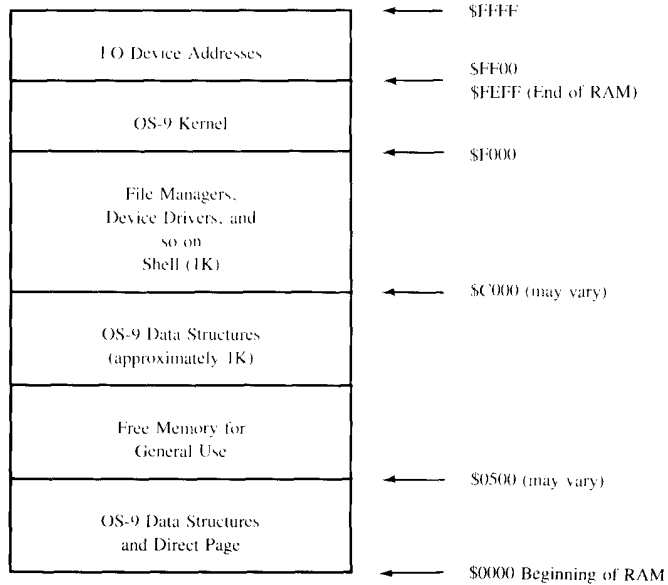
Multiprogramming

OS-9 is a multiprogramming operating system. This means
several independent programs called "processes" can be ex-
ecuted at the same time. By issuing the appropriate system call
to OS-9, each process can have access to any system resource.

9

Multiprogramming functions use a hardware real-time clock that generates interrupts at a regular rate of 60 times per second. Therefore, 6809 time usually is divided into periods of 16.67 milliseconds. These periods are called "ticks."

"Active" processes (those not waiting for some event) are run for a specific system-assigned period called a "time slice." The length of the time slice depends on a process' priority relative to all other active processes. Many OS-9 system calls are available to create, terminate, and control processes.

Process Creation

A process is created when an existing process executes a Fork system call (F$fork). This call's main argument is the name of the program module that the new process is to execute first (the "primary module").

Find the Module. OS-9 first attempts to find the module in the module directory. If it does not find the module, OS-9 usually attempts to load into memory a mass-storage file, giving it the requested module name as a filename.

Assign a Process Descriptor. Once OS-9 finds the module, it assigns the process a data structure called a "process descriptor." This is a 64-byte package that contains information about the process, its state, memory allocations, priority, queue pointers, and so on.

OS-9 automatically initializes and maintains the process descriptor. The process itself cannot access the descriptor; it has no need to do so.

Allocate RAM. The next step is allocation of RAM for the process. The primary module's header contains a storage size. OS-9 uses this size unless the Fork system call asked for a larger area. OS-9 then attempts to allocate a contiguous memory area of the specified size from the free memory space.

Create or Abort. If OS-9 can perform all of the previous steps, it adds the new process to the active process queue for execution scheduling. If it cannot, it aborts the creation. The process that originated the Fork is informed of the error.

Assign Process ID and User ID. OS-9 assigns the new process a unique number called a "process ID." Other processes can communicate with the process by referring to its ID in various system calls.

The process also has a "user ID," which is used to identify all processes and files belonging to a particular user. The user ID is inherited from the parent process.

Process Termination

A process terminates when it executes an Exit system call (F$exit) or when it receives a fatal signal. The termination closes any open paths, deallocates memory used by the process, and unlinks the primary module.

Process States

At any instant a process can be in one of three states:

- Active (The process is ready for execution.)

- Waiting (The process is suspended until a "child process" terminates or a signal is received.)

- Sleeping (The process is suspended for a specific period of time or until a signal is received.)

Each state has its own queue, a linked list of "descriptors" of processes in that state. State changes are performed by moving a process descriptor to another queue.

The Active State. Each active process is given a time slice for execution, according to its priority relative to all other active processes. The scheduler, which is in the kernel, makes sure that all active processes, even those of low priority, get some CPU time.

The Wait State. This state is entered when a process executes a Wait system call (F$wait). The process remains suspended until one of its "child" processes dies or until it receives a "signal." (See the "Signals" section below.)

The Sleeping State. This state is entered when a process executes a Sleep system call (F$Sleep), which specifies the number of ticks for which the process is to remain suspended. The process remains asleep until the specified time has elapsed or until it receives a signal.

Execution Scheduling

The OS-9 scheduler uses an algorithm that ensures that all active processes get some execution time.

All active processes are members of the "active process queue," which is kept sorted by process "age." Age is a count of how many process switches have occurred since the process' last time slice. When a process is moved to the active process queue from another queue, its age is set according to its priority — the higher the priority, the higher the age.

Whenever a new process becomes active, the ages of all other active processes are incremented. When the executing process' time slice has elapsed, the scheduler selects the next process to be executed (the one with the next highest age, the first one in the queue). At this time the ages of all other active processes are incremented. (Ages are never incremented beyond 255, however.)

An exception is a newly active process that was deactivated while in the system state. Such a process is given higher priority because it usually is executing critical routines that affect shared system resources. Therefore, it could be blocking other unrelated processes.

When there are no active processes, the kernel sets itself up to handle the next interrupt and then execute a Cwai instruction, which decreases interrupt latency time.

Signals

A signal is an asynchronous control mechanism used for interprocess communication and control. It behaves like a software interrupt (it can cause a process to suspend a program, execute a specific routine, and afterward return to the interrupted program).

Signals can be sent from one process to another process by the Send system call (F$send). Or, they can be sent from OS-9 service routines to a process.

The signal can convey status information in the form of a 1-byte numeric value. Some of the signal "codes" (values) are predefined, but you may define most. The signal codes are:

```
0  =  Kill (aborts the process; is non-interceptable)
1  =  Wakeup (wakes up a sleeping process)
2  =  Keyboard abort
3  =  Keyboard interrupt
4  =  User-defined (255)
```

When a signal is sent to a process, the signal is noted and saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. When it gets its next time slice, the signal is processed.

What happens next depends on whether or not the process has set up a "signal intercept trap" (signal service routine) by executing an Intercept system call (F$icpt).

If it has set up a signal intercept trap, the process resumes execution at the address given in the Intercept call. The signal code is passed to this routine, which should terminate with an RTI instruction to resume normal execution of the process.

> **Note:** A wakeup signal activates a sleeping process. It **does not** vector through the intercept routine.

If it has not set up a signal intercept trap, the process is aborted immediately. It is also aborted if the signal code is zero. If the process is in the system mode, the abort is deferred (it dies upon return to the user state).

A process may have a signal pending (usually because it has not been assigned a time slice since the signal was received). If it does, and another process tries to send it another signal, the new signal is aborted, and the Send system call returns an error. The sender should then execute a Sleep system call for a few ticks before trying to send the signal again. This gives the destination process time to process the pending signal.

Interrupt Processing

Interrupt processing is another important function of the kernel. OS-9 sends each hardware interrupt to a specific address. This address, in turn, specifies the address of the device service routine to be executed. This is called "vectoring" the interrupt. The address that points to the routine is called the "vector." It has the same name as the interrupt.

Physical Interrupt Processing

Addresses $FFF0 through $FFFF (in the Color BASIC ROM) contain the hardware vectors required by 6809. Each address points to a RAM vector, as follows:

Interrupt	Vector
SWI3 (Software Interrupt 3)	$0100
SWI2 (Software Interrupt 2)	$0103
FIRQ (Fast Interrupt Request)	$010F
IRQ (Interrupt Request)	$010C
SWI (Software Interrupt)	$0106
NMI (Non-Maskable Interrupt)	$0109

OS-9 initializes each of these RAM vectors to point to a specific service routine in the kernel.

Software Interrupts (SWI, SWI2, and SWI3.) The software interrupts are vectored to user-definable addresses, which are local to each procedure. SWI2, however, usually is used for OS-9 system calls.

The SWI, SWI2, and SWI3 vectors point to routines that read the corresponding pseudo vector from the process' descriptor and dispatch to it. This is why the Set SWI system call (F$sswi) is local to a process; it only changes a pseudo vector in the process descriptor.

FIRQ Interrupt. The system uses the FIRQ interrupt. The FIRQ vector, which points to an RTI instruction, is not available to you.

IRQ Interrupt. Most OS-9 I/O devices generate IRQ interrupts. The IRQ vector points to the real-time clock and keyboard scanner routines. These routines, in turn, jump to a special IRQ polling system which determines the source of the interrupt. The polling system is discussed in the next section, "Logical Interrupt Polling System."

NMI Interrupt. The system uses the NMI interrupt. The NMI vector, which points to the disk driver interrupt service routine, is not available to you.

Logical Interrupt Polling System

Because most OS-9 I/O devices use IRQ interrupts, OS-9 includes a sophisticated polling system. The IRQ polling system automatically identifies the source of the interrupt, and then executes its associated user- or system-defined service routine.

The Polling Table. The information required for IRQ polling is maintained in a data structure called the "IRQ polling table." The table has a 9-byte entry for each device that might generate an IRQ interrupt. The table size is permanent and is defined by an initialization constant in the INIT module.

Each entry in the polling table is given a number from 0 (lowest priority) to 255 (highest priority). In this way, the more important devices (those that have a higher interrupt frequency) can be polled before the less important ones.

Each entry has 6 variables:

- **Polling Address** — points to the device's status register. The register must have a bit or bits that indicate it is the source of an interrupt.
- **Flip Byte** — selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).

- **Mask Byte** — selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.

15

- **Service Routine Address** — points to the device's interrupt service routine. You supply this address.

- **Static Storage Address** — points to the permanent storage area required by the device service routine. You supply this address.

- **Priority** — sets the order in which the devices are to be polled (a number from 0 to 255).

Polling the Entries. When an IRQ interrupt occurs, the polling system is entered via the corresponding RAM interrupt vector. It starts polling the devices in order. Each entry's status register address is loaded into Accumulator A, using the device address from the table.

Then, an Exclusive-OR operation using the flip byte is executed, followed by a Logical-AND operation using the mask byte. If the result is non-zero, the device is assumed to be the cause of the interrupt.

The device's memory address and service routine address are read from the table and executed.

> **Note:** The interrupt service routine should terminate with an RTS instruction, **not** an RTI instruction.

Adding Entries to the Table. Using the Set IRQ system call (F$irq), you can make entries to the IRQ polling table. Set IRQ is a privileged system call, which can be executed only when OS-9 is in system mode (which is the case when device drivers are executed).

> **Note:** The actual code for the interrupt polling system is located in the IOMAN module. The kernel OS9 and OS9P2 modules contain the physical interrupt processing routines.

3 / Memory Modules

In *Chapter 2*, you learned that OS-9 is based on the concept that memory is modular. This means that each program is considered to be an individual, named "object."

You also learned that all objects that are loaded into memory must be in the module format. This format lets OS-9 manage the logical contents of memory, as well as the physical contents. Module types and formats are discussed in detail in this chapter.

Module Types

There are several types of modules, each of which has a different use and function. These are the main requirements of a module:

- It cannot modify itself.

- It must be position-independent so OS-9 can load or relocate it wherever space is available. In this respect, the module format is the OS-9 equivalent of "load records" used in older operating systems.

A module need not be a complete program or even 6809 machine language. It may contain BASIC09 "I-code," constants, single subroutines, and subroutine packages.

Module Format

Each module has three parts; a "module header," a "module body," and a "cyclic-redundancy-check value" (CRC value).

Module Format

Module Header
Program or Constants
CRC Value
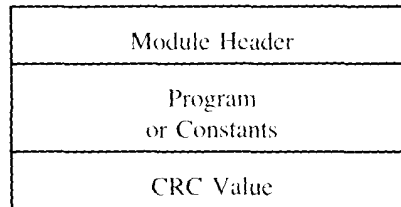
Module Header

At the beginning of the module (lowest address) is the module header. Its form depends upon the module's use.

The header contains information about the module and its use. This includes the following:

- Size

- Type (machine language, BASIC09 compiled code, and so on)

- Attributes (executable, reentrant, and so on)

- Data storage memory requirements

- Execution starting address

Usually, you don't need to write routines to generate the modules and headers. Most OS-9 programs — including BASIC09, Pascal, C, and the assembler — do this automatically.

Module Body

The module body contains the program or constants. It usually is pure code. The module name string is included somewhere in this area.

CRC Value

The last three bytes of the module are the Cyclic Redundancy Check (CRC) value. The CRC value is used to verify the integrity of a module.

The 24-bit CRC is performed over the entire module from the first byte of the module header to the byte just before the CRC itself. The CRC polynomial used is $800FE3.

As with the header, you usually don't need to write routines to generate the CRC value. Most OS-9 programs do this automatically.

Module Headers: Standard Information

The first nine bytes of all module headers are defined as follows:

Relative Address	Use
$00,$01	Sync bytes ($87,$CD)
$02,$03	Module size
$04,$05	Offset to module name
$06	Module type / Language type
$07	Attributes / Revision level
$08	Header check

Sync Bytes. Specify the location of the module. (The first sync byte is the start of the module.) These two bytes are constant.

Module Size. Specify the size of the module in bytes (includes CRC).

Offset to Module Name. Specifies the address of the module name string relative to the start of the module. The name string can be located anywhere in the module and consists of a string of ASCII characters having the most significant bit set on the last character.

Type/Language Byte. Specifies the type and language of the module.

The four most significant bits of this byte indicate the type. Eight types are pre-defined. Some of these are for OS-9's internal use only. The type codes are given here:

Code	Module Type
$01	Program module
$02	Subroutine module
$03	Multi-module (for future use)
$04	Data module
$05-$0B	User-definable
$0C	OS-9 system module
$0D	OS-9 file manager module
$0E	OS-9 device driver module
$0F	OS-9 device descriptor module

The four least significant bits of this byte indicate the language. The language codes are given here:

Code	Language
$00	Data (non-executable)
$01	6809 object code
$02	BASIC09 I-code
$03	PASCAL P-code
$04	COBOL I-code
$05-$15	Reserved for future use

By checking the language type, high-level language run-time systems can verify that a module is the correct type before attempting execution. BASIC09, for example, may run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

Attributes / Revision Byte. Specifies the attributes and revision level of the module.

The four most significant bits of this byte are reserved for module attributes. Currently, only Bit 7 is defined. When set, it indicates the module is reentrant and therefore "sharable."

The four least significant bits of this byte are a revision level from 0 to 15. If two or more modules have the same name, type, language, and so on, OS-9 keeps in the module directory only the module having the highest revision level. Therefore, you can replace or patch a ROM module, simply by loading a new, equivalent module that has a higher revision level.

> **Note:** A previously linked module cannot be replaced until all processes linked to it have unlinked it (after its link count goes to zero).

Header Check. Specifies the one's complement of the Exclusive-OR of the previous eight bytes.

Module Headers: Type-Dependent Information

More information usually follows the first nine bytes of the header. The layout and meaning vary, depending on the module type.

Module types $0C-$0F (system module, file manager module, device driver module, and device descriptor module) are used by OS-9 only. Their formats are given later in the manual.

Module types $01 through $0B have the general-purpose "user" format (executable format) shown on the next page. This format is used often for OS-9 programs that are called by Fork or Chain (F$fork and F$chain).

Executable Memory Module Format

Relative Address	Use	Check Range
$00 $01	Sync Bytes ($87CD)	
$02 $03	Module Size (bytes)	
$04 $05	Module Name Offset	header parity
$06	Type / Language	
$07	Attributes / Revision	module CRC
$08	Header Parity Check	
$09 $0A	Execution Offset	
$0B $0C	Permanent Storage Size	
$0D	(Additional optional header extensions located here) · · · · · · · Module Body object code, constants, and so on	
	CRC Check Value	

22
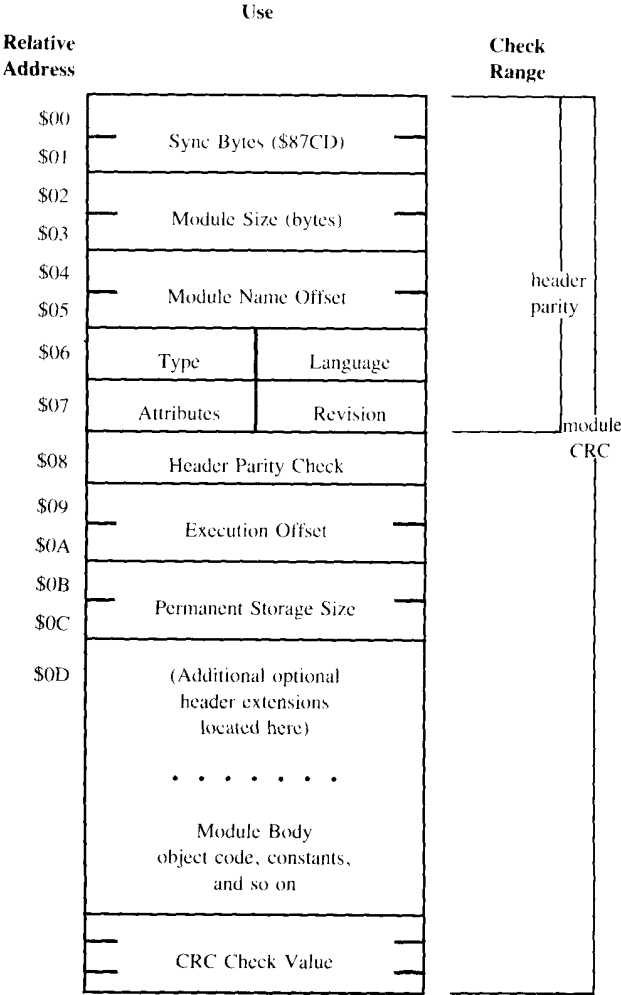
As you can see from the chart on the previous page, this module format has two extra bytes in its header. They are:

$09,$0A	Execution offset
$0B,$0C	Permanent storage size

Execution Offset. The program or subroutine's offset starting address, relative to the first byte of the sync code. A module that has multiple entry points (such as cold start and warm start) may have a branch table starting at this address.

Permanent Storage Size. The minimum number of bytes of data storage required to run. Fork and Chain use this number to allocate a process' data area.

If the module will not be directly executed by a Fork or Chain system call (for instance a subroutine package), this entry is not used by OS-9. It is commonly used to specify the maximum stack size required by reentrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

ROM Modules

When OS-9 starts after a system reset, it searches the entire memory space for ROM modules. It finds them by looking for the module header sync code ($87,$CD).

When this byte pattern is detected, OS-9 checks the header to see if it is correct. If it is, the system obtains the module size from the header and performs a 24-bit CRC over the entire module. If the CRC matches, OS-9 considers the module to be valid and enters it into the module directory. All ROM modules that are present in the system at startup are automatically included in the system module directory.

After the module search, OS-9 links to the component modules it found. This is the secret to OS-9's ability to adapt to almost any 6809 computer. It automatically locates its required and optional component modules and rebuilds the system each time it is started.

At startup, OS-9 also searches ROM for non-system modules. It locates any software you supply and, if the module's header and CRC information is correct, enters it into the module directory.

4 / OS-9's Unified Input/Output System

In Chapter 1, we mentioned that OS-9 has a unified I/O system, consisting of all modules except those on the kernel level. This chapter discusses the I/O modules in detail.

I/O System Modules

```
                        ┌──────────────────┐
                        │      IOMAN       │
                        └──────────────────┘
              ┌───────────────────┼───────────────────┐
    ┌──────────────┐      ┌──────────────┐      ┌──────────────┐
    │   RBFMAN     │      │   SCFMAN     │      │   PIPEMAN    │
    │    Disk      │      │    Char.     │      │    Pipe      │
    │   File Mgr.  │      │   File Mgr.  │      │   File Mgr.  │
    └──────────────┘      └──────────────┘      └──────────────┘
        ┌──────┴──────┐   ┌──────┴──────┬──────────┬────────────┐
  ┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐
  │ CCDISK  │ │  CCIO   │ │ PRINTER │ │  RS232  │ │  PIPES  │
  │  Disk   │ │ Video·  │ │ Printer │ │ Serial  │ │ Dummy   │
  │ Driver  │ │ Keybd.  │ │ Driver  │ │  Port   │ │ Driver  │
  │         │ │ Driver  │ │         │ │ Driver  │ │         │
  └─────────┘ └─────────┘ └─────────┘ └─────────┘ └─────────┘
   ┌────┬────┐    │           │           │           │
  ┌────┐┌────┐ ┌──────┐   ┌──────┐   ┌────────┐   ┌──────┐
  │ D0 ││ D1 │ │ TERM │   │  P   │   │ RS232  │   │ PIPE │
  └────┘└────┘ └──────┘   └──────┘   └────────┘   └──────┘
```
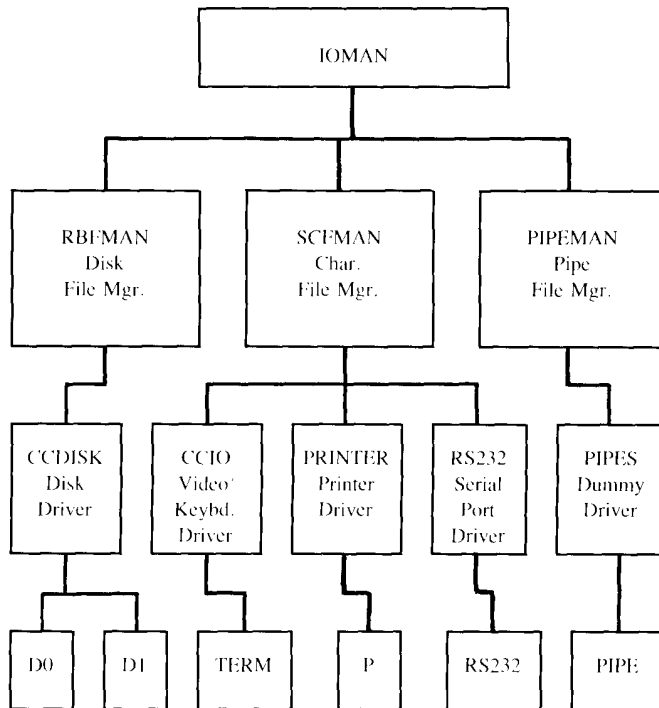
Device Descriptors

The I/O system provides system-wide, hardware-independent I/O services for user programs and OS-9 itself. All I/O system calls are received by the kernel and passed to IOMAN for processing.

25

IOMAN performs some processing, such as the allocation of data structures for the I/O path. Then, it calls the file managers and device drivers to do most of the work. Additional file manager, device driver, and device descriptor modules can be loaded into memory from files and used while the system is running.

IOMAN

IOMAN provides the first level of service of I/O system calls. It routes data on I/O process paths to or from the appropriate file managers and device drivers.

IOMAN also maintains two important internal OS-9 data structures — the "device table" and the "path table." Never modify IOMAN.

When a path is opened, IOMAN tries to link to a memory module that has the device name given or implied in the pathlist. This module is the device descriptor. It contains the names of the device driver and file manager for the device. IOMAN saves the names so later system calls can be routed to these modules.

File Managers

OS-9 can have any number of file manager modules. Each of these modules processes the raw data stream to or from a class of device drivers that have similar operational characteristics. It removes as many unique characteristics as possible from I/O operations. Thus, it assures that similar devices conform to the OS-9 standard I/O and file structure.

The file manager also is responsible for mass storage allocation and directory processing, if these are applicable to the class of devices it served.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They may also monitor and process the data stream, for example, adding line-feed characters after carriage-return characters.

The file managers are reentrant. The two standard OS-9 file managers are:

- Random Block File Manager (RBFMAN) — supports random-access, block-structured devices such as disk systems and bubble memories. (*Chapter 5* discusses RBFMAN in detail.)

- Sequential Character File Manager (SCFMAN) — supports single-character-oriented devices, such as CRT or hardcopy terminals, printers, and modems. (*Chapter 6* discusses SCFMAN in detail.)

Device Driver Modules

The device driver modules are subroutine packages that perform basic, low-level I/O transfers to or from a specific type of I/O device hardware controller. These modules are reentrant so one copy of the module can run at the same time several devices that use identical I/O controllers.

Device driver modules use a standard module header, in which the module type is specified as code $0E (device driver). The execution offset address in the module header points to a branch table that has a minimum of six 3-byte entries.

Each entry is typically a LBRA to the corresponding subroutine. The file managers call specific routines in the device driver through this table, passing a pointer to a path descriptor and the hardware control register address in the 6809 register. The branch table looks like this:

Code	Meaning
+ $00	Device initialization routine
+ $03	Read from device
+ $06	Write to device
+ $09	Get device status
+ $0C	Set device status
+ $0F	Device termination routine

(For a complete description of the parameters passed to these subroutines, see the "Device Driver Subroutines" sections in *Chapters 5 and 6.*)

Device Descriptor Modules

Device descriptor modules are small, non-executable modules. They provide information that associates a specific I/O device with its logical name, hardware controller address(es), device driver, file manager name, and initialization parameters.

Unlike the device drivers and file managers — which operate on classes of devices — each device descriptor tailors its functions to a specific device. Each device must have a device descriptor.

Device descriptor modules use a standard module header, in which the module type is specified as code $0F (device descriptor). The name of the module is the name by which the system and user know the device (the device name given in pathlists).

The rest of the device descriptor header consists of the information in the following chart:

Relative Address(es)	Use
$09,$0A	File manager name string relative address
$0B,$0C	Device driver name string relative address
$0D	Mode/Capabilities: D S PE PW PR E W R (directory, sharable, public execute, public write, public read, execute, write, read)
0E$,$0F,$10	Device controller absolute physical (24-bit) address
$11	Number of bytes (n bytes) in the intialization table)
$12,$12 + n	Initialization table

When a path to the device is opened, the initialization table is copied into the ''option section'' (PD.OPT) of the path descriptor. (See ''Path Descriptors'' in this chapter.)

The values in this table may be used to define the operating parameters that are changeable by the Get Status and Set Status system calls (I$getstt and I$setstt). For example, a terminal's initialization parameters define which control characters are used for such functions as backspace and delete.

The initialization table may be up to 32 bytes long. If the table is less than 32 bytes long, the remaining values in the path descriptor are set to 0.

You may wish to add devices to your system. If a similar device controller already exists, all you need to do is add the new hardware and load another device descriptor. Device descriptors can be in ROM or loaded into RAM from mass-storage files while the system is running.

The diagram on the next page illustrates the device descriptor format:

Device Descriptor Format

| | Use | |
Relative Address		Check Range

$00		
$01	Sync Bytes ($87CD)	
$02		
$03	Module Size	
$04		header
$05	Offset to Module Name	parity
$06	$F (Type) $1 (Lang)	
$07	Attributes Revision	
$08	Header Parity Check	
$09		
$0A	Offset to File Manager Name String	module
$0B		CRC
	Offset to Device Driver Name String	
$0D	Mode Byte	
$0E		
$0F	Device Controller Absolute Physical Addr.	
$10	(24 bit)	
$11	Initialization Table Size	
$12, 12 + n	(Initialization Table)	
	(Name Strings, and so on)	
	CRC Check Value	

30

Path Descriptors

Every open path is represented by a data structure called a path descriptor (PD). It contains the information the file managers and device drivers require to perform I/O functions.

PDs are 64 bytes long and are dynamically allocated and deallocated by IOMAN as paths are opened and closed.

They are internal data structures, which are not normally referenced from user or applications programs. The description of PDs is presented here mainly for those programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections. The first 10-byte section is the same for all file managers and device drivers, as shown in the chart on the next page:

Path Descriptor: Standard Information

Name	Relative Address	Size (Bytes)	Use
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

PD.FST	$0A	22	Defined by the file manager
PD.OPT	$20	32	Reserved for the Getstat/Setstat options

PD.FST is reserved for and defined by each type of file manager for file pointers, permanent variables, and so on.

PD.OPT is used as an option area for file or device operating parameters that are dynamically alterable. When the path is opened, IOMAN initializes these variables by copying the initialization table that is in the device descriptor module. User programs can change them later, using the Get Status and Set Status system calls.

PD.FST and PD.OPT are defined for the file manager in the assembly-language equate file (SCFDefs for SCFMAN and RBFDefs for RBFMAN).

5/ Random Block File Manager

The Random Block File Manager (RBFMAN) supports disk storage. It is a reentrant subroutine package called by IOMAN for I/O system calls to random-access devices. It maintains the logical and physical file structures.

During normal operation, RBFMAN requests allocation and deallocation of 256-byte data buffers. Usually, one is required for each open file. When physical I/O functions are necessary, RBFMAN directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks (pages).

RBFMAN does not deal directly with physical addresses such as tracks and cylinders. Instead, it passes to the device drivers address parameters, using a standard address called a "logical sector number", or "LSN." LSNs are integers from 0 to $n-1$, where n is the maximum number of sectors on the media. The driver translates the logical sector number to actual cylinder/track/sector values.

Because RBFMAN supports many devices that have different performance and storage capacities, it is highly parameter-driven. The physical parameters it uses are stored on the media itself.

On disk systems, this information is written on the first few sectors of Track 0. The device drivers also use this information, particularly the physical parameters stored on Sector 0. These parameters are written by the Format program that initializes and tests the media.

Logical and Physical Disk Organization

All disks used by OS-9 use the first few sectors to store basic identification, file structure, and storage allocation information.

LSN 0 is the "identification sector." LSN 1 is the "disk allocation map sector." LSN 2 marks the beginning of the disk's root directory. The following section tells more about LSN 0 and LSN 1.

Disk Sectors

Identification Sector (LSN 0)

LSN 0 contains a description of the physical and logical characteristics of the disk. These characteristics are set by the Format command program when the disk is initialized.

The table below gives the OS-9 mnemonic name, byte address, size, and description of each value stored in this LSN 0.

Name	Relative Address	Size (Bytes)	Use
DD.TOT	$00	3	Number of sectors on disk
DD.TKS	$03	1	Track size (in sectors)
DD.MAP	$04	2	Number of bytes in the allocation bit map
DD.BIT	$06	2	Number of sectors per cluster
DD.DIR	$08	3	Starting sector of the root directory
DD.OWN	$0B	2	Owner's user number
DD.ATT	$0D	1	Disk attributes
DD.DSK	$0E	2	Disk identification (for internal use)
DD.FMT	$10	1	Disk format, density, number of sides
DD.SPT	$11	2	Number of sectors per track
DD.RES	$13	2	Reserved for future use
DD.BT	$15	3	Starting sector of the bootstrap file
DD.BSZ	$18	2	Size of the bootstrap file (in bytes)
DD.DAT	$1A	5	Time of creation: Y:M:D:H:M
DD.NAM	$1F	32	Volume name: last character has the most significant bit set

Disk Allocation Map Sector (LSN 1)

LSN 1 contains the "disk allocation map," which is created by Format. This map specifies which sector clusters are available for file allocation.

The DD.MAP value in LSN 0 specifies the number of bytes (in LSN 1) that are used in the map.

Each bit on the map corresponds to one sector cluster on the disk. The DD.BIT value in LSN 0 specifies the number of sectors per cluster. The number is an integral power of 2 (1, 2, 4, 8, 16, and so on). The map may contain up to 4096 bits. Therefore, hard disks and double-density, doubled-sided disk-ettes have two or more sectors per cluster.

If a cluster is available, the corresponding bit is cleared. If it is allocated, non-existent, or physically defective, the corresponding bit is set.

File Sectors

File Descriptor Sector

The first sector of every file is the "file descriptor." It contains the logical and physical description of the file.

The table below describes the contents of the file descriptor.

Name	Relative Address	Size (Bytes)	Use
FD.ATT	$00	1	File attributes: D S PE PW PR E W R (see below)
FD.OWN	$01	2	Owner's user ID
FD.DAT	$03	5	Date last modified: Y M D H M
FS.LNK	$08	1	Link count
FD.SIZ	$09	4	File size (number of bytes)
FD.DCR	$0D	3	Date created: Y M D
FD.SEG	$10	240	Segment list (see below)

FD.ATT (the attribute byte) contains the file permission bits. When set the bits indicate the following:

Bit 7	Directory
Bit 6	Sharable
Bit 5	Public execute
Bit 4	Public write
Bit 3	Public read
Bit 2	Execute
Bit 1	Write
Bit 0	Read

FD.SEG (the segment list) consists of up to 48 5-byte entries that have the size and address of each file block in logical order. Each entry has the block's 3-byte LSN and 2-byte size (in sectors). The entry following the last segment will be zero.

After creation, a file has no data segments allocated to it until the first write. (Write operations past the current end-of-file cause sectors to be added to the file. The first write is always past the end-of-file.)

If the file has no segments, it is given an initial segment. Usually, this segment has the number of sectors specified by the minimum allocation entry in the device descriptor. If, however, the number of sectors requested is more than the minimum, the initial segment has the requested number.

Later expansions of the file usually are made in minimum allocation increments, also. OS-9 expands the last segment, whenever possible, instead of adding a segment. When the file is closed, unused sectors in the last segment are truncated.

> **Note:** OS-9 tries to minimize the number of storage segments used in a file. In fact, many files have only one segment. In such cases, no extra read operations are needed to randomly access any byte on the file.
>
> If a file is repeatedly closed, opened, and expanded, it may become fragmented and may have many segments. You can avoid this problem by writing a byte at the highest address to be used on a file. Do this before writing any other data.

Directories

"Disk directories" are files that have the D attribute set. A directory contains an integral number of entries, each of which can hold the name and LSN of a file or another directory.

Each directory entry contains 29 bytes for the filename, followed by the 3-byte LSN of the file's descriptor sector. The filename is left-justified in the field with the sign bit of the last character set. Unused entries have a zero byte in the first filename character position.

Every disk has a master directory called the "root directory." The DD.DIR value in LSN 0 (identification sector) specifies the starting sector of the root directory.

RBFMAN Definitions of the Path Descriptor

As stated earlier in this chapter, the PD.FST section of the path descriptor is reserved for and defined by the file manager. The table below describes the use of this section by RBFMAN. For your convenience, it also includes the other sections of the PD.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers and device drivers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
RBFMAN Path Decriptor Definitions (PD.FST Section)			
PD.SMF	$0A	1	State flag: 0 = current buffer is altered, 1 = current sector is in the buffer, 2 = descriptor sector is in the buffer
PD.CP	$0B	4	Current logical file position (byte address)
PD.SIZ	$0F	4	File size
PD.SBL	$13	3	Segment beginning logical sector number (LSN)
PD.SBP	$16	3	Segment beginning physical sector number (PSN)
PD.SSZ	$19	2	Segment size
PD.DSK	$1B	2	Disk ID (for internal use only)
PD.DTB	$1D	2	Address of drive table
RBFMAN Option Section Definitions (PD.OPT Section)			
(Copied from the device descriptor)			
	$20	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
PD.DRV	$21	1	Drive number (0 . . n)
PD.STP	$22	1	Step rate
PD.TYP	$23	1	Device type
PD.DNS	$24	1	Density capability
PD.CYL	$25	2	Number of cylinders (tracks)
PD.SID	$27	1	Number of sides (surfaces)
PD.VFY	$28	1	0 = verify disk writes
PD.SCT	$29	2	Default number of sectors per track
PD.T0S	$2B	2	Default number of sectors per track (Track 0)
PD.ILV	$2D	1	Sector interleave factor
PD.SAS	$2E	1	Segment allocation size
(Not copied from the device descriptor)			
PD.ATT	$33	1	File attributes (D S PE PW PR E W R)
PD.FD	$34	3	File descriptor PSN
PD.DFD	$37	3	Directory file descriptor PSN
PD.DCP	$3A	4	File's directory entry pointer
PS.DVT	$3E	2	Address of the device table entry

Any values not determined by this table default to zero.

RBF-Type Device Descriptor Modules

This section describes the use of the initialization table contained in the device descriptor modules for RBF-type devices. The values below are those IOMAN copies from the device descriptor to the path descriptor.

Name	Relative Address	Size (Bytes)	Use
	$0-$11		Standard device descriptor module header
IT.DTP	$12	1	Device type: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
IT.DRV	$13	1	Drive number (see below)
IT.STP	$14	1	Step rate (see below)
IT.TYP	$15	1	Device type (see below)
IT.DNS	$16	1	Media density: 0 = single, 1 = double (see below)
IT.CYL	$17	2	Number of cylinders (tracks)
IT.SID	$19	1	Number of sides
IT.VFY	$1A	1	0 = Verify disk writes
IT.SCT	$1B	2	Default number of sectors per track
IT.TOS	$1D	2	Default number of sectors per track (Track 0)
IT.ILV	$1F	1	Sector interleave factor
IT.SAS	$20	1	Minimum size of segment allocation (number of sectors to be allocated at one time)

IT.DRV is used to associate a 1-byte integer with each drive that a controller handles. The drives for each controller should be numbered 0 to *n-1*, where *n* is the maximum number of drives the controller can handle.

IT.STP (floppy disks) sets the head stepping rate to be used with a drive. To reduce access time, this should be the fastest rate the drive can support. The actual values stored depend on the specific disk controller and disk driver module used. Below are the values that are used by the popular Western Digital Floppy Disk Controller IC (FD179X 5-Inch):

Step Code	Rate
0	30ms
1	20ms
2	12ms
3	6 ms

IT.TYP specifies the device type (all types).

Bit 0 —   0 = 5-inch floppy disk
           1 = 8-inch floppy disk

Bit 5 —   0 = Non-Color Computer format
           1 = Color Computer format

Bit 6 —   0 = Standard OS-9 format
           1 = Non-standard format

Bit 7 —   0 = Floppy disk
           1 = Hard disk

IT.DNS specifies the density capabilities (floppy disk only).

Bit 0 —   0 = Single-bit density (FM)
           1 = Double-bit density (MFM)

Bit 1 —   0 = Single-track density (5-inch, 48 tracks per inch)
           1 = Double-track density (5-inch, 96 tracks per inch)

RBF-Type Device Driver Modules

An RBF-type device driver module contains a package of sub-routines that perform sector-oriented I/O to or from a specific hardware controller. These modules are usually reentrant. Because of this, one copy of the module can simultaneously run several devices that use identical I/O controllers.

IOMAN allocates a permanent memory area for each device (which may control several drives). The size of the memory area is given in the device driver module header. IOMAN and RBFMAN use some of this area. The device driver may use the rest in any manner. This area is used as follows:

RBF Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use	
V.PAGE	$00	1	Port extended address (A20 -A16)	24-bit device address
V.PORT	$01	2	Device base address	(defined by IOMAN)
V.LPRC	$03	1	ID of the last active process (Not used by RBF device drivers)	
V.BUSY	$04	1	ID of the active process; 0 = not busy (defined by RBFMAN)	
V.WAKE	$05	1	ID of the process to reawaken after device completes I/O; 0 = no process waiting (defined by device driver)	
V.USER		.	End of the OS-9 definitions	
V.NDRV	$06	1	Number of drives the controller can use (defined by the device driver)	
DRVBEG		.	Beginning of the drive tables	
TABLES	$07	DRVMEN*N	Number of tables reserved (n)	
FREE			Free for the driver to use	

V.PAGE through V.USER are pre-defined in the OS-9Defs file. V.NDRV, DRVBEG, DRVMEN are pre-defined in the RBFDefs file.

TABLES. This area contains one table for each drive that the controller will handle. (RBFMAN assumes that there are as many tables as indicated by V.NDRV). Some time after the driver Init routine has been called, RBFMAN issues a request

for the driver to read LSN 0 (the identification sector) from a drive table by copying the first part of LSN 0 (up to DD.SIZ) into it.

The format of each drive table is as given below:

Name	Relative Address	Size (Bytes)	Use
DD.TOT	$00	3	Number of sectors.
DD.TKS	$03	1	Track size (in sectors).
DD.MAP	$04	2	Number of bytes in the allocation bit map.
DD.BIT	$06	2	Number of sectors per bit (cluster size).
DD.DIR	$08	3	Address (LSN) of the root directory.
DD.OWN	$0B	2	Owner's user number.
DD.ATT	$0D	1	Disk access attributes: D S PE PW PR E W R.
DD.DSK	$0E	2	Disk ID (a pseudo-random number used to detect disk swaps).
DD.FMT	$10	1	Media format.
DD.SPT	$11	2	Number of sectors per track. (Track 0 may use a different value, specified by IT.TOS in the device descriptor.)
DD.RES	$15	2	Reserved for future use.
DD.SIZ		.	
V.TRAK	$15	2	Number of the current track (the track the head is on — the one updated by the driver).
V.BMB	$17	1	Bit-map use flag; 0 = the bit map is not in use. (Disk driver routines must not alter V.BMB.)
DRVMEN	$18	.	Size of each drive table.

The format attributes (DD.FMT) are these:

Bit B0 = Number of sides
    0 = Single-sided
    1 = Double-sided

Bit B1 = Density
    0 = Single-density
    1 = Double-density

Bit B2 = Track density
    0 = Single (48 tracks per inch)
    91 = Double (96 tracks per inch)

RBFMAN Device Driver Subroutines

Like all device driver modules, RBFMAN device drivers use a standard executable memory module format. In the header, the type is specified as code $0E (device driver).

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a long branch (LBRA) to the corresponding subroutine. The branch table is defined as follows:

ENTRY	LBRA	INIT	Initialize drive
	LBRA	READ	Read sector
	LBRA	WRITE	Write sector
	LBRA	GETSTA	Get status
	LBRA	SETSTA	Set status
	LBRA	TERM	Terminate device

Each subroutine should exist with the C bit of the Condition Code Register cleared if no error occurred. If an error occurred, the C bit should be set and an appropriate error code returned in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init

Function:

Initialize the device and its memory area.

Entry Conditions:

     U   = *address of the device memory area*
     Y   = *address of the device descriptor*

Exit Conditions:

     None

     If error:
     CC   = C bit set
     B   = *error code*

Technical Function:

1. If the disk writes are verified, use the Request Memory system call (F$srqmem) to allocate a 256-byte buffer area where a sector may be read back and verified after a write.

2. Initialize the device memory area. For floppy disk controllers this typically consists of:

    - Initializing V.NDRV to the number of drives with which the controller will work

    - Initializing DD.TOT (in the drive table) to a non-zero value so that Sector 0 may be read or written.

    - Initializing V.TRAK to $FF so that the first seek finds Track 0.

3. Place the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F$irq).

4. Initialize the device control registers (enable interrupts if necessary).

Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These contain the 24-bit device address.) The driver should initialize each drive table appropriately for the type of disk the driver expects to use on the corresponding drive.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it isn't clear, the driver goes back to sleep.

Function:

Terminates a device. This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).

Read

Function:

Read a 256-byte sector from the disk and place it in the 256-byte buffer.

Entry Conditions:

U   = *address of the device memory area*
Y   = *address of the path descriptor*
B   = *MSB of the disk LSN*
X   = *LSBs of the disk LSN*

Exit Conditions:

The sector is returned in the sector buffer.

If error:
CC   = *C bit set*
B   = *error code*

Technical Function:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.DRV in the path descriptor.

3. Compute the physical disk address from the logical sector number.

4. Initiate the read operation.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it isn't clear, the driver goes back to sleep.

Whenever LSN 0 is read, the first part of this sector must be copied into the proper drive table. (Get the drive number from PD.DRV in the path descriptor.) The number of bytes to copy is in DD.SIZ.

The drive number (PD.DRV) should be used to compute the offset to the corresponding drive table as follows:

```
LDA PD.DRV,Y        Get the drive number
LDB #DRVMEN         Get the size of a drive table
MUL
LEAX DRVBEG,U       Get the address of the first table
LEAX D,X            Compute the address of Table n
```

Write

Function:

Write a 256-byte sector buffer to the disk.

Entry Conditions:

U	= *address of the device memory area*
Y	= *address of the path descriptor*
B	= *MSB of the disk LSN*
X	= *LSBs of the disk LSN*

Exit Conditions:

The sector buffer is written out to the disk.

If error:

CC	= C bit set
B	= *error code*

Technical Function:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.DRV in the path descriptor.

3. Compute the physical disk address from the logical sector number.

4. Initiate the write operation.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending the wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it is not, the driver goes back to sleep. If the disk controller cannot be interrupt-driven, it is necessary to perform a programmed I/O transfer.

6. If PF.VFY in the path descriptor is equal to zero, read the sector back in and verify that it was written correctly. This usually does not involve a comparison of the data.

If disk writes are to be verified, the Init routine must request the buffer where the sector may be placed when it is read back in. Do not copy LSN 0 into the drive table when it is read back to be verified.

Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the Read routine.

Getsta
Setsta

Function:

Get/set the device's operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

    U   = *address of the device memory area*
    Y   = *address of the path descriptor*
    A   = *status code*

Exit Conditions:

    Depend upon the function code

    If error:
    CC   = C bit set
    B   = *error code*

It may be necessary to examine or change the register stack that contains the values of the 6809 register at the time of the call. The address of the register stack is in PD.RGS, which is located in the path descriptor.

The following offsets may be used to access any value in the register stack:

Name	Relative Address	Size (Bytes)	6809 Register
R$CC	$00	1	Condition Code Register
R$D	$01	.	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program Counter

Term

Function:

Terminate a device. This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).

Entry Conditions:

U   = *address of the device memory area*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Technical Function:

1. Wait until any pending I/O is completed.

2. Disable the device interrupts.

3. Remove the device from the IRQ polling list.

4. If the Init routine reserved a 256-byte buffer for verifying disk writes, return the memory with the Return Sysmem system call (F$srtmem).

IRQ Service Routine

This routine is not included in the device driver's branch table and is not called directly by RBFMAN, but it is a key routine in interrupt-driven device drivers.

The IRQ service routine services interrupts and, when I/O is complete, sends a wakeup signal to the process whose process ID is in V.WAKE. It also clears V.WAKE as a flag to the mainline program that the IRQ has indeed occurred.

When it finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.

Technical Function:

1. Service the device interrupts (receive data from device or send data to it). This routine should put its data into and get its data from buffers which are defined in the device memory area.

2. Wake up any process that is waiting for I/O to complete. To do this, it checks to see if there is a process ID in V.WAKE (non-zero); if so, it sends a wakeup signal to that process.

3. If the device is ready to send more data and the output buffer is empty, disable the device's "ready to transmit" interrupts.

4. If a pause character is received, set V.PAUS in the attached device storage area to a non-zero value. The address of the attached device memory area is in V.DEV2.

Boot (Bootstrap Module)

Function:

Load the Boot file into memory from mass-storage.

Entry Conditions:

None

Exit Conditions:

D   = *size of the boot file* (in bytes)
X   = *address where the boot file was loaded into memory*

If error:
CC   = *C bit set*
B   = *error code*

The Boot module is **not** part of the disk driver. It is a separate module which is normally co-resident with the OS9P2 module in the system firmware.

The bootstrap module contains one subroutine that loads the bootstrap file and some related information into memory. It uses the standard executable module format with a module type of "system" (code $C). The execution offset in the module header contains the offset to the entry point of this subroutine.

It obtains the starting sector number and size of the OS9Boot file from LSN 0. OS-9 is called to allocate a memory area large enough for the Boot file, and then it loads the Boot file into this memory area.

Technical Function:

1.   Read LSN 0 from the disk into a buffer area. Boot must pick its own buffer area. LSN 0 contains the values for DD.BT (the 24-bit LSN of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes).

2.   Get the 24-bit LSN of the bootstrap file from DD.BT.

3.   Get the size of the bootstrap file from DD.BSZ. The Boot is contained in one logically contiguous block beginning at the logical sector specified in DD.BT and extending for (DD.BSZ/256 + 1) sectors.

4.   Use the OS-9 Request Sysmem system call (F$srqmem) to request the memory area where the Boot file will be loaded.

5.   Read the Boot file into this memory area.

6.   Return the size of the Boot file and its location.

6 / Sequential Character File Manager

The Sequential Character File Manager (SCFMAN) supports devices that operate on a character-by-character basis. These include terminals, printers, and modems.

It is a reentrant subroutine package called by IOMAN for I/O system calls to sequential, character-oriented devices. It includes the extensive I/O editing functions typical of line-oriented operation (backspace, line delete, repeat line, auto line feed, screen pause, and return delay padding).

OS-9 is supplied with SCFMAN and one SCR-type device driver module. The module is an RS-232-type, which runs the serial interface.

SCFMAN Line Editing Functions

Read and Write

The Read and Write system calls (I$read and I$write) to SCFMAN-type devices correspond to the BASIC09 GET and PUT statements. They pass data to/from the device with little modification.

> **Note:** Although there is otherwise little modification, the keyboard interrupt, keyboard abort, and pause character are filtered out of the input. (Editing is disabled if the corresponding character in the path descriptor contains a zero.)

Carriage returns are **not** followed by line feeds or nulls automatically, and the high order bits are passed as sent/received.

Read Line and Write Line

The Read Line and Write Line system calls (I$readln and I$writln) to SCFMAN devices correspond to the BASIC09 INPUT, PRINT, READ, and WRITE statements. They perform full line editing of all functions enabled for the particular device.

These functions are initialized when the device is first used. (The option table is copied from the device descriptor table associated with the specific device.)

Later, they may be altered — either from assembly-language programs using the Get Status system call, or from the keyboard using the Tmode command. All bytes transferred in this mode will have the high order bit cleared.

SCFMAN Definitions of the Path Descriptor

As you know, the PD.FST and PD.OPT sections of the path descriptor are reserved for and used by the file manager.

The table below describes the use of PD.FST and PD.OPT by SCFMAN. For your convenience, it also includes the other sections of the PD.

The PD.OPT section contains the values that determine the line editing functions. It contains many device operating parameters which may be read or written by the Set Status or Get Status system call. Any values not set by this table default to zero.

> **Note:** It is possible to disable most of the editing functions by setting the corresponding control character in the path descriptor to zero. You can use the Set Status system call or the Tmode command to do this. Or, you may go a step further by setting the corresponding control character value in the device descriptor module to zero.

To determine the default settings for specific devices, you may inspect the device descriptors.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID

PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

SCFMAN Path Descriptor Definitions (PD.FST Section)

PD.DV2	$0A	2	Device table address of the second (echo)device
PD.RAW	$0C	1	Edit flag: 0 = raw mode, 1 = edit mode
PD.MAX	$0D	2	Read Line maximum character count
PD.MIN	$0F	1	Devices are ''mine'' if cleared
PD.STS	$10	2	Status routine module address
PD.STM	$12	2	Reserved for status routine

Name	Relative Address	Size (Bytes)	Use
SCFMAN Option Section Definition (PD.OPT Section)			
(Copied from the device descriptor)			
	$20	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
PD.UPC	$21	1	Case: 0 = upper and lower, 1 = upper only
PD.BSO	$22	1	Backspace: 0 = backspace, 1 = backspace then space and backspace
PD.DLO	$23	1	Delete: 0 = backspace over line, 1 = carriage return/line feed
PD.EKO	$24	1	Echo: 0 = no echo
PD.ALF	$25	1	Auto line feed after carriage return: 0 = no auto line feed
PD.NUL	$26	1	End-of-line null count: 1 = nulls ($00) sent after each carriage return/line feed
PD.PAU	$27	1	Number of lines (since last input) before each end-of-page pause: 0 = no pause
PD.PAG	$28	1	Lines per page
PD.BSP	$29	1	Backspace character
PD.DEL	$2A	1	Delete-line character
PD.EOR	$2B	1	End-of-record character (end-of-line character) read only: Normally set to $0D; 0 = Terminate Read Line only at the end of the file

Name	Relative Address	Size (Bytes)	Use
PD.EOF	$2C	1	End-of-file character (read only)
PD.RPR	$2D	1	Reprint-line character
PD.DUP	$2E	1	Duplicate-last-line character
PD.PSC	$2F	1	Pause character
PD.INT	$30	1	Keyboard-interrupt character (CLEAR) (C)
PD.QUT	$31	1	Keyboard-abort character (CLEAR) (Q)
PD.BSE	$32	1	Backspace-echo character
PD.OVF	$33	1	Line-overflow character (bell)
PD.PAR	$34	1	Device-initialization value (parity)
PD.BAU	$35	1	Software settable baud rate
PD.D2P	$36	2	Offset to second device name string
PD.STN	$38	2	Offset of status routine name
PD.ERR	$3A	1	Most recent I/O error status

PD.EOF specifies the end-of-file character. If this is the first and only character input, SCFMAN returns an end-of-file error on Read or Readln.

PD.PSC specifies the pause character, which suspends output before the next end-of-record character. The pause character also deletes any type-ahead input for Readln.

PD.INT specifies the keyboard-interrupt character. When the character is received on input, a keyboard interrupt signal is sent to the last user of this path. The character also terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code.

PD.QUT specifies the keyboard-abort character. When this character is received on input, a keyboard abort signal is sent to the last user of this path. It also terminates the current I/O request (if any) with an error code identical to the keyboard

SCF-Type Device Descriptor Modules

The chart below shows the use of the initialization table in the device descriptors for SCF-type devices. The values are those IOMAN copies from the device descriptor to the path descriptor.

An SCF editing function is turned off if its corresponding value is set to zero. For example, if IT.EOF is set to zero, there is no end-of-file character.

Name	Relative Address	Size (Bytes)	Use
TABLE	.		Beginning of the option table
IT.DVC	$12	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SCF
IT.UPC	$13	1	Case: 0 = upper and lower case, 1 = upper only
IT.BSO	$14	1	Backspace: 0 - backspace; 1 = backspace then space and back-space
IT.DLO	$15	1	Delete: 0 = backspace over line, 1 = carriage return
IT.EKO	$16	1	Echo: 0 - no echo
IT.ALF	$17	1	Auto line feed: 0 = no auto line feed

Name	Relative Address	Size (Bytes)	Use
IT.NUL	$18	1	End-of-line null count
IT.PAU	$19	1	Pause: 0 = no end-of-page pause
IT.PAG	$1A	1	Lines per page
IT.BSP	$1B	1	Backspace character
IT.DEL	$1C	1	Delete-line character
IT.EOR	$1D	1	End-of-record character
IT.EOF	$1E	1	End-of-file character
IT.RPR	$1F	1	Reprint-line character
IT.DUP	$20	1	Duplicate-last-line character
IT.PSC	$21	1	Pause character
IT.INT	$22	1	Interrupt character
IT.QUT	$23	1	Quit character
IT.BSE	$24	1	Backspace echo character

IT.OVF	$25	1	Line-overflow character (bell)
IT.PAR	$26	1	Initialization value (parity) used to initialize the device's control register when a path is opened to it
IT.BAU	$27	1	Baud rate
IT.D2P	$28	2	Attached device name string offset
IT.STN	$2A	2	Offset to status routine
IT.ERR	$2C	1	Initial error status

SCF-Type Device Driver Modules

A SCFMAN-type device driver module contains a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. For each "incarnation" of the driver, IOMAN allocates a permanent memory area for that device.

The size of the memory area is given in the device driver module header. IOMAN and SCFMAN use some of this area. The device driver may use the rest in any way (typically as variables and buffers). The area is used as follows:

SFC Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use	
V.PAGE	$00	1	Port extended address	24-bit device
V.PORT	$01	2	Device base address	address (defined by IOMAN)
V.LPRC	$03	1	ID of the last active process	
V.BUSY	$04	1	ID of the active process: 0 = not busy (defined by RBFMAN)	
V.WAKE	$05	1	ID of the process to reawaken after the device completes I/O: 0 = no process is waiting (defined by the device driver)	
V.USER		.	End of the OS-9 definitions	
V.TYPE	$06	1	Device type of parity	
V.LINE	$07	1	Lines left until the end of the page	
V.PAUS	$08	1	Pause request: 0 = no pause	
V.DEV2	$09	2	Attached device memory area	
V.INTR	$0B	1	Interrupt character	
V.QUIT	$0C	1	Quit character	
V.PCHR	$0D	1	Pause character	
V.ERR	$0E	1	Error accumulator	
V.SCF	$0F	1	End of the SCFMAN definitions	
FREE		.	Free for the device driver to use	

V.LPRC contains the process-ID of the last process to use the device. The IRQ service routine is responsible for sending this process the proper signal in case a quit character or an interrupt character is received. V.LPRC is defined by SCFMAN.

V.BUSY contains the process ID of the process that is using the device. (If the device is not being used, V.BUSY contains a zero.) This is used by SCFMAN to prevent more than one process from using the device at the same time. V.BUSY is defined by SCFMAN.

SCFMAN Device Driver Subroutines

Like all device drivers, SCFMAN device drivers use a standard executable memory module format. In the header the type is specified as code $0E (device driver).

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY      LBRA      INIT      Initialize drive
           LBRA      READ      Read sector
           LBRA      WRITE     Write sector
           LBRA      GETSTA    Get status
           LBRA      SETSTA    Set status
           LBRA      TERM      Terminate device
```

Each subroutine should exist with the C bit in the Condition Code Register cleared if no error occurred. If an error occurred, the C bit should be set and an appropriate error code returned in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init

Function:

Initialize the device control registers (enable interrupts, if necessary).

Entry Conditions:

U = *address of the device memory area*
Y = *address of the device descriptor*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Technical Function:

1.   Initialize the device memory area.

2.   Place the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F$irq).

3.   Initialize the device control registers.

Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These contain the device address.) There is no need to initialize the part of the memory area used by IOMAN and SCFMAN.

Read

Function:

Read the next character from the input buffer.

Entry Conditions:

U   = *address of the device memory area*
Y   = *address of the path descriptor*

Exit Conditions:

A   = *character read*
If error:
CC   = C bit set
B   = *error code*

Technical Function:

1. Get the next character from the input buffer.

2. If no data is ready, Read copies its process ID from V.BUSY into V.WAKE. It then uses the Sleep system call to put itself to sleep.

   Later, when data is received, the Set IRQ system call leaves the data in a buffer, then checks V.WAKE to see if any process is waiting for the device to complete I/O. If so, Set IRQ should send a wakeup signal to it.

Data buffers are **not** allocated automatically. If a buffer is used, it should be defined in the device memory area.

Write

Function:

Output a character (place a data byte into an output buffer) and enable the device output interrupts.

Entry Conditions:

U   = *address of the device memory area*
Y   = *address of the path descriptor*
A   = *character to write*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

1. If the data buffer is already full, Write copies its process ID from V.BUSY into V.WAKE. It then puts itself to sleep.

2. Later, when the IRQ service routine transmits a character and makes room for more data, Write checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, it sends a wakeup signal to that process.

Write must ensure that the IRQ service routine will start up when data is placed into the buffer. After an interrupt is generated, the IRQ service routine continues to transmit data until the data buffer is empty. Then, it disables the device's "ready to transmit" interrupts.

Data buffers are **not** automatically allocated. If a buffer is used, it should be defined in the device memory area.

Getsta
Setsta

Function:

Get/set the device's operating parameter (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

U = *address of the device memory area*
Y = *address of the path descriptor*
A = *status code*

Exit Conditions:

Depend upon function code

Currently, all of the function codes defined by Microware for SCF-type devices are handled by IOMAN or SCFMAN. Any codes not defined by Microware are passed to the device driver.

It may be necessary to examine or change the register stack that contains the values of the 6809 registers at the time of the call. The address of the register stack may be found in PD.RGS, which is located in the path descriptor.

The following offsets may be used to access any value in the register packet:

Name	Relative Address	Size (Bytes)	6809 Register
R$CC	$00	1	Conditions Code Register
R$D	$01	.	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program Counter

Term

Function:

Terminate a device. This routine is called when a device is no longer in use (when the link count of its device descriptor module becomes zero).

Entry Conditions:

U   = *pointer to the device memory area*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Technical Function:

1. Wait until the output buffer is emptied (by the IRQ service routine).

2. Disable the device interrupts.

3. Remove the device from the IRQ polling list.

Note: Permanent storage used by device drivers is never returned to the free memory pool. Therefore, you should **not** terminate any device that might be used again. Modules contained in the Boot file will **never** be terminated.

IRQ Service Routine

This routine is not included in the device driver's branch table and is not called directly by SCFMAN, but it is a key routine in device drivers.

The IRQ service routine services device interrupts, and when I/O is complete, it sends a wakeup signal to the process whose process ID is in V.WAKE. It also clears V.WAKE as a flag to the mainline program that the IRQ has occurred.

When it finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.

For technical information, see "IRQ Service Routine" in *Chapter 5*.

7 / Assembly-Language Programming Techniques

There are four key rules to follow when you write OS-9 assembly-language programs:

- The program must use position-independent code (PIC). OS-9 selects program load addresses based on available memory at runtime. You cannot select them.

- The program must use the standard OS-9 memory module format; otherwise, it cannot be loaded and run. It cannot use self-modifying code. It must not change anything in a memory module or use any part of the module for variables.

- Storage for all variables and data structures must be within the data area OS-9 assigns at runtime. This area is separate from the program memory module.

- All input and output operations should be made using OS-9 system calls.

The 6809's versatile addressing modes make it easy for you to follow these rules. The OS-9 assembler also helps; it has special capabilities to assist you in creating programs and memory modules for the OS-9 execution environment.

How to Write Position-Independent Code

The 6809 instruction set allows efficient use of position-independent code.

The basic technique is to always use PC-relative addressing — for example, BRA, LBRA, BSR and LBSR. Get addresses of constants and tables using LEA instructions instead of load immediate instructions. If you use dispatch tables, use tables of relative addresses, not absolute addresses.

Incorrect	Correct
LDX #CONSTANT	LEAX CONSTANT,PCR
JSR SUBR	BSR SUBR or LBSR SUBR
JMP LABEL	BRA LABEL or LBRA LABEL

Addressing Variables and Data Structures

A program that is executed as a process (by the Fork and Chain system calls or by the shell) is assigned part of RAM for variables, stacks, and data structures at execution time. The RAM area cannot be specified or determined ahead of time. However, a minimum size for the area is specified in the program's module header.

When the program is first entered, Register Y contains the address of the top bound of the process's data memory area. Register U contains the address of the lower bound, and Register DP contains the area's page number.

The creating process may have passed a parameter area. If so, this parameter area is located from the value of X and the SP to the top of memory (Register Y), and Register D contains the area's size in bytes.

If the new process was called by the shell, the parameter area contains the part of the shell command line that includes the argument (parameter) text. (I/O redirection arguments are not included.)

The most important rule is this: **Do not use extended addressing.** You should use only indexed and direct page addressing to access data area values and structures. Do not use program-counter relative addressing to find addresses in the data area, but do use it to refer to addresses within the program area.

The most efficient way to handle tables, buffers, and stacks is to have the program's initialization routine compute their absolute addresses. It does this, using the data area bounds passed by OS-9 in the registers.

The absolute addresses then can be saved in the direct page where they can be loaded into registers quickly, using short instructions. Direct page addressing has these advantages: it is faster than extended addressing, and the program is inherently reentrant.

Stack Requirements

OS-9 uses interrupts extensively, and many reentrant 6809 programs use the 6809 register stack for local variable storage. Because of this, a generous stack should be maintained at all times. We recommend no fewer than 200 bytes.

Interrupt Masks

User programs should keep the F and I bits (FIRQ mask and IRQ mask) of the Condition Codes Register off. To avoid task-switching or interrupts during critical program sequences, you can set these bits. You should set them for no longer than a tick, however. Otherwise, system time-keeping may no longer be accurate.

Using Standard I/O Paths

Write your programs to use standard I/O paths wherever practical. Usually, this involves I/O calls that are intended to communicate to the user's terminal, or any other case where the OS-9 redirected I/O capability is desirable.

All three standard I/O paths are open when the program is entered; they are inherited from the parent process. Programs should **not** close these paths except under special circumstances.

The standard I/O paths are assigned as follows:

- Path 0 — Standard input. Analogous to the keyboard or other main data input source.

- Path 1 — Standard output. Analogous to the terminal display or other main data output destination.

- Path 2 — Standard error/status. This path is provided so output messages that are not part of the actual program output can be kept separate. Paths 1 and 2 are often directed to the same device.

Writing Interrupt-Driven Device Drivers

OS-9 programs do not use interrupts directly. Any interrupt-driven function should be implemented as a device driver module which should handle all interrupt-related functions. When a program must be synchronized to an interrupt-causing event, a driver can send a semaphore to a program (or vice versa), using OS-9's signal facilities.

It is important to understand that interrupt service routines are asynchronous and are not distinct processes. They are, in effect, subroutines called by OS-9 when an interrupt occurs.

Therefore, all interrupt-driven device drivers have two basic parts: the "mainline" subroutines that execute as part of the calling process, and a separate interrupt service routine. The two routines are asynchronous and therefore must use signals for communications and coordination.

The Init initialization subroutine within the driver package should do the following:

1. Allocate memory area for the service routine

2. Get the service routine address

3. Execute the Set IRQ system call to add it to the IRQ polling table

When a device driver routine does something that results in an interrupt, it should immediately execute a Sleep system call. This turns off the process.

When the interrupt occurs, its service routine is executed after some random interval. It should then do the least amount of processing required, and send a wakeup signal to its associated process using the Send system call. It may also put some data in its memory area (I/O data and status), which is shared with its associated sleeping process.

Later, the signal awakens the device driver mainline routine. The routine can now process the data or status returned by the interrupt service routine.

The OS-9 List utility command program is shown on this and the next page as an example of assembly-language programming.

LIST — OS-9 System Symbol Definitions

```
00001          **********
00002          *
00003          * LIST UTILITY COMMAND
00004          * Syntax:  list ·pathname·
00005          * COPIES INPUT FROM SPECIFIED FILE TO STANDARD OUTPUT
00006          *
00007                          NAM   LIST
00008          * NOTE: A USE /D0/DEFS/OS9DEFS
00009          *        IS ENCLOSED WITHIN THE IFP1/ENDC STATEMENT
00010                          IFP1
00012                          ENDC
00013    0011          TYPE    SET   PRGRM+OBJCT
00014    0081          REVS    SET   REENT+1
00015    0000  87CD004E        MOD   LSTEND,LSTNAM,PRGRM+OBJECT,
                                      REENT+1,LSTENT,
00016          * NOTE THE `MOD´ CMD CONSISTS OF:
00017          * LSTEND,LSTNAM,RPGRM+OBJECT,REENT+1,LSTENT,LSTMEM
00018    000D  4C4953D4  LSTNAM   FCS    "LIST"
00019          *
00020          * STATIC STORAGE OFFSETS
00021          *
00022    00C8          BUFSIZ   equ    200     size of input buffer
00023 D  0000                   org    0
00024 D  0000          IPATH    rmb    1       input path number
00025 D  0001          PRMPTR   rmb    2       parameter pointer
00026 D  0003          BUFFER   rmb    BUFSIZ  allocate line buffer
00027 D  00CB                   rmb    200     allocate stack
00028 D  0193                   rmb    200     room for parameter
                                                list
00029 D  025B          LSTMEM   EQU    .
00030    0011  9F01    LSTENT   stx    PRMPTR  save parameter ptr
00031    0013  8601             lda    #read   select read access
                                                mode
00032    0015  103F84           os9    I$open  open input file
00033    0018  252E             bcs    LIST50  exit if error
00034    001A  9700             sta    IPATH   save input path
                                                number
00035    001C  9F01             stx    PRMPTR  save updated param
                                                ptr
00036    001E  9600    LIST20   lda    IPATH   load input path
                                                number
00037    0020  3043             leax   BUFFER,U load buffer ptr
00038    0022  108E00C8         ldy    #BUFSIZ maximum bytes to be
                                                read
```

```
00039    0026    103F8B              os9     I$readln    read line of input
00040    0029    2509                bcs     LIST30      exit if error
00041    002B    8601                lda     #1          load std. out path
00042    002D    103F8C              os9     I$writln    output the line
00043    0030    24EC                bcc     LIST20      repeat if no error
00044    0032    2014                bra     LIST50      exit if error
00045    0034    C1D3    LIST30      cmpb    #E$EOF      at end of file?
00046    0036    2610                bne     LIST50      branch if not
00047    0038    9600                lda     IPATH       load input path
                                                          number
00048    003A    103F8F              os9     I$close     close input path
00049    003D    2509                bcs     LIST50      ..exit if error
00050    003F    9E01                ldx     PRMPTR      restore param ptr
00051    0041    A684                lda     0,X
00052    0043    810D                cmpa    #$0D        end of parameter
                                                          line?
00053    0045    26CA                bne     LSTENT      ..no; list next file
00054    0047    5F                  clrb
00055    0048    103F06    LIST50    os9     F$exit      ...terminate
00056    004B    BF7690              emod                module crc
00057    004E              LSTEND    equ     *
00058                                end

00000 error(s)
00000 warning(s)
```

LIST — OS-9 System Symbol Definitions

```
$004E   00078 program bytes generated
$025B   00603 data bytes allocated
$1CAE   bytes used for symbols
```

73

8 / System Calls

System calls are used to communicate between the OS-9 operating system and assembly-language programs. There are two major types of calls — I/O calls and function calls. Function calls include user mode calls and system mode calls.

Each system call has a mnemonic name. Names of I/O calls start with I$. For example, the Change Directory call is I$chgdir. Names of function calls start with F$. For example, the Allocate Bits call is F$allbit. The names are defined in the assembler-entry conditions equate file called OS9Defs.

The I/O calls are:

Attach	Make Directory
Change Directory	Open Path
Close Path	Read
Create File	Read Line
Delete File	Seek
Detach Device	Set Status
Duplicate Path	Write
Get Status	Write Line

The user calls are:

Allocate Bits	Parse Name
Chain	Print Error
Compare Names	Search Bits
CRC	Send
Deallocate Bits	Set Priority
Exit	Set SVC
Fork	Set SWI
Get ID	Set Time
Intercept	Sleep
Link	Time
Load	Unlink
Memory	Wait

The system mode calls are:

Allocate 64	Request Sysmem
Find 64	Return 64
I/O Delete	Return Sysmem
I/O Queue	Set IRQ
Insert Process	Verify Module
Next Process	

System mode calls are privileged. They may be executed only while OS-9 is in the system state (when it is processing another system call, executing a file manager or device driver, and so on).

System mode calls are included in this manual primarily for programmers who will be writing device drivers and other system-level applications. In the "System Call Descriptions" section of this chapter, system mode calls are listed separately from I/O and user calls.

Appendices A and B, the system call quick reference lists, summarize each system call's function.

Calling Procedure

All system calls are executed via an **SWI2** instruction.

1. Load the 6809 register with any appropriate parameters.

2. Execute a SWI2 instruction, followed immediately by a constant byte, which is the request code.

3. After OS-9 processes the call, it returns any parameters in the 6809 register. If an error occurred, the C bit of the Condition Code Register is set, and Accumulator B contains the appropriate error code. This permits a BCS or BCC instruction immediately following the system call to branch on error/no error.

As an example, here is the Close system call:

```
LDA    PATHNUM
SWI2
FCB    $8B
BCS    ERROR
```

You can use the assembler's "OS9" directive to simplify the call. Here is the simplified Close system call:

```
LDA    PATHNUM
OS9    I$CLOSE
BCS    ERROR
```

OS-9 accepts system calls in **any combination of upper- or lower-case letters.**

I/O System Calls

OS-9's I/O calls are easier to use than many other systems' I/O calls. This is because the calling program does not have to allocate and set up "file control blocks", "sector buffers," and so on.

Instead, OS-9 returns a 1-byte path number when a path to a file/device is opened or created. Until the path is closed, this path number may be used in later I/O requests to identify the file or device.

In addition, OS-9 allocates and maintains its own data structures. You need not deal with them.

System Call Descriptions

The rest of this chapter consists of the system call descriptions. At the top of each description is the system call name, followed by its mnemonic and code. Next is a summary of the call's function, entry conditions, and exit conditions. When further explanation is required, it is included under a section called "Technical Function."

In the system call descriptions, registers not specified as entry or exit conditions are not altered. Strings passed as parameters are normally terminated by having Bit 7 of the last character set, a space character, or an end-of-line character.

If an error occurs on a system call, the C bit of Register CC is set, and Register B contains the *error code*. If no error occurs, the C bit is clear, and Register B contains a value of zero.

Allocate Bits

OS9 F$allbit

Function:

Sets bits in the allocation bit map specified by Register X.

Bit numbers range from 0 to n-1, where n is the number of bits in the allocation bit map.

Entry Conditions:

X   = *starting address of the allocation bit map*
D   = *number of the first bit to set*
Y   = *number of bits to set*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Attach

OS9 I$attach

Function:

Attaches a new device to the system or verifies that a device is attached.

Attach does not "reserve" the device. It only prepares it for later use by any process.

Most devices are installed automatically. Therefore, you need to use Attach only when installing a device dynamically or when verifying the existence of a device. You need not use the Attach system call to perform routine I/O.

The *access mode* parameter specifies the read and/or write operations to be allowed. These are:

    0   = Use device capabilities
    1   = Read
    2   = Write
    3   = Update

Entry Conditions:

    X   = *address of the device name string*
    A   = *access mode*

Exit Conditions:

    U   = *address of the device table entry*

    If error:
    CC   = C bit set
    B   = *error code*

Technical Function:

1. OS-9 searches the system module to see if memory contains a device descriptor that has the same name as the device.

2a. If it finds the descriptor and if the device is not already attached, OS-9 links to its file manager and device driver. It then places their addresses in a new device table entry. OS-9 then allocates any memory needed by the device driver and calls the driver's initialization routine (which usually initializes the hardware).

2b. If it finds the descriptor and if the device is already attached, OS-9 does not reinitialize the device.

Chain

OS9 F$chain

Function:

Loads and executes a new primary module, but does not create a new process. A Chain system call is similar to a Fork followed by an Exit, but with less processing overhead.

Chain "resets" the calling process's program and data memory areas and begins executing a new primary module. It does not affect open paths.

Warning: The hardware stack pointer (Register SP) should be located somewhere in the direct page before the Chain is executed. This helps prevent a "suicide" (system crash) or a "suicide attempt" error. It also prevents a suicide in the event that the new module requires a smaller data area than that in use. You should allow approximately 200 bytes of stack space for execution of the Chain system call.

Entry Conditions:

X	= *address of the module name or filename*
Y	= *parameter area size* (in pages); defaults to zero if not specified
U	= *starting address of the parameter area*
A	= *language/type code*
B	= *size of the data area* (in pages); must be at least one page

Exit Conditions:

None

If error:

CC	= C bit set
B	= *error code*

Technical Function:

1. OS-9 passes the name string of the new process's primary module (the program that is to be executed first). Then, it searches the system module directory to see if a module with the same name, type, and language is already in memory.

2a.   If the module is in memory, it is linked to.

2b.   If the module is not in memory, the name string is used as the pathlist of a file which is to be loaded into memory. Then, the first module in this file is "linked to." (Several modules may have been loaded from a single file.)

3.   OS-9 unlinks the process's old primary module.

4.   The data memory area is reconfigured to the size specified in the new primary module's header.

The diagram below shows how Chain sets up the data memory area and registers for the new module.
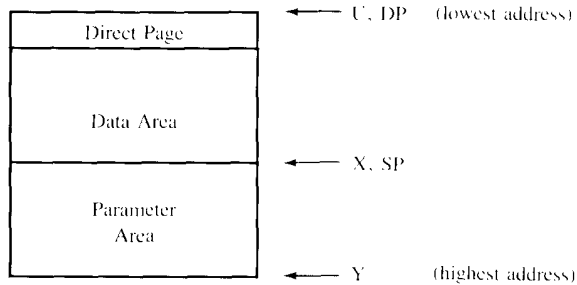
Direct Page	←—— U, DP (lowest address)
Data Area	
	←—— X, SP
Parameter Area	←—— Y (highest address)

D   = *parameter area size*
PC   = *module entry point absolute address*
CC   = F = 0, I = 0; others are undefined

Registers Y and U (the top-of-memory and bottom-of-memory pointers, respectively) always have values at page boundaries. If the parent process does not specify a size for the parameter area, the size (Register D) defaults to zero. The data area must be at least one page long.

(For more information, see the Fork system call.)

Change Directory

OS9 I$chgdir 103F 86

Function:

Changes a process's working directory to the directory specified by the pathlist.

If the access mode is read, write, or update, the current data directory is changed. If the access mode is execute, the current execution directory is changed.

The calling process must have read access to the directory specified (public read if the directory is not owned by the calling process).

The *access modes* are:

    1   = Read
    2   = Write
    3   = Update
    4   = Execute

Entry Conditions:

    X   = *address of the pathlist*
    A   = *access mode*

Exit Conditions:

    None

    If error:
    CC   = C bit set
    B   = *error code*

Close Path

OS9 I$close

8F

Function:

Terminates the I/O path to the file or device specified by the *path number*. Unless you use another Open or Create system call, you can no longer perform I/O to the file or device.

Non-sharable devices become available to other requesting processes. All OS-9 internally managed buffers and descriptors are deallocated.

The Exit system call automatically closes all open paths (except the standard I/O paths). Therefore, you may not need to use the Close Path system call to close them.

Do not close standard I/O paths unless you want to change the files or devices to which they correspond.

Entry Conditions:

> A   = *path number*

Exit Conditions:

> None

> If error:
> CC   = C bit set
> B   = *error code*

85

Compare Names

OS9 F$cmpnam

Function:

Compares two strings and indicates whether they match. Use this call with the Parse Names system call.

The second name must have the most significant bit (Bit 7) of the last character set.

Entry Conditions:

X	= *address of string1*
B	= *length of string1*
Y	= *address of string2*

Exit Conditions:

CC	= C bit clear if the strings match

If error:

CC	= C bit set
B	= *error code*

CRC

Function:

Calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators.

The calculation begins at the *starting byte address* and continues over the specified *number of bytes*.

You need not cover an entire module in one call, since the CRC may be "accumulated" over several calls. The CRC accumulator can be any 3-byte memory area and must be initialized to $FFFFFF before the first CRC call.

The last three bytes in the module are not included in the CRC generation. The three CRC bytes are to be stored here.

Entry Conditions:

X	= *starting byte address*
Y	= *number of bytes*
D	= *address of the 3-byte CRC accumulator*

Exit Conditions:

The CRC accumulator is updated.

If error:
None

Create File

OS9 I$create

Function:

Creates and opens a file on a disk.

OS-9 parses the pathlist and enters the new filename in the specified directory — or the working directory, if none is specified.

The file is given the attributes passed in Register B, which has bits defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Sharable file

The *access mode* parameter passed in Register A must be either write or update. This mode affects the file only until it is closed. The file can be reopened in any access mode allowed by the file attributes (see the Open system call).

Files opened for write may allow faster data transfer than those opened for update because update sometimes needs to pre-read sectors. These access codes are defined as follows: Bit 2 = write; Bit 3 = update.

> **Note:** If the execute bit (Bit 2) is set, the file is created in the working execution directory instead of the working data directory.

The path number returned by OS-9 is used to identify the file in later I/O system calls until the file is closed.

No data storage is initially allocated for the file at the time it is created. This is done automatically by the Write subroutine or explicitly by the Setsta subroutine.

If the filename already exists in the directory, an error occurs. If the call specifies a non-multiple file device (such as a printer or terminal), Create behaves the same as an Open.

You cannot use Create to make directories. (See the Make Directory system call for instructions on how to do this.)

Entry Conditions:

X   = *address of the pathlist* (see example below)
A   = *access mode*
B   = *file attributes*

Exit Conditions:

X   = *address of the last byte of the pathlist* + 1;
          any trailing blanks are skipped (see example
          below)
A   = *path number*

If error:
CC   = C bit set
B   = *error code*

Example:

Before the Create File call:

/	D	0	/	W	0	R	K	$0D

↑
X
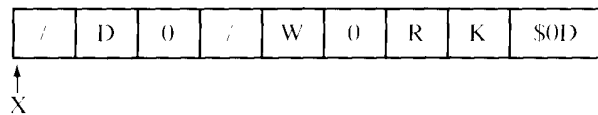
After the Create File call:

/	D	0		W	0	R	K	$0D

↑
X
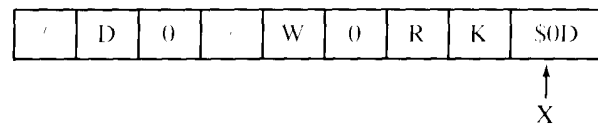
Deallocate Bits

OS9 F$delbit

Function:

Clears bits in the allocation bit map pointed to by Register X.

Bit numbers range from 0 to $n-1$, where n is the number of bits in the allocation bit map.

Entry Conditions:

X = *starting address of the allocation bit map*
D = *number of the first bit to set*
Y = *number of bits to set*

Exit Conditions:

None

If error:
CC = C bit set
B = *error code*

Delete File

OS9 I$delete

Function:

Deletes the disk file specified by the pathlist.

The file must have write permission attributes (public write if the calling process is not the owner).

An attempt to delete a device results in an error.

Entry Conditions:

    X   = *address of the pathlist* (see example below)

Exit Conditions:

    X   = *address of the last byte of the pathlist* + *1*; any trailing blanks are skipped (see example below)

    If error:
    CC   = C bit set
    B   = *error code*

Example:

Before the Delete File call:

/	D	0	/	W	0	R	K	b	b	b	M	E	M	O	$0D

↑
X
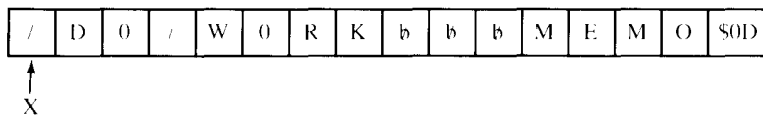
After the Delete File call:

/	D	0		W	0	R	K	b	b	b	M	E	M	O	$0D

↑
X
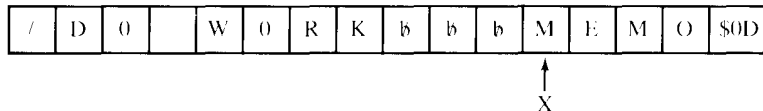
Detach Device

OS9 I$detach

Function:

Removes a device from the system and the system device table, if the device is not being used by another other process.

You must use this call to detach devices that were attached using the Attach system call. Both calls are used mainly by IOMAN. SCFMAN also uses Attach/Detach to set up its second device (echo device).

Entry Conditions:

>   U   = *address of the device table entry*

Exit Conditions:

>   None

>   If error:
>   CC   = C bit set
>   B   = *error code*

Technical Function:

1.   The device driver's termination routine is called, then OS-9 deallocates any memory assigned to the driver.

2.   The associated device driver and file manager modules are unlinked. If not being used by any other module, the driver is removed from RAM.

Duplicate Path

OS9 I$dup

Function:

Returns another, synonymous path number for the file or device specified by the *old path number*.

The shell uses this system call when it redirects I/O. System calls that use either path number (old or new) operate on the same file or device.

Entry Conditions:

A   = *old path number* (number of path to duplicate)

Exit Conditions:

B   = *new path number*

If error:
CC   = C bit set
B   = *error code*

93

Exit

OS9 F$exit

Function:

Terminates the calling process.

The Exit system call is the only way a process can "kill" itself. Exit deallocates the process's data memory area and unlinks its primary module. It also closes all open paths automatically.

The Wait system call always returns to the parent the status code passed by the child in its Exit call. Therefore, if the parent executes a Wait and receives the status code, it knows the child has "died."

Entry Conditions:

B = *status code to return to the parent*

Exit Conditions:

The process is terminated.

Fork

OS9 F$fork

Function:

Creates a new process, a "child" of the calling process. Fork also sets up the child process's memory and 6809 register.

Entry Conditions:

X = *address of the module name or filename* (see example below)
Y = *size of the parameter area* (in pages); defaults to zero if not specified
U = *starting address of the parameter area*
A = *language/type code*
B = *size of the optional data area* (in pages); must be at least one page

Exit Conditions:

X = *address of the last byte of the name + 1* (see example below)

If error:
CC = C bit set
B = *error code*

Example:

Before the Fork call:

T	E	S	T	$0D

↑
X
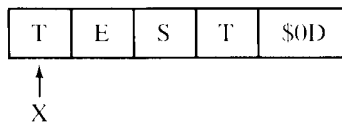
After the Fork call:
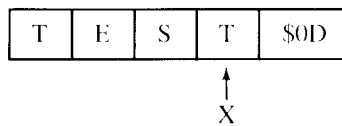
T	E	S	T	$0D

↑
X

Technical Function:

1. OS-9 passes the name string of the new process's "primary module" (the program that is to be executed first). Then, it searches the system module directory to see if the program already is in memory.

2a. If the program is in memory, the module is linked to and executed.

2b. If the program is not in memory, OS-9 uses the name as the pathlist of the file that is to be loaded into memory. Then, the first module in this file is linked to and executed. (Several modules may have been loaded from one file.)

3. OS-9 uses the primary module's header to determine the initial size of the process's data area. It then tries to allocate a contiguous RAM area of that size. (This area includes the parameter passing area, which is copied from the parent process's data area.)

4. The new process's registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point.

When the shell processes a command line, it passes a string in the parameter area. This string is a copy of the parameter part of the command line. To simplify string-oriented processing, it also inserts an end-of-line character at the end of the parameter string.

Register X points to the starting of the parameter string. If the command line included the optional memory size specification (#*n* or #*n*K), the shell passes that as the requested memory size when executing the Fork.

If any of the above operations is unsuccessful, the Fork is aborted and the caller is returned an error.

The diagram below shows how Fork sets up the data memory area and registers for a newly created process.

```
+------------------+        ◄——— U, DP   (lowest address)
|   Direct Page    |
+------------------+
|                  |
|    Data Area     |
|                  |
+------------------+        ◄——— X, SP
|                  |
|    Parameter     |
|      Area        |
+------------------+        ◄——— Y       (highest address)
```
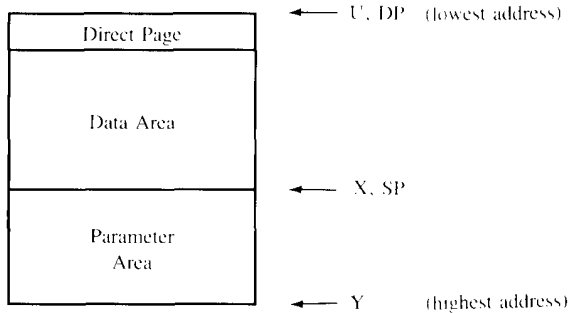
D   = *size of the parameter area*
PC   = *module entry point absolute address*
CC   = F = 0, I = 0, others are undefined

Registers Y and U (the top-of-memory pointer and bottom-of-memory pointer, respectively) always have values at page boundaries.

As stated earlier under "Entry Conditions," if the parent does not specify the size of the parameter area, the size defaults to zero. The minimum overall data area size is one page.

> **Note:** The child and parent processes execute at the same time. If the parent executes a Wait system call immediately after the Fork, it waits until the child "dies" before it resumes execution.

> Be careful when recursively calling a program that uses the Fork system call; another child may be created with each "incarnation." This continues until the process table becomes full.

Get ID

OS9 F$id

Function:

Returns the caller's *process ID* and *user ID*.

The *process ID* is a byte value from 1 to 255. It is assigned by OS-9 and is unique to the process.

The *user ID* is an integer from 0 to 65535. It is defined in the system password file, and is used by the file security system and a few other functions. Several processes can have the same user ID.

Entry Conditions:

> None

Exit Conditions:

> A   = *process ID*
> Y   = *user ID*
>
> If error:
> CC   = C bit set
> B   = *error code*

Get Status

Function:

Returns the status of a file or device.

This is a "wildcard" call. It is used to handle device parameters that:

- Are not the same for all devices
- Are highly hardware-dependent
- Must be user-changeable

The exact operation of the Get Status system call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's parameters for such functions as backspace character and echo on/off. The Get Status call is commonly used with the Set Status call.

The Get Status function codes that are currently defined are listed in the "Uses of Get Status" section below.

Entry Conditions:

A   = *path number*
X   = *MS 16 bits of the desired file position*
U   = *LS 16 bits of the desired file position*

Exit Conditions:

None

If error:
CC   = *C bit set*
B   = *error code*

Uses of Get Status

Function codes 7 through 127 are reserved for future use.

Codes 128 through 255 and their parameter-passing conventions are user-definable. (See the sections on writing device drivers). The function code and register stack are passed to the device driver.

The following function codes are defined: $00, $01, $02, $05, $06, $12, and $ 13. The parameter-passing conventions for these function codes are listed below.

SS.OPT (Function code $00): Reads the option section of the path descriptor, and copies it into the 32-byte area pointed to by Register X.

Use this to determine the current settings for editing functions, such as echo and auto line feed. For a complete description of the status packet, see the section on path descriptors.

Entry Conditions:
      A   = *path number*
      B   = $00
      X   = *address to receive status packet*

Exit Conditions:
      Status packet

      If error:
      CC   = C bit set
      B   = *error code*

SS.RDY (Function code $01): Tests for data available on SCFMAN-supported devices.

Entry Conditions:
      A   = *path number*
      B   = $01

Exit Conditions:
      If ready:
      CC   = C bit clear
      B   = $00

      If not ready:
      CC   = C bit set
      B   = $F6 (E$SRNDY)

      If error:
      CC   = C bit set
      B   = *error code*

SS.SIZ (Function code $02): Gets the current file size (RBFMAN-supported devices only).

Entry Conditions
    A   = *path number*
    B   = $02

Exit Conditions
    X   = *ms 16 bits of the current file size*
    U   = *ls 16 bits of the current file size*

    If error:
    CC   = C bit set
    B   = *error code*

SS.POS (Function code $05): Gets the current file position (RBFMAN-supported devices only).

Entry Conditions:
    A   = *path number*
    B   = $05

Exit Conditions:
    X   = *MS 16 bits of the current file position*
    U   = *LS 16 bits of the current file position*

    If error:
    CC   = C bit set
    B   = *error code*

SS.EOF (Function code $06): Tests for the end of the file (EOF).

Entry Conditions:
    A   = *path number*
    B   = $06

Exit Conditions:
    If not EOF:
    CC   = C bit clear
    B   = $00

    If EOF:
    CC   – C bit set
    B   – $D3 (E$EOF)

If error:
```
CC      = C bit set
B       = error code
```

SS.DSTAT (Function code $12): Returns the display status.

Entry Conditions:
```
A       = path number
B       = $12
```

Exit Conditions:
```
X       = address of the graphics display memory
Y       = graphics cursor address; x = MSB,
          y = LSB
A       = color code of the pixel at the cursor address
```

SS.JOY (Function code $13): Returns the joystick values.

Entry Conditions:
```
A       = path number
B       = $13
X       = 0 (right joystick), or
X       = 1 (left joystick)
```

Exit Conditions:
```
X       = selected joystick x value (0-63)
Y       = selected joystick y value (0-63)
A       = $FF (if the fire button is on), or
A       = $00 (if the fire button is off)
```

Intercept

OS9 F$icpt

Function:

Tells OS-9 to set a signal intercept trap. Then, whenever the process receives a signal, the process's intercept routine is executed.

Once the signal trap is set, the intercept routine may be executed at any time because a signal may occur at any time.

The intercept routine should terminate with an RTI instruction.

Note: If a process has not used the Intercept system call to set a signal trap, the process aborts if it receives a signal.

Entry Conditions:

X   = *address of the intercept routine*
U   = *starting address of the routine's memory area*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Technical Function:

1.   When the process receives a signal, OS-9 sets Registers U and B as follows:

U   = *starting address of the intercept routine's memory area*
B   = *signal code* (process's termination status)

Note: The value of Register DP may not be the same as it was when the Intercept call was made.

2.   After setting the registers, OS-9 transfers execution to the intercept routine.

Link

Function:

Links to a memory module that has the specified name, language, and type.

The module's "link count" is incremented whenever Link references its name. This keeps track of how many processes are using the module.

If the module requested is not sharable (not reentrant), only one process may link to it at a time.

Entry Conditions:

X = *address of the module name* (see example below)
A = *type/language byte*

Exit Conditions:

X = *address of the last byte of the module name + 1* (see example below)
Y = *module entry point absolute address*
U = *module header absolute address*
A = *type/language code*
B = *attributes / revision level*

If error:
CC = C bit set
B = *error code*

Example:

Before the Link call:

T	E	S	T	$0D

↑
X
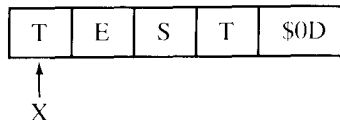
After the Link call:

T	E	S	T	$0D

↑
X
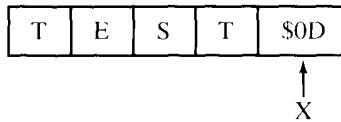
Technical Function:

1. OS-9 searches the module directory for a module that has the specified name, language, and type.

2a. If OS-9 finds the module, the address of the module's header is returned in Register U, and the absolute address of the module's execution entry point is returned in Register Y. (This and other information can be obtained from the module header.)

2b. If OS-9 doesn't find the module — or if the type/language codes in the entry and exit conditions don't match — OS-9 returns an error.

Possible Errors:

- Module not found
- Module busy (not sharable and in use)
- Incorrect or defective module header

Load

OS9 F$load

Function:

Loads a module or modules from the file specified by the pathlist or from the working execution directory (if no pathlist is given).

The file must have the execute access mode. It also must contain a module or modules that have a proper module header.

All modules loaded are added to the system module directory. The first module read is linked. The exit conditions apply only to the first module loaded.

Entry Conditions:

X   = *address of the pathlist (filename)* (see example below)
A   = *language/type code*; 0 = any language/type

Exit Conditions:

X   = *address of the last byte of the pathlist (filename) + 1* (see example below)
Y   = *primary module entry point address*
U   = *address of the module header*
A   = *language/type code*
B   = *attributes / revision level*

If error:
CC   = C bit set
B   = *error code*

Example:

Before the Load call:

/	D	0	/	A	C	C	T	S	R	C	V	$0D

↑
X
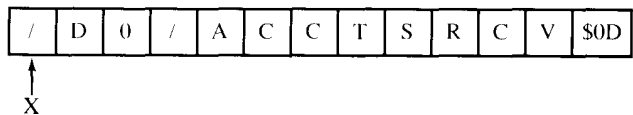
After the Load call:

/	D	0	/	A	C	C	T	S	R	C	V	$0D

↑
X
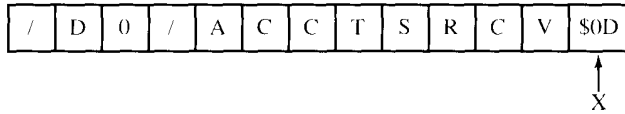
Possible errors:

- Module directory full
- Memory full
- Errors that occur on the Open, Read, Close, and Link system calls

Make Directory

OS9 I$makdir 103F 85

Function:

Creates and initializes a directory as specified by the pathlist. The directory contains no entries, except for an entry for itself (.) and its parent directory (..)

The caller is made the owner of the directory. Because the Make Directory call does not open the directory, it does not return a path number.

The new directory automatically has its "directory" bit set in the access permission *attributes*. The remaining attributes are specified by the byte passed in Register B. The bits are defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Sharable
7	Don't care

Entry Conditions:

    X   = *address of the pathlist* (see example below)
    B   = *directory attributes*

Exit Conditions:

    X   = *address of the last byte of the pathlist* + *1*; any trailing blanks are skipped (see example below)

    If error:
    CC   = C bit set
    B   = *error code*

Example:

Before the Make Directory call:

| / | D | 0 | / | N | E | W | D | I | R | $0D |
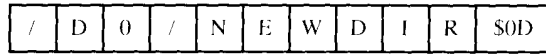
↑
X

After the Make Directory call:

| / | D | 0 | / | N | E | W | D | I | R | $0D |

↑
X
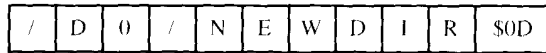
Memory

Function:

Expands or contracts the process's data memory area to the specified size. Or, if you specify zero as the new size, the call returns the current size of data memory.

The size requested is rounded off to the next page boundary. Additional memory is allocated contiguously upward or deallocated downward from the old highest address.

Even if enough free memory exists, the call may return an error upon an expansion request. This is because the data area is always made contiguous. Therefore, memory requests by other processes may fragment free memory into smaller, scattered blocks that are contiguous with the caller's present data area.

Entry Conditions:

D = *size of the new memory area* (in bytes);
0 = return current size and upper bound

Exit Conditions:

Y = *address of the new memory area upper bound*
D = *actual size of the new memory* (in bytes)

If error:
CC = C bit set
B = *error code*

Open Path

OS9 I$open

Function:

Opens a path to an existing file or device as specified by the pathlist.

OS-9 searches for the file in one of the following:

- The directory specified by the pathlist if the pathlist begins with a slash

- The working data directory, if the pathlist does not begin with a slash

- The working execution directory, if the pathlist does not begin with a slash and if the execution bit is set in the access mode

A path number is returned; it is used in later system calls to identify the file.

The *access mode* parameter specifies which read and/or write operations are to be permitted. When set, the bits allow access as follows:

Bit(s)	Access
Read	Read
Write	Write
Read and write	Update
Directory	Directory I/O

The access mode must conform to the access permission attributes associated with the file or device (see the Create system call). Only the owner may access a file unless the appropriate "public permit" bits are set.

The update mode may be slightly slower than the others because pre-reading of sectors may be required for random access of bytes within sectors.

Several processes (users) may open files at the same time. Each device has an attribute that specifies whether or not it is sharable.

111

Entry Conditions:

X = *address of the pathlist* (see example below)
A = *access mode* (D S PE PW PR E W R)

Exit Conditions:

X = *address of the last byte of the pathlist + 1* (see example below)
A = *path number*

If error:
CC = C bit set
B = *error code*

Example:

Before the Open Path call:

/	D	0	/	A	C	C	T	S	P	A	Y	$0D

↑
X
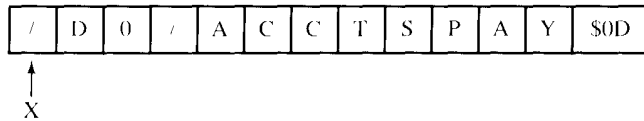
After the Open Path call:

/	D	0	/	A	C	C	T	S	P	A	Y	$0D

↑
X
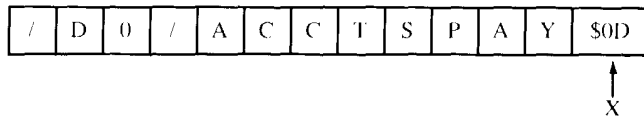
Parse Name

Function:

Parses (analyzes) the input text string for a legal OS-9 name. The name is terminated by any character that is not a legal name character.

This system call is useful for processing pathlist arguments passed to new processes.

Because Parse Name processes only one name, several calls may be needed to process a pathlist that has more than one name. As you can see from the example below, Parse Name finishes with Register X in position for the next parse.

If Register X was at the end of a pathlist, a bad name error is returned. Then, Register X is moved past any space characters so that the next pathlist in a command line may be parsed.

Entry Conditions:

> X = *address of the pathlist* (see example below)

Exit Conditions:

> X = *address of the optional slash + 1*
> Y = *address of the last character of the name + 1* (see example below)
> B = *length of the name*
>
> If error:
> CC = *C bit set*
> B = *error code*
> X = *address of the last trailing blank + 1*

Example:

Before the Parse Name call:

	D	O	/	P	A	Y	R	O	L	L	♭	♭	♭

↑
X
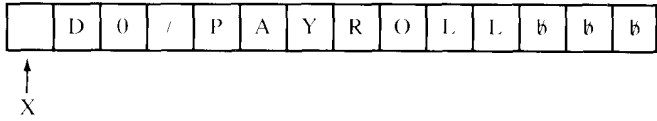
After the Parse Name call:

/	D	O		P	A	Y	R	O	L	L	♭	♭	♭

↑   ↑
X   Y   B = 2
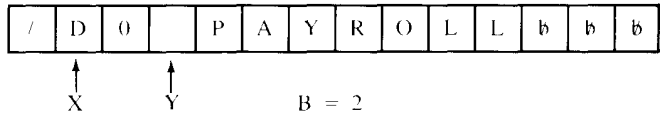
Print Error

OS9 F$perr

Function:

Writes an error message to the output path specified. By default,
OS-9 displays:

    ERROR #*decimal number*

The error reporting routine is vectored and can be replaced with
a more elaborate reporting module. To replace this routine use
the Set SVC system call.

Entry Conditions:

    A   = *output path number*
    B   = *error code*

Exit Conditions:

    None
    If error:
    CC   = C bit set
    B   = *error code*

Read

OS9 I$read

103F 89

Function:

Reads the specified number of bytes from the specified path. The data is returned exactly as read from the file/device without additional processing or editing.

The path must have been opened in the read or update mode.

After all data in a file is read, the next Read call returns an end-of-file error.

The keyboard abort, keyboard interrupt, and end-of-file characters may be filtered out of the entry conditions data on SCFMAN-type devices unless the corresponding entries in the path descriptor have been set to zero. You may want to modify the device descriptor so that these values are initialized to zero when the path is opened.

The number of bytes requested is read unless any of the following is true:

- An end-of-file occurs
- An end-of-record occurs (SCFMAN only)
- An error occurs

Entry Conditions:

X   = *address to store data*
Y   = *number of bytes to read*
A   = *path number*

Exit Conditions:

Y   = *number of bytes read*

If error:
CC   = C bit set
B   = *error code*

Read Line

OS9 I$readln
103F 8B

Function:

Reads a text line with editing.

Read Line is similar to Read except it reads the input file or device until a carriage return character is encountered or until the maximum byte count specified is reached. Line editing is also activated on character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.

SCFMAN requires that the last byte entered be an end-of-record character (usually a carriage return). If more data is entered than the maximum specified, it is not accepted and a PD.OVF character (usually bell) is echoed.

After all data in a file is read, the next Read Line call generates an end-of-file error.

(For more information about line editing, see "SCFMAN Line Editing Functions" in Chapter 6.)

Entry Conditions:

X   = *starting address to store data*
Y   = *maximum number of bytes to read*
A   = *path number*

Exit Conditions:

Y   = *number of bytes read*

If error:
CC   = C bit set
B   = *error code*

•

117

Search Bits

OS9 F$schbit

Function:

Searches the specified allocation bit map for a free block (cleared bits) of the required length.

The search starts at the *starting bit number*. If no block of the specified size exists, the call returns with the carry set, starting bit number, and size of the largest block.

Entry Conditions:

X	= *starting address of the map*
D	= *starting bit number*
Y	= *bit count (free bit block size)*
U	= *ending address of the map*

Exit Conditions:

D	= *starting bit number*
Y	= *bit count*

Seek

OS9 I$seek

Function:

Repositions the path's logical "file pointer," the 32-bit address of the next byte in the file to be read from or written to.

A seek may be performed to any value, regardless of the file's size. Later writes automatically expand the file to the required size (if possible). Reads, however, return an end-of-file condition. Note that a seek to Address 0 is the same as a rewind operation.

Seeks to non-random access devices usually are ignored and return without error.

Entry Conditions:

A   = *path number*
X   = *MS 16 bits of the desired file position*
U   = *LS 16 bits of the desired file position*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

119

Send

Function:

Sends a signal to the specified process. The signal code is a single byte value from 1 through 255.

If the destination process is sleeping or waiting, it is activated so that it may process the signal.

If a signal trap was set up, the signal processing routine (Intercept) is executed (see the Intercept system call). If none was set up, the signal aborts the destination process, and the signal code becomes the exit status (see the Wait system call). An exception is the wakeup signal; it does not cause the signal intercept routine to be executed.

Signal codes are defined as follows:

```
0        = System abort (cannot be intercepted)
1        = Wake up the process
2        = Keyboard abort
3        = Keyboard interrupt
4-255    = User defined
```

If you try to send a signal to a process that has a signal pending, the current Send call is cancelled, and an error is returned. Issue a Sleep call for a few ticks, then try again. (The Sleep call saves CPU time.)

(See the Intercept, Wait, and Sleep system calls for more information.)

Entry Conditions:

A   = *destination's process ID*
B   = *signal code*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Set Priority

OS9 F$sprior 103F 0D

Function:

Changes the process's priority to the *priority* specified.

A process can change another process's priority only if it has the same user ID.

Entry Conditions:

    A   = process ID
    B   = priority; 0 = lowest, 255 = highest

Exit Conditions:

    None

    If error:
    CC   = C bit set
    B   = error code

Set Status

OS9 I$setstt

Function:

Sets the status of a file or device.

This is a "wildcard" call. It is used to handle device parameters that:

- Are not the same for all devices
- Are highly hardware-dependent
- Must be user-changeable

The exact operation of the Set Status system call depends on the device driver and file manager associated with the path. A typical use is to set a terminal's parameters for such functions as backspace character and echo on/off. The Set Status call is commonly used with the Get Status call.

The Set Status function codes that are currently defined are listed in the "Uses of Set Status" section below.

Entry Conditions:

```
A       = path number
B       = function code
Other registers depend upon the function code
```

Exit Conditions:

```
Depend upon the function code

If error:
CC      = C bit set
B       = error code
```

Uses of Set Status

Function codes 128 through 255 and their parameter-passing conventions are user-definable. (For more information, see the "RBF-Type Device Drivers" section in Chapter 5 and the "SCF-Type Device Drivers" section in Chapter 6.) The function code and register stack are passed to the device driver.

The following function codes are defined: $00, $02, $04, $09, $0A, $0B, $0C, and $0D. The parameter-passing conventions for these function codes are listed below.

SS.OPT (Function code $00): Writes the option section of the path descriptor from the 32-byte status packet pointed to by Register X. Use this to set the device operating parameters, such as echo and line feed.

Entry Conditions:

    A   = *path number*
    B   = $00
    C   = *address of the status packet*

Exit Conditions:
    None

SS.SIZ (Function code $02): Changes the file's size (RBFMAN-type devices only).

Entry Conditions:

    A   = *path number*
    B   = $02
    X   = *MS 16 bits of the desired file size*
    U   = *LS 16 bits of the desired file size*

Exit Conditions:
    None

SS.RST (Function code $03): Restores the head to Track 0. This is used for formatting and error recovery.

Entry Conditions:

    A   = *path number*
    B   = $03

Exit Conditions:
    None

SS.WTK (Function code $04): Formats (writes) a track on a floppy disk. For hard disks or floppy disks that have a "format entire disk command." SS.WTK formats the entire disk only when the *track number* is zero.

Entry Conditions:

A	= *path number*
B	= $04
X	= *address of the track buffer*
U	= *track number* (least significant 8 bits)
Y	= *side/density*

Bit B0 = side (0 = Side 0, 1 = Side 1) Bit B1 = density (0 = single, 1 = double)

Exit Conditions:
    None

SS.FEE (Function code $09): Issues a form feed.

Entry Conditions:
    B   = $09

Exit Conditions:
    None

SS.FRZ (Function code $0A): Freezes the DD. information. (Inhibits the reading of LSN 0 to DDD.xxx variables, which define disk formats.) This enables the reading of non-standard disks.

Entry Conditions:
    B   = $0A

Exit Conditions:
    None

SS.SPT (Function code $0B): Sets a different number of sectors per track so non-standard disks may be read.

Entry Conditions:
    B   = $0B
    X   = *new sectors per track*

    Exit Conditions:
    None

SS.SQD (Function code $0C): Starts the power-down sequence for hard disks that have sequence-down requirements prior to removal of power.

Entry Conditions:
        B   = $0C

Exit Conditions:
        None

SS.DCM (Function code $0D): Transmits a command directly to a disk controller for specific functions. Parameters and commands are hardware-dependent for specific systems.

Entry Conditions:
        B   = $0D
        Other registers vary

Exit Conditions:
        Vary

Set SVC

OS9 F$ssvc

Function:

Adds or replaces a system call, which you have written, in OS-9's user and system mode system call tables.

Entry Conditions:

      Y   = *address of the system call initialization table*

Exit Conditions:

    None

    If error:
    CC   = C bit set
    B   = *error code*

Technical Function:

1.   Register Y passes the address of a table, which contains the function codes and offsets, to the corresponding system call handler routines. This table has the following format:

                  Use

Relative
Address

$00	Function Code	◄——— **First entry**
$01	Offset From Byte 3	
$02	To Function Handler	
$03	Function Code	◄——— **Second entry**
$04	Offset From Byte 6	
$05	To Function Handler	
	More Entries	◄——— **Third entry . . .**
	$80	◄——— **End-of-table mark**
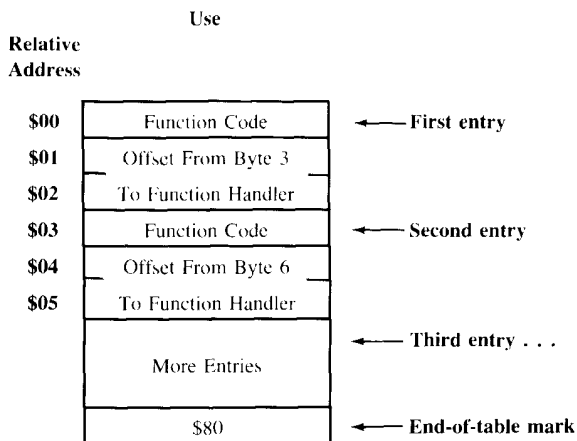
2a. If the most significant bit of the function code is set, the system table is updated.

2b. If the most significant bit of the function code is not set, the system and user tables are updated.

Function request codes are broken into these two categories:

$00 - $27   User mode system call codes.

$29 - $34   Privileged system mode system call codes. When these system calls are being installed, the most significant bit should be set if it is to be placed into the system table only.

These categories are defined by convention; they are not enforced by OS-9.

To use a privileged system call, you must be executing a program that has the type code $0C (OS-9 system module).

The system call handler routine should process the system call and return from subroutine with an RTS instruction. The handler routine may alter all CPU registers (except Register SP). Register U passes the address of the register stack to the system call handler as shown in the following diagram:

	Relative Address	Name
U → CC	$00	R$CC
	$01	R$D
A	$01	R$A
B	$02	R$B
DP	$03	R$DP
X	$04	R$X
Y	$06	R$Y
U	$08	R$U
PC	$0A	R$PC
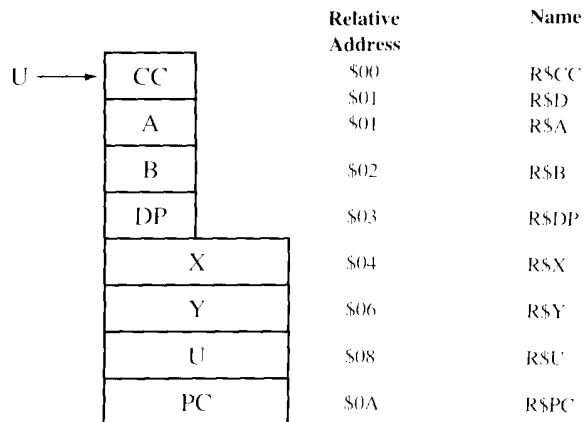
Codes $25 through $27 and $70 through $7F are user-definable.

Set SWI

OS9 F$sswi   103F 0E

Function:

Sets the interrupt vectors for SWI2 and SWI3 instructions.

Each process has its own local vectors. Each Set SWI call sets one type of vector according to the code number passed in Register A:

1 = SWI
2 = SWI2
3 = SWI3

When a process is created, all three vectors are initialized with the address of the OS-9 service call processor.

Warning: Microware-supplied software uses SWI2 to call OS-9. If you reset this vector these programs cannot work. If you change all three vectors, you cannot call OS-9 at all.

Entry Conditions:

A   = *SWI type code*
X   = *address of the user software interrupt routine*

Exit Conditions:
None

If error:
CC   = C bit set
B   = *error code*

128

Set Time

OS9 F$stime

Function:

Sets the current system date/time and starts the system real-time clock. The date and time are passed in a time packet as follows:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Entry Conditions:

    X   = *relative address of the time packet*

Exit Conditions:

The system time and date are set.

If error:
CC   = C bit set
B   = *error code*

Sleep

OS9 F$sleep

0A

Function:

Turns off the calling process temporarily.

If Register X contains 0, the process is turned off "indefinitely" (until it receives a signal). This is a good way to wait for a signal or interrupt without wasting CPU time.

If Register X contains 1, the process is turned off for the remainder of its current time slice. The process is inserted into the active process queue immediately, and resumes execution when it reaches the front of the queue. If the process receives a signal, it "awakens" before the time has elapsed.

If Register X contains an integer from 2 through 255, the process is turned off for the specified number of ticks, *n*. The process is inserted into the active process queue after *n-1* ticks. It resumes execution when it reaches the front of the queue. If the process receives a signal, it awakens before the time has elapsed.

Entry Conditions:

X   = *sleep time* (in ticks), or
X   = 0 (sleep indefinitely), or
X   = 1 (sleep for remainder of current time slice)

Exit Conditions:

X   = *sleep time minus the number of ticks that the process was asleep*

If error:
CC   = C bit set
B   = *error code*

Time

OS9 F$time

Function:

Returns the current system date and time in the form of a 6-byte packet (in binary). The packet is copied to the address passed in Register X.

The packet looks like this:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Entry Conditions:

X = *address of the area to store the time packet*

Exit Conditions:

time packet

If error:
CC = C bit set
B = *error code*

Unlink

Function:

Unlinks (destroys) a module, if it is not being used by another process.

Device driver modules that are being used and certain system modules cannot be unlinked. ROM modules can be unlinked; they cannot, however, be deleted from the module directory.

Entry Conditions:

U = *address of the module header*

Exit Conditions:

None

If error:
CC = C bit set
B = *error code*

Technical Function:

1. Unlink tells OS-9 that the module is no longer needed by the calling process.

2. OS-9 decrements the module's link count.

3a. If the resulting link count is zero, OS-9 destroys the module.

3b. If any other process is using the module, the module's link count cannot decrement to zero. Therefore, OS-9 does not destroy the module.

Wait

OS9 F$wait

Function:

Turns off the calling process until a child process "dies" by executing an Exit system call, or by receiving a signal. The Wait call helps you save system time.

The child's process ID and exit status are returned to the parent. If the child died because of a signal, the exit status byte (Register B) is the signal code.

If the caller has several children, the caller is activated when the first one dies. Therefore, one wait system call is required to detect termination of each child.

If a child died before the Wait call, the caller is reactivated almost immediately. If the caller has no children, Wait returns an error.

(See the Exit system call for more information.)

Entry Conditions:

None

Exit Conditions:

A = *deceased child process's ID*
B = *deceased child process's exit status code*

If error:
CC = C bit set
B = *error code*

Write

OS9 I$write

103F 8A

Function:

Writes to the file or device associated with the path number specified.

The path must have been opened or created in the write or update access mode. Data is written to the file or device without processing or editing. If data is written past the present end-of-file, the file is automatically expanded.

Entry Conditions:

X = *starting address of data to write*
Y = *number of bytes to write*
A = *path number*

Exit Conditions:

Y = *number of bytes written*

If error:
CC = C bit set
B = *error code*

134

Write Line

OS9 I$writln

Function:

Writes to the file or device associated with the path number specified.

Write Line is similar to Write except it writes data until a carriage return character is encountered. Line editing is also activated for character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.

The path must have been opened or created in the write or update access mode.

(For more information about line editing, see "SCFMAN Line Editing Functions" in Chapter 6.)

Entry Conditions:

X	= *starting address of the data to write*
Y	= *maximum number of bytes to write*
A	= *path number*

Exit Conditions:

Y	= *number of bytes written*

If error:

CC	= C bit set
B	= *error code*

Allocate 64

OS9 F$all64

Function:

Dynamically allocates 64-byte blocks of memory by splitting pages (256 bytes) into four sections.

OS-9 uses the first 64 bytes of the base page as a "page table." This table contains the MSB of all pages in the memory structure. If Register X passes a value of zero, the call allocates a new base page and the first 64-byte memory block.

Whenever a new page is needed, a Request Sysmem system call (F$srqmem) executes automatically. The first byte of each block contains the block number. Routines that use Allocate 64 call should not alter this byte.

The diagram below shows how seven blocks might be allocated:

	Any Memory Page	Any Memory Page
Base Page ⟶	Page Table (64 bytes)	X Block 4
	X Block 1 (64 bytes)	X Block 5 (64 bytes)
	X Block 2 (64 byte)	X Block 6 (64 byte)
	X Block 3 (64 byte)	X Block 7 (64 byte)
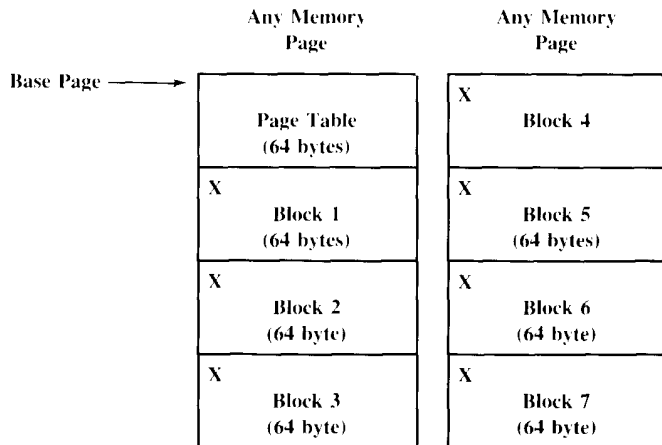
Allocate 64 is a privileged system mode call.

Entry Conditions:

X   = *base address of the page table; 0 = the page table has not been allocated*

Exit Conditions:

A   = *block number*
X   = *base address of the page table*
Y   = *address of the block*

If error:
CC   = *C bit set*
B   — *error code*

Find 64

OS9 F$find64

Function:

Returns the address of a 64-byte memory block.

OS-9 used Find 64 to find process descriptors and path descriptors when given their block number. The block number may be any positive integer.

Find 64 is a privileged system mode call.

Entry Conditions:

      X   = *address of the block*
      A   = *block number*

Exit Conditions:

      Y   = *address of the block*

      If error:
      CC   = C bit set
      B   = *error code*

I/O Delete

OS9 F$iodel

Function:

Deletes the specified I/O module from the system, if the module is not in use.

This system call is used mainly by IOMAN and may be of limited or no use for other applications.

I/O Delete is a privileged system mode call.

Entry Conditions:

X = *address of an I/O module*

Exit Conditions:
None

If error:
CC = C bit set
B = *error code*

Technical Function:

1. Register X passes the address of a device descriptor module, device driver module, or file manager module.

2. OS-9 searches the device table for the address.

3. If OS-9 finds the address, it checks the module's use count. If the count is zero, the module is not being used; OS-9 deletes it. If the count is not zero, the module is being used; OS-9 returns an error.

I/O Queue

OS9 F$ioqu

Function:

Inserts the calling process into the I/O queue of the specified process and puts the calling process to sleep indefinitely.

It is assumed that routines associated with the specified process will send a wakeup signal to the calling process.

I/O Queue is a privileged system mode call.

Entry Conditions:

A   = *process number*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Insert Process

OS9 F$aproc 103F 2C

Function:

Inserts a process into the active process queue so that the process may be scheduled for execution.

All processes already in the queue are sorted by process "age." Age is a count of how many process switches have occurred since the process's last time slice. When a process is moved to the active process queue, its age is set according to its priority — the higher the priority, the higher the age.

An exception is a newly active process that was deactivated while in the system state. Such a process is given higher priority because it usually is executing critical routines that affect shared system resources.

Insert Process is a privileged system mode call.

Entry Conditions:

     X   = *address of the process descriptor*

Exit Conditions:

    None

    If error:
    CC   = C bit set
    B   = *error code*

Next Process

Function:

Takes the next process out of the active process queue and executes it.

If there is no process in the queue. OS-9 waits for an interrupt, and then checks the queue again.

Next Process is a privileged system mode call.

Entry Conditions:

None

Exit Conditions:

Control does not return to caller.

Request Sysmem

OS9 F$srqmem 103F 28

Function:

Allocates a block of memory of the specified size from the top of available RAM. The size request is rounded to the next page boundary.

Request Sysmem is a privileged system mode call.

Entry Conditions:

D   = *byte count*

Exit Conditions:

U   = *starting address of the memory area*

If error:
CC   = C bit set
B   = *error code*

Return 64

OS9 F$ret64

Function:

Deallocates a 64-byte block of memory. (See the Allocate 64 system call for more information.)

Return 64 is a privileged system mode call.

Entry Conditions:

X   = *address of the base page*
A   = *block number*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

Return Sysmem

OS9 F$srtmem

Function:

Deallocates a block of contiguous pages.

Return Sysmem is a privileged system mode call.

Entry Conditions:

U   = *starting address of memory to return*; must point to an even page boundary

D   – *number of bytes to return*

Exit Conditions:

None

If error:
CC   = C bit set
B   = *error code*

SET IRQ

OS9 F$irq

Function:

Adds a device to or removes it from the IRQ polling table.

This system call is used mainly by device driver routines. (See "Interrupt Processing" in Chapter 2 for a complete discussion of the interrupt polling system.)

Packet Definitions:

Flip Byte   Selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).

Mask Byte   Selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.

Priority   The device priority number: 0 = lowest, 255 = highest

Set IRQ is a privileged system mode call.

Entry Conditions:

```
X  -   0 to remove a device, or
X  -   address of a packet to add a device
       [x]      = flip byte
       [x + 1]  = mask byte
       [X + 2]  = priority
U       = address of the service routine's memory area
Y       = address of the device IRQ service routine
D       = address of the device status register
```

Exit Conditions:

None

If error:
CC   = C bit set
B   = error code

Verify Module

Function:

Checks the module header parity and CRC bytes of a module.

If these values are valid, OS-9 searches the module directory for a module that has the same name. If one exists, OS-9 keeps the module that has the higher revision level.

Verify Module is a privileged system mode call.

Entry Conditions:

    X   = *address of the module to verify*

Exit Conditions:

    U   = *address of the module directory entry*

    If error:
    CC   = *C bit set*
    B   = *error code*

Appendix A / Alphabetical System Call Lists

User and I/O System Calls

Name	Call	Purpose	Code
Allocate Bits	F$allbit	Allocate in a bit map	13
Attach	I$attach	Attach a new device to the system	80
Chain	F$chain	Load and execute a new primary module	05
Change Directory	I$chgdir	Change the working directory	86
Close Path	I$close	Close a path to a file device	8F
Compare Names	F$cmpnam	Compare two names	11
CRC	F$crc	Compute the CRC	17
Create File	I$create	Create a path to a new file	83
Deallocate Bits	F$delbit	Deallocate in a bit map	14
Delete File	I$delete	Delete a file	87
Detach Device	I$detach	Remove a device from the system	81
Duplicate Path	I$dup	Duplicate a path	82
Exit	F$exit	Terminate the calling process	06
Fork	F$fork	Create a new process	03
Get ID	F$id	Get a process ID user ID	0C
Get Status	I$getstt	Get file device status	8D
Intercept	F$icpt	Set up a signal intercept trap	09
Link	F$link	Link to a memory module	00
Load	F$load	Load module(s) from a file	01
Make Directory	I$makdir	Make a new directory	85
Memory	F$mem	Set the memory size	07
Open Path	I$open	Open a path to a file/device	84
Parse Name	F$prsnam	Parse a path name	10
Print Error	F$perr	Print error message	0F
Read	I$read	Read data from a file device	89
Read Line	I$readln	Read a text line with editing	8B
Search Bits	F$schbit	Search the bit map for a free area	12

149

User and I/O System Calls

Name	Call	Purpose	Code
Seek	I$seek	Reposition the logical file pointer	88
Send	F$send	Send a signal to another process	08
Set Priority	F$sprior	Set the process priority	0D
Set Status	I$setstt	Set the file device status	8E
Set SVC	F$ssvc	Install a function request	32
Set SWI	F$sswi	Set an SWI vector	0E
Set Time	F$stime	Set the system date and time	16
Sleep	F$sleep	Put the calling process to sleep	0A
Time	F$time	Get the system date and time	15
Unlink	F$unlink	Unlink a module	02
Wait	F$wait	Wait for a child process to die	04
Write	I$write	Write to a file device	8A
Write Line	I$writln	Write a line of text with editing	8C

System Mode System Calls

Name	Call	Purpose	Code
Allocate 64	F$all64	Allocate a 64-byte memory	30
Find 64	F$find64	Find a 64-byte memory block	2F
I/O Delete	F$iodel	Delete an I/O device from the system	33
I/O Queue	F$ioqu	Insert a process into the I/O queue	2B
Insert Process	F$aproc	Insert a process into the active process queue	2C
Next Process	F$nproc	Start the next process	2D
Request Sysmem	F$srqmem	Request system memory	28

System Mode System Calls

Code Name	Call	Purpose	
Return 64	F$ret64	Deallocate (return) a 64-byte memory block	31
Return Sysmem	F$srtmem	Return system memory	29
Set IRQ	F$irq	Add or remove a device from the IRQ tables	2A
Verify Module	F$vmodul	Verify a module	2E

Appendix B / Numerical System Call Lists

User and I/O System Calls

| Code	Name	Call	Purpose
00	Link	F$link	Link to a memory module
01	Load	F$load	Load module(s) from a file
02	Unlink	F$unlink	Unlink a module
03	Fork	F$fork	Create a new process
04	Wait	F$wait	Wait for a child process to die
05	Chain	F$chain	Load and execute a new primary module
06	Exit	F$exit	Terminate the calling process
07	Memory	F$mem	Set the memory size
08	Send	F$send	Send a signal to another process
09	Intercept	F$icpt	Set up a signal intercept trap
0A	Sleep	F$sleep	Put the calling process to sleep
0C	Get ID	F$id	Get a process ID / user ID
0D	Set Priority	F$sprior	Set the process priority
0E	Set SWI	F$sswi	Set an SWI vector
0F	Print Error	F$perr	Print error message
10	Parse Name	F$prsnam	Parse a path name
11	Compare Names	F$cmpnam	Compare two names
12	Search Bits	F$schbit	Search the bit map for a free area
13	Allocate Bits	F$allbit	Allocate in a bit map
14	Deallocate Bits	F$delbit	Deallocate in a bit map
15	Time	F$time	Get the system date and time
16	Set Time	F$stime	Set the system date and time
17	Crc	F$crc	Compute the CRC
32	Set SVC	F$ssvc	Install a function request
80	Attach Device	I$attach	Attach a new device to the system
81	Detach Device	I$detach	Remove a device from the system

User and I/O System Calls

Code	Name	Call	Purpose
82	Duplicate Path	I$dup	Duplicate a path
83	Create File	I$create	Create a path to a new file
84	Open Path	I$open	Open a path to a file device
85	Make Directory	I$makdir	Make a new directory
86	Change Directory	I$chgdir	Change the working directory
87	Delete File	I$delete	Delete a file
88	Seek	I$seek	Reposition the logical file pointer
89	Read	I$read	Read from a file/device
8A	Write	I$write	Write to a file/device
8B	Read Line	I$readln	Read a text file with editing
8C	Write Line	I$writln	Write a line of text with editing
8D	Get Status	I$getstt	Get the file/device status
8E	Set Status	I$setstt	Set the file/device status
8F	Close Path	I$close	Close a path to a file/device

System Mode System Calls

| Code	Name	Call	Purpose
2A	Set IRQ	F$irq	Add or remove a device from the IRQ tables
2B	I/O Queue	F$ioqu	Insert a process into the I/O queue
2C	Insert Process	F$aproc	Insert a process into the active process queue
2D	Next Process	F$nproc	Start the next process
2E	Verify Module	F$vmodul	Verify a module
2F	Find 64	F$find64	Find a 64-byte memory block
28	Request Sysmem	F$srqmem	Request system memory
29	Return Sysmem	F$srtmem	Return system memory
30	Allocate 64	F$all64	Allocate a 64-byte memory block
31	Return 64	F$ret64	Deallocate (return) a 64-byte memory block
33	I/O Delete	F$iodel	Delete an I/O device from the system

Appendix C / Memory Module Diagrams

Executable Memory Module Format

Relative Address	Use	Check Range
$00	Sync Bytes ($87CD)	
$01		
$02	Module Size (bytes)	
$03		
$04	Module Name Offset	header parity
$05		
$06	Type / Language	
$07	Attributes / Revision	module CRC
$08	Header Parity Check	
$09	Execution Offset	
$0A		
$0B	Permanent Storage Size	
$0C		
$0D	(Additional optional header extensions located here) . . . . Module Body object code, constants, and so on	
	CRC Check Value	

157
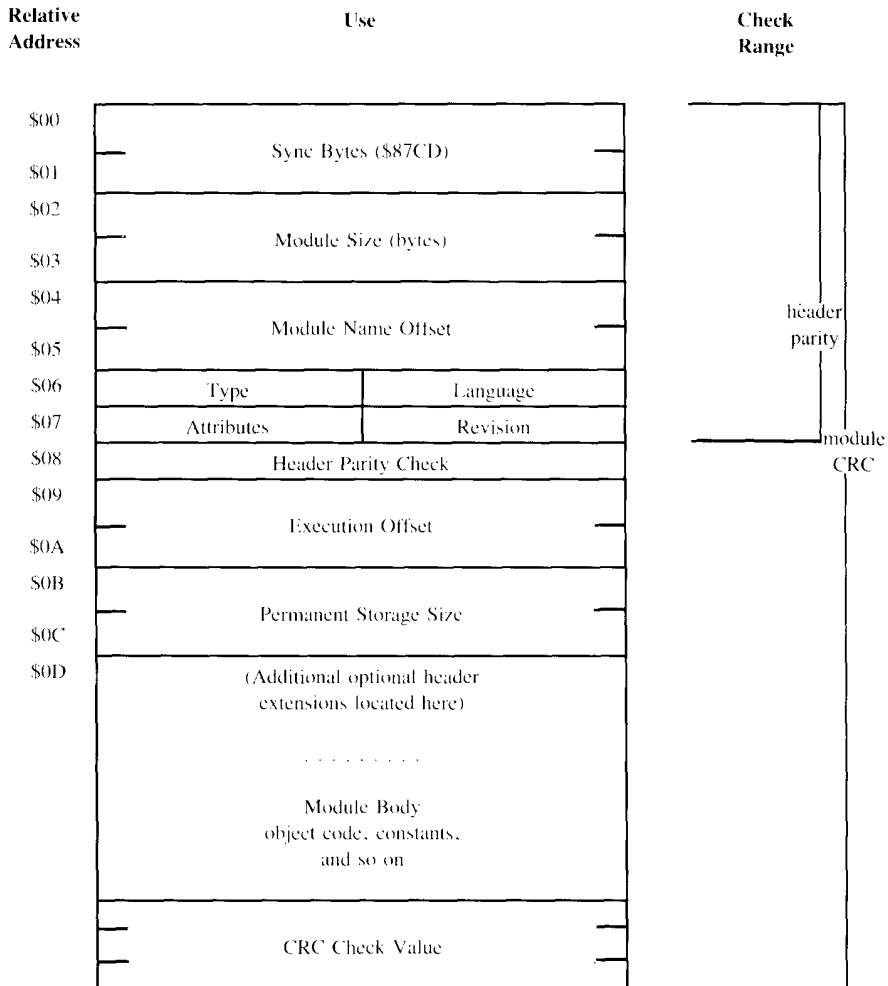
Device Descriptor Format

Relative Address	Use	Check Range

```
$00  ┌─────────────────────────────┐        ┐
     │─      Sync Bytes ($87CD)    ─│        │
$01  │                             │         │
$02  ├─────────────────────────────┤         │
     │─        Module Size        ─│         │
$03  │                             │         │
$04  ├─────────────────────────────┤    header│
     │─     Offset to Module Name ─│    parity │
$05  │                             │         │
$06  ├──────────────┬──────────────┤         │
     │   $F (Type)  │   $1 (Lang)  │         │
$07  ├──────────────┼──────────────┤         │
     │  Attributes  │   Revision   │        ┘
$08  ├─────────────────────────────┤
     │   Header Parity Check       │
$09  ├─────────────────────────────┤
     │─    Offset to File Manager ─│       module
$0A  │        Name String          │        CRC
$0B  ├─────────────────────────────┤
     │─    Offset to Device Driver─│
     │        Name String          │
$0D  ├─────────────────────────────┤
     │        Mode Byte            │
$0E  ├─────────────────────────────┤
     │─     Device Controller     ─│
$0F  │─   Absolute Physical Addr. ─│
     │           (24 bit)          │
$10  │                             │
$11  ├─────────────────────────────┤
     │   Initialization Table Size │
$12,$12+n ├──────────────────────────┤
     │     (Initialization Table)  │
     │                             │
     ├─────────────────────────────┤
     │   (Name Strings, and so on) │
     │                             │
     ├─────────────────────────────┤
     │      CRC Check Value        │
     └─────────────────────────────┘
```

INIT Module Format

Relative Address	Use	Check Range

Relative Address	Use	Check Range
$00	Sync Bytes ($87CD)	
$01		
$02	Module Size (bytes)	
$03		
$04	Module Name Offset	header parity
$05		
$06	$F (Type)	$1 (Lang)	
$07	Attributes	Revision	
$08	Header Parity Check	
$09	Forced Limit of Top	module CRC
$0A	of Free RAM	
$0B		
$0C	#IRQ Polling Table Entries	
$0D	#Device Table Entries	
$E 0	Offset to Startup	
$0F	Module Name String	
$10	Offset to Default Mass	
$11	Storage Device Name String	
$12	Offset to Bootstrap	
$13	Module Name String	
$14-n	Name Strings	
	CRC Check Value	

159

Appendix D / Standard Floppy Disk Format

TRS-80 Color Computer

Physical Track Format Pattern

Format	Bytes (Dec)	Value (Hex)
Header pattern	32	4E
(once per track)	12	00
	3	F5
	1	FC
	32	4E
Sector pattern	8	00
(repeated 18 times)	3	F5
	1	tracknumber (0-34)
	1	side number (0)
	1	sector number (1-18)
	1	sector length code (1)
	2	CRC
	22	4E
	12	00
	3	F5
	1	FB
	256	data area
	2	CRC
	24	4E
Trailer pattern	N	4E (fill to index mark)
(once per track)		

Appendix E / System Call Error Codes

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

Dec	Hex	Message	Meaning
200	$C8	PATH TABLE FULL	The file cannot be opened because the system path table is full.
201	$C9	ILLEGAL PATH NUMBER	The number is too large or is for a nonexistent path.
202	$CA	INTERRUPT POLLING TABLE FULL	You can't add an interrupt.
203	$CB	ILLEGAL MODE	You tried to perform an I/O function of which the device or file is incapable.
204	$CC	DEVICE TABLE FULL	You can't add a device.
205	$CD	ILLEGAL MODULE HEADER	The module is not loaded; its sync code, header parity, or CRC is incorrect.
206	$CE	MODULE DIRECTORY FULL	You can't add a module.
207	$CF	MEMORY FULL	There is not enough contiguous RAM free.
208	$D0	ILLEGAL SERVICE REQUEST	The system call contains an illegal code number.
209	$D1	MODULE BUSY	The module is non-sharable and is being used by another process.
210	$D2	BOUNDARY ERROR	The memory allocation or deallocation request is not on a page boundary.
211	$D3	END OF FILE	The end of the file was encountered on a read.
212	$D4	NOT YOUR MEMORY	You tried to deallocate memory that was not assigned.
213	$D5	NON-EXISTING SEGMENT	The device's file structure is damaged.
214	$D6	FILE NOT ACCESSIBLE	The file attributes do not permit the access requested.
215	$D7	BAD PATH NAME	The pathlist contained a syntax error — possibly an illegal character

Dec	Hex	Message	Meaning
216	$D8	FILE NOT FOUND	The specified pathlist does not exist.
217	$D9	SEGMENT LIST FULL	The file is too fragmented to be expanded
218	$DA	FILE ALREADY EXISTS	The specified filename already exists in the current directory.
219	$DB	ILLEGAL BLOCK ADDRESS	The device's file structure is damaged.
220	$DC	ILLEGAL BLOCK SIZE	The device's file structure is damaged.
221	$DD	MODULE NOT FOUND	You tried to link to a module that is not in the directory.
222	$DE	SECTOR OUT OF RANGE	The device's file structure is damaged or incorrectly formatted.
223	$DF	SUICIDE ATTEMPT	You tried to return the memory that contains your stack.
224	$E0	ILLEGAL PROCESS ID NUMBER	The specified process does not exist.
226	$E2	NO CHILDREN	The process can't do a Wait because it has no children.
227	$E3	ILLEGAL SWI CODE	The code must be from 1 to 3.
228	$E4	KEYBOARD ABORT	The process was aborted by Signal Code 2.
229	$E5	PROCESS TABLE FULL	The process can't do a fork now.
230	$E6	ILLEGAL PARAMETER AREA	The high and low bounds passed in Fork call are incorrect.
231	$E7	KNOWN MODULE	This module is for internal use only.
232	$E8	INCORRECT CRC	The module has a bad CRC value.
233	$E9	SIGNAL ERROR	The receiving process has a previous, unprocessed signal pending.
234	$EA	NONEXISTENT MODULE	The specified module does not exist.

Dec	Hex	Message	Meaning
235	$EB	BAD NAME	The name uses an illegal syntax.
236	$EC	BAD HEADER	The module header parity is incorrect.
237	$ED	RAM FULL	There is no free system RAM at this time.
238	$EE	BAD PROCESS ID	The process ID is incorrect.
239	$EF	NO TASK NUMBER AVAILABLE	All task numbers are in use.
240	$F0	UNIT ERROR	The device unit does not exist.
241	$F1	SECTOR ERROR	The sector number is out of range.
242	$F2	WRITE PROTECT	The device is write protected.
243	$F3	CRC ERROR	A CRC error occurred on a read or write verify.
244	$F4	READ ERROR	A data transfer error occurred during a disk read or an SCF (terminal) input buffer overrun.
245	$F5	WRITE ERROR	A hardware error occurred during a disk write.
246	$F6	NOT READY	The device has the "not ready" status.
247	$F7	SEEK ERROR	You attempted a physical seek to a nonexistent sector.
248	$F8	MEDIA FULL	The media does not contain enough free space.
249	$F9	WRONG TYPE	You tried to read an incompatible media, such as a double-sided disk on a single-sided drive.
250	$FA	DEVICE BUSY	A non-sharable device is in use.

Appendix F / Module and I/O Attributes

Standard I/O Paths

$00	-	Standard Input
$01	=	Standard Output
$02	=	Standard Error Output

Module Languages

$00	Data
$01	6809 Object Code
$02	BASIC 09 I-Code
$03	Pascal P-Code
$04	Cobol I-Code

File Access Codes

$01	=	Read
$02	=	Write
Read	+ Write =	Update
$04	=	Exec
$08	=	PRead
$10	=	PWrite
$20	=	PExec
$40	=	Share
$80	=	Dir

Module Types

$01	= Program Module
$02	Subroutine Module
$03	= Multi Module
$04	= Data Module
$0C	System Module
$0D	File Manager
$0E	Device Driver
$0F	Device Descriptor

Module Attributes

$8	= Reentrant

INDEX

INDEX

INDEX

E

Editing
    see Line Editing Functions 53-54
Error message 115, 163-165
    Writing (F$perr) 115
Exit (F$exit) 94

F

File 2, 8, 14, 26-27, 35-35, 88, 91, 99-102, 111-112,
    122-125, 135
    Creating and opening (I$create) 88
    Deleting (I$delete) 91
    Opening a path to (I$open) 11-112
    Pointer, repositioning (I$seek) 119
    Status, returning (I$gttstt) 99-102
    Status, setting (I$setstt) 122-125
    Terminating path to (I$close) 85
    Writing to (I$write) 134
    Writing to with editing (I$writln) 135
File access codes 111
File descriptor sector 35-36
File manager module 2, 26-27
    see also RBFMAN, SCFMAN, and PIPEMAN
Find 64 (F$find64) 138
Firq interrupt 14
Fork (F$fork) 10, 95
Free block, searching bit map for (F$schbit) 118
Free memory 8
Function calls 7, 75

G

Get ID (F$id) 98
Get Status (I$gttstt) 99-102

INDEX

INDEX

IRQ polling table 15-16
    Adding to or removing from (F$irq) 16, 146
    Flip byte 15
    Mask byte 15
    Polling address 15 Priority 16
    Service routine address 16
    Static storage address 16

K

Kernel 2, 5-16
    Functions of 5

L

Line editing functions 53-54
Link (F$link) 104-105
Load (F$load) 106-107
Logical interrupt polling system 15-16
Logical sector number 33
LSN 0 34
LSN 1 35

M

Make Directory (I$makdir) 108
Memory 110
    Allocating automatically 8
    Allocating block from top of available RAM
      (F$srqmem) 143
    Deallocating block of contiguous pages
      (F$srtmem) 145
    Deallocating 64-byte block (F$ret64) 144
    Dynamically allocating 64-byte blocks (F$all64)
      136-137
    Size, setting (F$mem) 110
Memory (F$mem) 110
Memory management, advantages of 7
Memory map 9
Memory module iii, 1
    see Module

INDEX

Module iii, 1
>   Attributes 17, 167
>   Deleting an I/O module (F$iodel) 139
>   Executable memory module format 22, 157
>   Format (parts) 18-19
>   Languages 20, 167
>   Linking to (F$link) 104-105
>   Loading and executing (F$chain) 82-83
>   Loading from a file (F$load) 106-107
>   Loading from the working executing directory
>   (F$load)
>   106-107
>   ROM 23-24
>   Types 17, 167
>   Unlinking (F$unlink) 132

Module body 19
Module CRC value 19
Module header 18
>   See Header
Multiprogramming 9-13

N

Name 86, 113-114
>   Analyzing a text string for (F$prsnam) 113-114
>   Comparing names (F$cmpnam) 86

Next Process (F$nproc) 142
NMI interrupt 15
Non-disk I/O (SCFMAN) 2, 27, 53-65

O

Open Path (I$open) 111-112
OS9Defs 75

P

Page 8
Parity byte, checking (F$vmodul) 147
Parse Name (F$prsnam) 113-114

INDEX

INDEX

INDEX

SWI interrupt 14
SWI2 interrupt 14
    Setting interrupt vectors for (F$sswi) 128
SWI3 interrupt 14
    Setting interrupt vectors for (F$sswi) 128
SYSGO 5
System call 6, 75
    Adding or replacing (F$ssvc) 126-127
    Lists 75, 149-155
    Processing 5, 6-7, 76-77
    Types 6-7
System initialization 5
System mode calls 76
System startup task (SYSGO) 5

T

Tick 10
Time 10, 19, 131
    Returning (F$time) 131
    Setting (F$stime) 129
    Slice 10

U

Unix operating system iii
Unlink (F$unlink) 132
User calls 149-150, 153-154
User ID 11
    Returning (F$id) 98

V

Vectors 14
    Hardware 14
    Setting for SWI2 and SWI3 instructions (F$sswi)
        14, 128
Verify Module (F$vmodul) 147

INDEX

W

26-3030
700-2331

OS-9 Addendum

Upgrade to Version 02.00.00

OS-9 Enhancements

OS-9 Enhancements

This addendum introduces the changes and new features of OS-9, Version 02.00.00. Some of the information is technical and is only of interest to programmers. However, other features of the new OS-9 Version 02.00.00 are of interest to all system users.

Contents

Section 1 / Commands and Utilities

CONFIG

CONFIG provides menus of all I/O options and all system commands. You select the device drivers and commands you want to include on a new system diskette from these menus. Selecting only the device drivers and commands you and your system require lets you make the most efficient use of computer memory and system diskette storage.

The CONFIG utility is on a separate CONFIG/BOOT Diskette. Make a copy of this diskette using the OS-9 BACKUP command and use the copy as your working diskette. Keep the original CONFIG/BOOT Diskette in a safe place to use for future backups. You can use the CONFIG/ BOOT Diskette for booting OS-9 from Color Disk BASIC from Drive /D0.

CONFIG requires no initial parameters. You establish parameters during the operation of CONFIG. Be sure that the execution directory is at /D0/CMDS before executing the command.

Examples:

CONFIG [ENTER]

CONFIG executes and a prompt asks you to indicate whether you wish to use one or two disk drives. Press [1] for single- or [2] for two-drive operation.

Next, CONFIG builds a list of the various devices from the MODULES directory. When the list is complete, a screen menu appears. Use the up and down arrow keys to move to a device. Then, press [S] to either select or exclude a particular device. Press [S] once to display an X to the right of the selected device. Press [S] again to erase the X. The device is selected only when "X" appears. Information about each device is available with a special *help* command. To display the information on the current device (the device indicated by the arrow (→), press [H].

If there are more than ten devices in a CONFIG menu, use $\boxed{\rightarrow}$ to move ahead page-by-page and $\boxed{\leftarrow}$ to move back.

The devices you can select are:

term 32	The computer keyboard and standard TV display
term 80	The computer keyboard and optional 80 column video display
d0	Disk Drive 0
d1	Disk Drive 1
d2	Disk Drive 2
d3	Disk Drive 3
h0-15	A 15 meg hard disk drive 0
h1-15	A 15 meg hard disk drive 1
h0-35	A 35 meg hard disk drive 0
h1-35	A 35 meg hard disk drive 1
p	A printer using the RS-232 serial port
t1	A terminal port using the standard RS-232 port
t2	A terminal port using the optional RS-232 communications pak
t3	A terminal port using the optional RS-232 communications pak
m1	A modem
m2	A modem
ssc	Speech/Sound Cartridge

To use your computer keyboard and video display, you must select one *term*. You must select *d0* as your first disk drive. Select *d1*, *d2*, and *d3* for additional floppy disk drives. Select *p* to use a printer with OS-9, select *ssc* to use a Speech/ Sound Cartridge from a Multi-Pak slot, and so forth.

After selecting the devices you desire, press $\boxed{D}$. The screen displays, ARE YOU SURE (Y/N) ? If you are satisfied with your selections, press $\boxed{Y}$. If you wish to make further changes, press $\boxed{N}$.

When the driver selection is complete, a screen prompt requests that you select among the Color Computer terminal I/O subroutines. Select these subroutines in the same manner that you selected the device drivers. You have the following modules from which to make your selections:

CO32	A video output module for a 32 column TV display
CO80	A video output module for a 80 column video display
GRFO	A graphics module for TV display

When choosing subroutine modules, you must select the video output module that matches the terminal module you previously selected for your console device.

CONFIG builds a boot list from the selected devices and their associated drivers and managers. "Bootlist" is created in the ROOT directory of Drive 0. CONFIG next displays two clock options:

1 - 60hz (American)
2 - 50hz (European)

If you live in the United States, Canada, or 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 to insert the SOURCE diskette, insert the CONFIG/BOOT Diskette. When asked to insert the DESTINATION diskette, insert the diskette on which you wish to create the new OS-9 System.

If you have more than one drive, a screen prompt asks you to insert a blank formatted diskette (the DESTINATION diskette) in /D1. The rest of the boot file creation is automatic.

Following the boot file generation, a menu lets you select the commands you wish to include on your system diskette. You have the following choices:

```
[N]o Commands, Stop Now
    -- Do not add any commands
[B]asic Command Set
    -- Add the basic OS-9 commands
[F]ull Command Set
    -- Add all OS-9 commands
[I]ndividually Select
    -- Select desired commands one by one
```

— Get help on the command set

Press N if you want to create a boot file, but do not wish to add any commands to the new system diskette. Use this option to create a new boot file on 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 CON-FIG utility to complete the entire system transfer, as less diskette swaps are required.

Press B if you wish to add a basic command set (the most commonly used commands) to your new diskette. This selection does not copy the following:

• Assembly language development tools, asm, debug, and edit and the DEFS directory
• Timesharing utilities, tsmon, login and the SYS/motd and SYS/password files
• The system maintenance utilities, such as dsave, dcheck, and cobbler.

Press F to copy all of the commands (an exact copy of the standard OS-9 system diskette, with a new boot file).

Press I to individually select commands to copy on the new diskette. This option displays a selection similar to the device selection screen. Again, press S to select or exclude commands, and use the arrow keys to move among the commands in the menu. Commands marked with an X are selected. If a command does not have an X beside it, it is excluded on the new system diskette.

If you have a multi-drive system, a prompt appears asking you to insert your OS-9 system diskette in /D0. Press the spacebar. 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, instead of the CONFIG/BOOT Diskette. The DESTINATION diskette is the new system

diskette you are creating. The number of swaps in this proce-
dure, as well as in the boot file creation procedure, depends
on the number of options you select.

It would be quicker and easier to use BACKUP to create a
system disk, use CONFIG to create a new boot file, then
delete unwanted commands. However, this process causes
fragmentation of diskette space. Fragmentation results in
slower diskette access, and free memory is broken into seg-
ments that might not be large enough for some OS-9 opera-
tions. CONFIG causes no fragmentation.

The MODULES directory of the CONFIG/BOOT diskette
contains all the device drivers and 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

FORMAT

FORMAT *devname* [R] [*diskname*]

Initializes, verifies, and establishes an initial file structure on a diskette. You must format all blank diskettes before you can use them on an OS-9 system.

Options:

devname is the drive name of the diskette you wish to format, D0 or D1.

R causes the format to proceed automatically, without displaying prompts.

diskname is the name you wish to assign to the newly formatted diskette. The diskname must be enclosed in double quotation marks.

Notes:

Be sure the diskette you want to format is *NOT* write protected. If a write-protect tab is in place, the system returns to the OS-9: prompt without formatting the diskette.

The formatting process works this way:

1. FORMAT physically initializes the diskette and organizes its surface into sectors.

2. FORMAT reads back and verifies each sector. If FORMAT fails to verify a sector after several attempts, it excludes that sector from the initial free space on the diskette. As the verification proceeds, track numbers appear on the screen.

3. FORMAT writes the diskette allocation map, ROOT directory, and identification sector to the first few sectors of track zero. These sectors must not be defective.

At the prompt, type a diskette volume name. You can use as many as 32 characters, including 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:

FORMAT /D1 [ENTER]

formats a diskette in Drive 1.

FORMAT /D1 R "test disk" [ENTER]

formats the disk in Drive 1, with the name *Test Disk*.

HELP

HELP *command name* [. . .]

Displays the usage and syntax of OS-9 commands.

Options:

command   is the command for which you want syntax
name   help. Include as many command names in one
             HELP line as you wish. The proper form and
             syntax appears for each valid command you
             include.

Notes:

To use HELP, first copy Cmds.hp from the SYS directory of
the CONFIG/BOOT diskette to the SYS directory of your
system diskette. Next, copy HELP from the CMDS directory
of the CONFIG/BOOT Diskette to the CMDS directory of
your system diskette as follows:

Procedure for one disk drive:

1. With OS-9 booted and the system diskette in your drive,
   type:

   ```
   LOAD COPY  [ENTER]
   ```

2. Replace the system diskette with the CONFIG/BOOT
   Diskette, and type:

   ```
   COPY /D0/SYS/CMDS.HP
        /D0/SYS/CMDS.HP  -S #30K  [ENTER]
   ```

3. Exchange the two diskettes as requested by the screen
   prompts until the process is complete.

4. Again, place the CONFIG/BOOT Diskette in the drive,
   and type:

   ```
   COPY /D0/CMDS/help /D0/CMDS/help
        -S #30K  [ENTER]
   ```

5. Swap diskettes as requested until the process is complete.

8

Procedure for two disk drives:

1. With OS-9 booted, place the CONFIG/BOOT Diskette in Drive 1. Be sure the system diskette is in Drive 0.

2. Type:

```
COPY /D1/SYS/CMDS.HP
     /D0/SYS/CMDS.HP [ENTER]
```

3. When the first copy is complete, type:

```
COPY /D1/CMDS/help
     /D0/CMDS/help [ENTER]
```

Cmds.hp is a data file, not a text file, and you cannot successfully display it on your screen or edit it with a standard text editor. It contains help for standard OS-9 commands.

HELP displays the form and syntax of the specified command. If you use a non-standard command name, a screen display tells you that help is not available for that command.

Examples:

```
HELP BACKUP [ENTER]
BACKUP [e][s][-v][dev][dev]
```

Copies all data from one device to another

```
HELP ME [ENTER]
ME Help not available
```

```
HELP [ENTER]
HELP [command name][...]
```

INIZ

INIZ *devicename* [. . .]

Links 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. If the device is already installed, INIZ does not reinitialize it.

Options:

devicename   is the name of the device driver you want to initialize. Specify as many device drivers as you wish with one INIZ command.

Notes:

You can use INIZ in the startup file or at the system startup to initialize devices and allocate their static storage at the top of memory (to reduce memory fragmentation).

Example:

```
INIZ P T2  [ENTER]
```

initializes the P (printer) and T2 (terminal 2) devices.

OS9GEN

OS9GEN *devname* [-s] [#*n*K]

Creates and links the required OS9Boot file on any diskette from which you wish to boot OS-9.

Options:

devicename	is the disk drive containing the diskette on which you wish the new boot file to be created.
-s	causes OS9GEN to perform a single-drive boot file generation operation. In a single-drive operation, OS9GEN reads the modules from the source diskette and requests that you exchange diskettes and press C as the modules are read and copied.
#*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, lessen the number of diskette swaps.

Notes:

OS9GEN adds modules to an existing boot file, or creates a new one. (If you want an exact copy of the existing OS9Boot, you can also use COBBLER).

OS9GEN creates a working file called tempboot on the device specified by *devname*. Next, it reads filenames (pathnames) from its standard input (the keyboard), one pathname per line. OS9GEN opens each file and copies it to tempboot. The process repeats until it reaches an end-of-file marker or a blank line. All boot files must contain the OS-9 component modules listed in *Section 5.1* of the *OS-9 Commands* manual.

With all input files copied to tempboot, OS9GEN deletes the OS9Boot file, if it exists. Tempboot is renamed OS9Boot,

and its starting address and size are written 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 02.00.00 kernel, an error message displays and the operation terminates.

An OS9Boot file must be in physically contiguous sectors. Therefore, you normally use OS9GEN on a newly formatted diskette. If the OS9Boot file is fragmented, OS9GEN prints a warning message indicating that you cannot use the diskette to bootstrap OS-9.

You can either enter a list of filenames from the keyboard for OS9GEN or you can direct input to OS9GEN from a file containing a list of filenames. If you enter names manually, no prompts are given. Press [ENTER] after typing each filename. After typing the last filename and [ENTER], press [ENTER] again, or press [CTRL] [BREAK] to complete the list.

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

Examples:

```
OS9GEN /D1 [ENTER]
/D0/OS9Boot [ENTER]
[CTRL][BREAK]
```

Manually installs 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]
/D0/tape.driver [ENTER]
/D2/video.driver [ENTER]
[CTRL][BREAK]
```

Manually installs a boot file on device /D1 that is a copy of the OS9Boot file on device /D0 plus the modules stored in

the files /D0/tape.driver and /D2/video.driver. The first command line runs 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.

```
BUILD /D0/bootlist [ENTER]
? /D0/OS9Boot [ENTER]
? /D0/CMDS/DIR [ENTER]
? /D0/CMDS/COPY [ENTER]
? [ENTER]
OS9GEN /D1</D0/bootlist [ENTER]
```

Generates a new boot file on Drive 1 that includes all the old boot file functions plus loads DIR and COPY into memory on startup. Line 1 uses BUILD to create a file called bootlist. The next three lines enter the names of the three files within bootlist. Line 5 terminates BUILD, and Line 6 runs OS9GEN with input redirected from the new bootlist file.

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

```
BUILD /D0/Bootlist [ENTER]
? IOMAN [ENTER]
? RBF [ENTER]
? CCDisk [ENTER]
? D0 [ENTER]
? SCF [ENTER]
? CCIO [ENTER]
? TERM [ENTER]
? PRINTER [ENTER]
? P [ENTER]
? CLOCK [ENTER]
? SHELL [ENTER]
? SYSGO [ENTER]
? [ENTER]
```

This procedure creates a bootlist on Drive 0 in preparation for a single-drive OS9GEN operation. You must locate all of the files ·in the current data directory by SAVEing the modules from memory into the directory. 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 buffer space, and the filenames previously stored in the bootlist file.

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

TMODE

TMODE [*.pathnum*] [*paramlist*] [. . .]

Displays or changes the operating parameter's of the terminal.

Options:

upc	Displays uppercase characters only. Lowercase characters automatically convert to uppercase.
-upc	Displays both upper- and lowercase characters.
bsb	Causes 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 *backspaces 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*. Lines are *deleted* by printing a *new line* sequence (for hard-copy terminals).
echo	Input characters *echo* back to the terminal. This setting is the system default.
-echo	Turns off the echo default.
lf	Turns on the auto line feed. Line feeds automatically echo to the terminal on input and output carriage returns. This setting is the system default.
-lf	Turns off the auto line feed default.

15

pause	Turns on the screen pause. This setting suspends output when the screen is full. See the *pag* parameter for definition of screen size. Resume output by pressing the space bar.
-pause	Turns off the screen pause mode.
null = *n*	Sets the null count, sets the number of null ($00) characters transmitted after carriage returns for return delay. The number is in decimals. Default = 0.
pag = *n*	Sets video display page length to *n* (decimal) lines. This setting is used for *pause* mode.
bsp = *h*	Sets the input backspace character and requires a hexadecimal value. Default = 08.
bse = *h*	Sets the output backspace character and requires a hexadecimal value. Default = 08.
del = *h*	Sets input delete line character and requires a hexadecimal value. Default = 18.
bell = *h*	Sets the bell (alert) output character and requires a hexadecimal value. Default = 07.
eor = *h*	Sets the end-of-record (carriage return) input character. Requires a value in hexadecimal. Default = 0D.
eof = *h*	Sets the end-of-file input character and requires a hexadecimal value. Default = 1B.

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

> 0 = no parity
> 101 = MARK parity transmitted, no checking
> 111 = SPACE parity transmitted, no checking
>
> 011 = even parity   only available with the
> 001 = odd parity   external ACIA pak and Modpak devices.

Bits 0-3 are reserved for future use.

Bit 4 is used to select auto-answer modem support features.

> 1 = on
> 0 = off
> See *Color Computer I/O Devices* for further information.

For TERM, the type byte has a different use:

> Bit 0 specifies a machine with true lowercase capability. Set bit 0 to turn on true lowercase.
>
> Bit 1 specifies output screen size. Clear bit 1 for 32 column output to the video screen. Set bit 1 for 80 column output to the video screen through the 80 column card.

reprint = *h*	Sets the reprint line character and requires a hexadecimal value.

dup = *h*	Sets the character to duplicate the last input line and requires a hexadecimal value.
psc = *h*	Sets the pause character. The numeric value of the character is in hexadecimal.
abort = *h*	Sets the terminate character (normally CONTROL C). The numeric value of the character is in hexadecimal.
quit = *h*	Sets the quit character (normally CONTROL E). The numeric value of the character is in hexadecimal.
baud = *h*	Sets baud, word length, and stop bits rate for software-controllable interface. The numeric codes for baud rate are: $0 = 6E$, $1 = 12C$, $2 = 258$, $3 = 4B0$, $4 = 960$, $5 = 12C0$, $6 = 2580$, $7 = 4B00$ in hexadecimal.

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.

Notes:

You can specify any number of parameters from the options list, separating them by spaces or commas. If you don't specify parameters, the output is determined by the current TMODE status.

You can also use a period and a number to specify the output pathnumber. If you don't specify any, TMODE affects the standard input path.

This utility adjusts its output for either an 80 or 32 column display.

If this command is used in a SHELL procedure file to change the terminal's operating characteristics, you must use the parameter *.pathnum* to specify one of the standard output paths (1 = standard output, and 2 = standard error). The change remains in effect until the path closes. To make a permanent change to a device characteristic, change the device descriptor.

TMODE works only if a path to the file/device is open. You can alter the device descriptor to set a device's initial operating parameter using XMODE (see the *OS-9 Commands* manual).

TMODE can also alter the baud, word length, stop bits, and parity for devices already initialized.

Examples:

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

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

Note: If you use TMODE in a procedure file, it is necessary to specify one of the standard output paths (.1 or .2), because the SHELL's standard input path is redirected to the diskette file. (You can use TMODE only on SCFMAN-type devices.)

Example:

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

This sets lines per page for standard output.

TUNEPORT

TUNEPORT </P or /T1> [-s = value]

Lets you test and set delay loop values for the current baud
rate and select the best value for your printer (/P) or terminal
(/T1).

Examples:

    TUNEPORT /P [ENTER]

Provides a test operation for your printer. After a short delay,
TUNEPORT displays the current baud rate and sends data to
the printer to test 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, test
data is sent 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 /T1
terminal.

    TUNEPORT /P -s = 225 [ENTER]

Sets the delay loop value for your printer at 255. 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 TUNEPORT in your
system startup file to set the value using the -S option.

XMODE

XMODE *devname* [*paramlist*]

Displays or changes the initialization parameters of any SCF-type device such as the video display, printer, RS-232 port, and others.

XMODE is commonly used to change baud rates and control key definitions.

Options:

upc	Sets the screen display to uppercase only. Lowercase characters automatically convert to uppercase.
-upc	Sets the screen display to upper- and lowercase.
bsb	Causes a backspace to erase characters. A backspace character echoes as a backspace-space-backspace sequence.
-bsb	Turns off *erase on backspace*. Only a single backspace echoes.
bsl	Enables *backspace over line* function. Lines are deleted by sending backspace-space-backspace sequences. This option is for video terminals and is the default option.
-bsl	Disables *backspace over line*. Lines are deleted by producing a *new line* sequence (this option is for hard-copy terminals).
echo	Causes input characters to echo to the terminal. This is the default option.
-echo	Disables the echo function.

lf	Turns on the auto line feed. Line feeds automatically echo to the terminal on input and output carriage returns. This is the default option.
-lf	Turns off the auto line feed.
pause	Turns on the screen pause mode. This setting suspends output when the screen is full. See *pag* parameter for definition of screen size. Resume output by pressing any key.
-pause	Turns off the screen pause mode
null = n	Sets the null count, the number of null characters ($00) transmitted after a carriage return. The number is a decimal value. Default = 0.
pag = n	Sets video display page length to n (decimal) lines. This setting is used for *pause* mode.
bsp = h	Sets the input backspace character and requires a hexadecimal value. Default = 08.
bse = h	Sets the output backspace character and requires a hexadecimal value. Default = 08.
del = h	Sets the input delete line character and requires a hexadecimal value. Default = 18.
bell = h	Sets the bell (alert) output character and requires a hexadecimal value. Default = 07.
eor = h	Sets the end-of-record (carriage return) input character. Requires a value in hexadecimal. Default = 0D.
eof = h	Sets the end-of-file input character. The numeric value the character is hexadecimal. Default = 1B.

22

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

$$0 = \text{no parity}$$
$$101 = \text{MARK parity transmitted, no checking}$$
$$111 = \text{SPACE parity transmitted, no checking}$$

$011 =$ even parity	only available with the ex-	
$001 =$ odd parity	ternal ACIA pak and Modpak devices.	

Bits 0-3 are reserved for future use.

Bit 4 selects auto-answer modem support features.
$$1 = \text{on}$$
$$0 = \text{off}$$
See *Color Computer I/O Devices* for further information.

For TERM, the type byte has a different use:
Bit 0 specifies a machine with true lowercase capability. Set bit 0 to turn on true lowercase.

Bit 1 specifies output screen size. Clear bit 1 for 32 column output to the video screen. Set bit 1 for 80 column output to the video screen through the 80 column card.

reprint = *h*   Sets the reprint line character. Use a hexadecimal numeric value.

dup = *h*   Sets character to duplicate last input line character and requires a hexadecimal value.

psc = *h*   Sets pause character and requires a hexa-decimal value.

abort = *h*   Sets the terminate character (normally CONTROL C) and requires a hexadecimal value.

quit = *h*   Sets the quit character (normally CONTROL E) and requires a hexadecimal value.

baud = *h*   Sets baud, word length, and stop bits rate for software-controllable interface. The numeric codes for baud rate are: $0 = 6E$ $1 = 12C$, $2 = 258$, $3 = 4B0$, $4 = 960$, $5 = 12C0$, $6 = 2580$, $7 = 4B00$ value is in hexadecimal.

> 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.

Notes:

XMODE is similar to TMODE, but there are differences. TMODE operates only on open paths, so its effect is tempo-rary. XMODE updates the device descriptor; its change per-sists 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 parameter(s) and the COB-BLER program to make a new system diskette or re-write system tracks on the current system diskette, the changed parameter is permanently on the new system diskette.

XMODE requires that you specify a device name. If you do not specify parameters, the present values for each parameter display. You can use any number of parameters separating them by spaces or commas.

Examples:

```
XMODE /TERM -upc lf null=4 bse=1F
   pause [ENTER]

XMODE /T1 pag=24 pause bsl -echo
   bsp=8 [ENTER]

XMODE /P baud=3 -lf [ENTER]
```

Other Utilities

Several other OS-9 utilities are altered to adjust their screen size to either a 32 or 80 column display. This is done using the new SS.ScSiz GETSTAT call (see the GETSTAT call section of this addendum for full information.)

The utilities altered in this manner are:

CONFIG	DIR	DUMP	LOGIN
MDIR	PROCS	TMODE	XMODE

Section 2 — System Changes

Beep To create a short sound from your terminal speaker, send a CTRL G or character 7 to the terminal. You can cause a beep from BASIC with the command PRINT CHR$(7). To cause a beep from OS-9, type DISPLAY 07 [ENTER].

Error Codes System Error $DC (220 decimal) is defined as a *HANG UP* error and indicates that the RS232 data carrier detect is lost. It replaces the *ILLEGAL BLOCK SIZE* error on page 164 of the *OS-9 Technical Information* manual.

CCIO The Color Computer Input/Output module is now divided into three separate sections. The graphics portion of the module does not load on startup. To load the graphics portion, type LOAD GRFO at the OS-9 prompt. Because graphics draw routines are not pre-loaded, more memory is available for programs that do not require graphics capabilities. See Appendix B for more information on graphics.

Video Output Separate subroutine modules, CO32 and CO80 handle video output. CO32 provides a standard 32 column TV display and is included in the standard boot. CO80 provides an 80 column video display when used in conjunction with an optional 80 column card.

Hard Disks A driver is available for one or two 15 or 35 meg hard disk units. Use the CONFIG utility to prepare your system for hard disk capability. See *Appendix E* for more information on hard disks.

Clock Interrupt A system change now permits clock interrupt accessibility to device drivers. See the VIRQ system call in Section 3.

Drive Descriptors The descriptors for Disk Drives 2 and 3 are removed from the OS-9 boot file. The descriptors are not resident in memory when the system is initialized. If you wish to use more than two disk drives, replace the descriptors in the boot file using OS9GEN, CONFIG, or COBBLER.

Graphics Display Buffer A "Setstat" function now allows for allocation and deallocation of additional graphics buffers. See the SETSTAT system call in the next section.

Graphics Driver Enhancements Several new graphics functions are added to the graphics module. See the information in the *Appendix B Update*. This is the updated version of the *OS-9 Commands* manual Appendix B, *Display System Functions*.

Speech/Sound There is now a speech and sound driver added for the Speech/Sound Cartridge. Select the Speech/Sound (*ssc*) driver through the CONFIG utility (See CONFIG in Section 1).

Keyboard Driver Enhancements The keyboard driver now has an auto key-repeat and the capability of generating new codes and status values. The following new features are added through the GETSTAT system call:

● GETSTAT now returns the current key press status for the following keys:

<pre>
    [ALT] ([@])
    [SHIFT]
    [CTRL] ([CLEAR])
    [↑]
    [↓]
    [←]
    [→]
    the spacebar
</pre>

See GETSTAT in *System Calls* for more information.

● [@] serves as an ALT key to set the high order bits on characters typed from the keyboard. For instance, pressing [A] generates a code of 65 (producing a character A), pressing [ALT][A] generates a code of 193 (producing a graphics character)

● The auto key-repeat feature causes a key to repeat when you press it for longer than one second.

SCF Descriptor Tables

Add the following information to the path descriptor table *SCFMAN* *Option Section Definition* (PD.OPT Section), beginning on Page 55 of the *OS-9 Technical Information* manual. PD.XON replaces the previous PD.STN entry.

Name	Relative Address	Size (Bytes)	Use
PD.XON	$38	1	ACIA XON char
PD.XOFF	$39	1	ACIA XOFF char

PD.XON   contains either the character to enable transmission of characters or a null to disable XON.

PD.XOFF   contains either the character to disable transmission of characters or a null to disable XOFF.

PD.PAR   specifies parity information for external serial devices. See the explanation of the TMODE type byte in this addendum for more information.

PD.BAU   specifies baud rate, word length, and stop bit information for serial devices. See the explanation of the TMODE baud byte in this addendum for more information.

SCF-Device Memory Area Definitions

Add the following information to the *SCF-Device Memory Area Definitions* table, beginning on page 59 of the *OS-9 Technical Information* manual. (V.XON replaces the previous entry, V.SCF in the table.) See the OS9DEFS directory on your system diskette for additional information.

Name	Relative Address	Size (Bytes)	Use
V.XON	$0F	1	XON character
V.XOFF	$10	1	XOFF character
V.KANJI	$11	1	Reserved
V.KBUF	$12	2	Reserved
V.MODAPR	$14	2	Reserved
V.PDLHD	$16	2	Header of path descriptor list
V.RSV	$18	5	Reserved
V.SCF equ	$1D		End of SCF memory requirements

Add the following information to the *SCF-Type Device Descriptor Modules* table on Page 57 of the *OS-9 Technical Information* manual:

Name	Relative Address	Size (Bytes)	Use
IT.XON	$2A	1	X-ON character
IT.XOFF	$2B	1	X-OFF character
IT.COL	$2C	1	Number of screen columns
IT.ROW	$2D	1	Number of screen rows

Note: IT.XON replaces the previous IT.STN entry in the device descriptor.

Section 3 — System Calls

GETSTAT

OS-9 I$GETSTT   **103F 8D**

Function:

Gets the status of a device. This call is a "wild card" call used to handle individual device parameters that:

- are not uniform on all devices
- are highly hardware dependent
- need to be user-changeable

Following are the currently defined function codes for GETSTAT:

Mnemonic	Code	Function
SS.Opt	$00	Reads the 32 byte options section of the path descriptor
SS.Ready	$01	Tests for data ready (RBF, ACIA)
SS.Size	$02	Returns the current file size (RBF)
SS.Pos	$05	Gets the current file position (RBF)
SS.Eof	$06	Test for the end of file (RBF, ACIA)
SS.DevNm	$0E	Returns the device name (IOMAN)
SS.DStat	$12	Returns graphics display information
SS.Joy	$13	Returns joystick information
SS.AlfaS	$1C	Returns alpha screen information
SS.Cursr	$25	Returns cursor information
SS.ScSiz	$26	Returns screen size information
SS.KySns	$27	Returns the key down information packet
SS.ComSt	$28	Returns parity, stop bit, word length, baud rate information for SCF devices

Following are the parameter passing conventions for the GETSTATS function codes.

SS.Opt (Code $00) Reads the option section of the path descriptor and copies it into the 32-byte area pointed to by Register X. Use this code to determine the current settings for editing functions such as echo and auto line feed. For a complete description of the status packet, see the section on path descriptors.

Entry Conditions:
      A   = path number
      B   = $00
      X   = address of place to put a 32-byte status packet

Exit Conditions:
      Status packet

Error:
      CC = C bit set
      B   = error code

SS.Ready (Code $01) Tests for data available on SCF-supported devices.

Entry Conditions:
      A   = path number
      B   = $01

Exit Conditions:
      If ready:
      CC = C bit clear
      B   = number of bytes available, or 0 if the byte count
            is not supported by the driver.

      If not ready:
      CC = C bit set
      B   = $F6   (E$NotRDY)

Error:
      CC = C bit set
      B   = error code

SS.Size (Code $02) Gets the current file size (RBF-supported devices only).

Entry Conditions:
    A = path number
    B = $02

Exit Conditions:
    X = ms 16 bits of the current file size
    U = ls 16 bits of current file size

Error:
    CC = C bit set
    B = error code

SS.Pos (Code $05): Gets the current file positions (RBF-supported devices only).

Entry Conditions:
    A = path number
    B = $05

Exit Conditions:
    X = ms 16 bits of the current file position
    U = ls 16 bits of the current file position

Error:
    CC = C bit set
    B = error code

SS.EOF (Code $06): Tests for end of file.

Entry Conditions:
    A = path number
    B = $06

Exit Conditions:

Not EOF	C bit set	Error
C bit clear	C bit set	C bit set
zero	$D3 (E$EOF)	Error codes

Error:
CC = C bit set
B = appropriate error code

SS.DevNm (Function Code $0E): Returns the device name.

Entry Conditions:
A = path number
B = $0E
X = address of 32-byte area for device name

Exit Conditions:
Device name in 32-byte storage area

SS.DStat (Code $12): Returns the display status.

Entry Conditions:
A = path number
B = $12

Exit Conditions:
X = address of the graphics display memory
Y = graphics cursor address; x = msb, y = lsb
A = color code of the pixel at the cursor address

SS.Joy (Code $13): Returns the joystick values.

Entry Conditions:
A = path number
B = $13
X = 0 (right joystick) or 1 (left joystick)

Exit Conditions:
X = selected joystick x value (0-63)
Y = selected joystick y value (0-63)
A = $FF (if fire button is on) or $00 (if fire button is off)

SS. AlfaS (Code $1C): Returns information about alpha screen memory.

Entry Conditions:
A = path number
B = $1C

Exit Conditions:
           A   = caps lock status
                  $00 = lowercase
                  $FF = uppercase
           X   = address of buffer in memory
           Y   = address of cursor in memory

SS.Cursr (Code $25):   Returns cursor information from alpha screen.

Entry Conditions:
           A   = pathnumber
           B   = $25

Exit Conditions:
           A   = character under cursor
           X   = x position of cursor (column)
           Y   = y position of cursor (row)

SS.ScSiz (Code $26):   Returns size of screen (default from descriptor).

Entry Conditions
           A   = pathnumber
           B   = $26

Exit Conditions:
           X   = number of columns on screen
           Y   = number of rows on screen

           Note: These are the bytes IT.COL and IT.ROW follow-
           ing the IT.XOFF byte in the device descriptor. See
           SCF-Type Device Descriptor Modules in the *OS-9 Tech-
           nical Information* manual for full information.

SS.KySns (Code $27):   Returns key down information packet.

Entry Conditions:
           A   = path number
           B   = $27

Exit Conditions:

A = byte with bits set to indicate which key was down at the last keyboard scan.

∅ = bit not set and key not down
1 = bit set and key down

Bit   Key
∅   = Shift Key
1   = Control/Clear Key
2   = Alt/@ key
3   = Up arrow key
4   = Down arrow key
5   = Left arrow key
6   = Right arrow key
7   = Space key

The following information can be included as equates for the different bit positions:

SHIFTBIT   equ %00000001
CNTRLBIT   equ %00000010
ALTERBIT   equ %00000100
UPBIT   equ %00001000
DOWNBIT   equ %00010000
LEFTBIT   equ %00100000
RIGHTBIT   equ %01000000
SPACEBIT   equ %10000000

SS.ComSt (Code $28): Return information on parity, stop bits, word length, and baud rate for SCF devices.

Entry Conditions:
A = path number
B = $28

Exit Conditions:
Y = msb parity byte of path descriptor (parity)
    lsb baud rate of path descriptor (baud, word length, stop bits)

High order byte of Y:
    Bits 0-4 - Reserved
    Bits 5-7 - Parity code
        000 = no parity
        001 = odd parity transmitted and received
        011 = even parity transmitted and received
        101 = mark parity transmitted, no receiver parity checked
        111 = space parity transmitted, no receiver parity checked

    Note: Reserved bits are used by some Color Computer drivers.

Low order byte of Y:
    Bits 0-3 - Baud rate code
        0 = 110
        1 = 300
        2 = 600
        3 = 1200
        4 = 2400
        5 = 4800
        6 = 9600
        7 = 19200

    Bit 4   - Reserved
    Bits 5-6 - Word length
        00 = 8 bits
        01 = 7 bits
    Bit 7   - Stop Bits
        0 = 1 stop bit
        1 = 2 stop bits

SCF uses this call to determine what the current values for PD.PAR and PD.BAU should be. You can make this call, but doing so does not update the path descriptor. Use SS.Opt to maintain the path descriptor in a valid state.

SETSTAT

EOS-9 I$SETSTT

Function:

Sets the status of a device. This call is a "wild card" call used to handle individual device parameters that:

- are not uniform on all devices
- are highly hardware dependent
- need to be user-changeable

Following are the presently defined function codes for GETSTAT:

Mnemonic	Code	Function
SS.Opt	$00	Writes the 32 byte options section of the path descriptor
SS.Size	$02	Sets the file size (RBF)
SS.Reset	$03	Restores head to track zero
SS.WTrk	$04	Writes (format) track
SS.Feed	$09	Issues Form Feed (SCF)
SS.FRZ	$0A	Freezes DD. information
SS.SPT	$0B	Sets sectors per track
SS.SQD	$0C	Disk drive sequence down
SS.DCMD	$0D	Directs command to disk collector
SS.KySns	$27	Enables/Disables keyboard sense ability
SS.ComSt	$28	Sets baud, parity, stop bit, word length information for SCF devices.
SS.AAGBf	$80	Allocates additional graphics buffers
SS.SLGBf	$81	Select graphics buffer

SS.Opt (Code $00):   Writes the option section of the path descriptor from the 32-byte status packet pointed to by the X register. It is typically used to set the device operating parameters, such as echo and auto line feed.

Entry Conditions:
      A   = path number
      B   = $00
      X   = address of a 32-byte status packet

Exit Conditions: none

SS.Size (Code $02):   Set file size for RBF-type devices.

Entry Conditions:
      A   = path number
      B   = $02
      X   = ms 16 bits of desired file size
      U   = ls 16 bits of desired file size

Exit Conditions: none

SS.Reset (Code $03):   Restores disk drive head to track zero.

Entry Conditions:
      A   = path number
      B   = $03

Exit Conditions: none

SS.Wtrk (Code $04):   Formats a floppy diskette track. The entire diskette is formatted when the track number equals zero.

Entry Conditions:
      A   = path number
      B   = $04
      X   = address of track buffer
      U   = track number

Y = side/density
   Bit B0 = side (0 = side zero, 1 = side one)
   Bit B1 = density (0 = single, 1 = double)
   (The high order byte contains the number of sides)

Exit Conditions: none

SS.FRZ (Code $0A):   Inhibits the reading of the identification sector (LSN 0) to memory so non-standard disks can be read.

Entry Conditions:
   A = path number
   B = $0A

Exit Conditions: none

SS.SPT (Code $0B):   Sets a different number of sectors per track so non-standard disks can be read.

Entry Conditions:
   A = path number
   B = $0B
   X = new sectors per track

Exit Conditions: none

SS.SQD (Code $0C):   Sequence down—initiates power-down sequence for Winchester or other hard disks which have sequence-down requirements prior to the removal of power.

Entry Conditions:
   A = path number
   B = $0C

Exit Conditions: none

SS.KySns (Code $27):   Sets or resets special key sense function according to the contents of X.

Entry Conditions:
   A = path number

B   = $27
X   = Keysns switch value
    0 = disable key sense and return to normal key operation
    1 = enable key sense mode

Exit Conditions: none

Error:
CC = C set on error
B   = Error code if one exists

With the keyboard in the key sense operation mode, the keys in the following list do not send characters to SCF devices or to the screen when pressed. The only method of determining a keypress is through the GETSTAT call. Take care to re-enable the standard mode before reading commands from the terminal. The KeySns GETSTAT call returns the proper bits set even if the keys are all generating output.

Key Sense Code:

Bit	Key
0	SHIFT
1	CTRL CLEAR
2	ALT / @
3	↑
4	↓
5	←
6	→
7	The spacebar

The following is suggested equates for the bit positions:

SHIFTBIT	equ	%00000001
CNTRLBIT	equ	%00000010
ALTERBIT	equ	%00000100
UPBIT	equ	%00001000
DOWNBIT	equ	%00010000
LEFTBIT	equ	%00100000
RIGHTBIT	equ	%01000000
SPACEBIT	equ	%10000000

SS.ComSt (Code $28): Sets parity, baud, word length, and stop bit information for SCF-type devices.

Entry Conditions:
    A   = path number
    B   = $28

Exit Conditions:
    Y   = msb parity bite
            lsb baud byte (baud, word length and stop bits)

SSCF devices currently use this Setstat to inform a driver that the parity or baud bytes are changed in the path descriptor. If you make this Setstat call, the path descriptor is not updated until you make the next SS.Opt Getstat call. Then, the path descriptor is updated with the new information.

SS.HngUp (Code $2B): Flushes buffers and shuts down modem port on phone hang up.

Entry Conditions:
    A   = path number
    B   = $2B

Exit Conditions: none

You can use this call with true ACIA drivers (T2, T3, M1, M2).

SS.AAGBf (Code $80): Allocates additional graphics buffer—sets additional 6K graphic buffers. Select the first buffer with the standard DISPLAY graphics command (writing control code 15 to the standard terminal driver). Request additional buffers (as many as three) by using this Setstat. Buffers are numbered 0-2 where 0 is the buffer allocated by the DISPLAY graphics command and 1 and 2 are buffers allocated using the SS.AAGBf Setstat. The additional buffers affected by this routine remain allocated until the *end graphics* command is received.

Entry Conditions:
    A   = path number
    B   = $80

Exit Conditions:
        X   = address of buffer
        Y   = buffer number (0, 1, 2)

Error:
        CC = C bit set
        B   = error code

SS.SLGBf (Code $81):   Selects graphics buffer as the current draw and display buffer or the current display buffer. You can select buffers as foreground buffers without displaying them. The SS.Dstat call returns the address of the current foreground buffer and not necessarily the displayed buffer.

Entry Conditions:
        A   = path number
        B   = $81
        X   = mode
                    0 = select buffer as current use buffer
                non 0 = select buffer as current use buffer and
                        cause the buffer to be displayed
                        beginning at the next clock interrupt.
        Y   = buffer number (0, 1, 2)

You can select an additionally allocated buffer and draw in it before you display it. Use SS.SLGBF to cause the foreground screen to display at the next clock interrupt.

Exit Conditions:
        X   = mode (unchanged)
        Y   = buffer number (0, 1, 2) (unchanged)

Error:
        CC = C bit set
        B   = error code

VIRQ

Function

Sets the number of clock ticks at which a device can be interrupted. VIRQ is useful with devices in the Multi-Pak expansion slots that can generate physical interrupts on the processor but which cannot be recognized by the processor because of the lack of an IRQ line from the Multi-Pak. It is also useful for devices without physical interrupts. VIRQ can be used to simulate such a physical interrupt.

Entry Conditions
      Y   = address of 5-byte packet
      X   = 0 to delete entry, 1 to install entry
      D   = initial count value

Exit Conditions: none

The VIRQ polling table is handled by the CLOCK module and is dependent on the clock being initialized. SYSGO forces the clock to start in OS-9 versions 02.00.00 and later.

The virtual interrupt is set so that a device can interrupt at a given number of clock ticks. The interrupt can occur once, or can be repeated as long as the device is used. The VIRQ call requires a five byte packet specified for use in its table. This packet contains:

- Bytes for an actual counter
- A reset value for the counter
- A status byte that indicates whether a Virtual Interrupt has occurred and whether the VIRQ is to be reinstalled in the table after being issued
- A tick count that is used as an initial count for the interrupt

44

The five-byte packet is defined as follows:

Name	Offset	Function
Vi.Cnt	$00	Actual counter
Vi.Rst	$02	Reset value for counter
Vi.Stat	$04	Status byte

Two of the bits in the status byte are used, these are:
Bit 0 - Set if VIRQ occurs
Bit 7 - Set if count is to be reset

When making a F$VIRQ call, initialize the packet with a reset value, if required. Also, you must either set or clear bit 7 of the status byte to signify a reset of the counter or a one time VIRQ call. The reset value does not need to be the same as the initial counter value. When the call is processed, the address of the packet is installed on the VIRQ table, where it can be accessed by the clock module.

At each clock tick the table is *parsed* (scanned) and the timers are decreased by one. When the count becomes zero, the following actions occur:

- Bit 0 is set in the status byte, specifying a Virtual IRQ

- Bit 7 of the status byte is checked to see whether the count is to be reset

- If bit 7 is set, the count is reset using the reset value, otherwise the entry is deleted from the table.

When a Virtual IRQ results from a counter dropping to zero, the standard polling routine is executed, and the interrupt is serviced. Thus, you must install entries on both the VIRQ and IRQ polling tables to use the VIRQ.

Install the device on the IRQ polling table using the F$IRQ system call before you place it on the VIRQ table, unless the device has an actual physical interrupt that can be found on a hardware register. If the device has such an interrupt, use that as the polling address for the F$IRQ system call, along with the proper flip and mask bytes for the device. If the device has no interrupts, and is totally VIRQ driven, use the status byte from the VIRQ packet as the status byte, and use

a mask byte of %00000001 defined as VI.IFlag in the defs file. Use a flip byte of 0.

F$VIRQ is a privileged system call.

Appendix B Update

Display System Functions

Color Computer OS-9 lets you use the video display in alphanumeric, semigraphic, and graphic modes. There are many built-in functions to control the display. You activate these functions using ASCII control characters. They are available to any software written in a language using standard output statements (such as PRINT in BASIC). Color Computer BASICO9 has a Graphics Interface Module that can automatically generate most of these codes using BASICO9 RUN statements.

The display system has two display modes: Alphanumeric (*Alpha*) and Graphics. The Alphanumeric mode also includes *semigraphic* box-graphics. The Color Computer's display system uses a separate memory area for each display mode so operations on the Alpha display do not affect the Graphics display, and vice-versa. Either display can be selected under software control. (See *Getting Started With Extended Color BASIC* for more detailed information.)

Eight-bit characters sent to the display system are interpreted according to their numerical value, as shown in this chart:

Character Range (Hex)	Mode/Used For
00 - 0E	Alpha Mode—Cursor and screen control.
0F - 1D	Graphics Mode—Drawing and screen control.
1E - 1F	Escape loading for 80 column display.
20 - 5F	Alpha Mode—Upper case characters.
60 - 7F	Alpha Mode—Lower case characters.
80 - FF	Alpha Mode—Semigraphic patterns.

The Color Computer OS-9 alphanumeric functions are handled by the OS-9 device driver module called CCIO, and one of several output routines. The basic graphics functions are all handled by CCIO using the GRFO subroutine procedure for processing.

Alpha Mode Display

This is the standard operational mode. Use it to display alphanumeric characters and semigraphic box graphics, and to simulate the operation of a typical computer terminal with functions for scrolling, cursor positioning, clear screen, line delete, and so on.

This mode assumes that each 8-bit character is an ASCII character. This character displays if its high order bit (sign bit) is cleared. If the high order bit of the character is set, the character is assumed to be a *Semigraphic 6* graphics box. See *Getting Started With Extended Color BASIC* for an explanation of semigraphics functions.

The standard Alpha mode display is handled by the I/O subroutine module CO32. CCIO calls this module (included in the standard boot file) to process all text and semigraphic output.

If you have the optional 80-Column Cartridge and a monitor capable of 80 columns, you can use the CO80 subroutine module to give an 80 column text display. If the CO80 subroutine module is not found in memory it loads automatically. This subroutine is not included in the standard boot file, but is in the CMDS directory of the CONFIG/BOOT diskette. You can included the subroutine in a boot file using the CONFIG utility.

Both text display subroutine modules can be present in memory to allow alternate access to both the 32 column television output and the 80 column monitor output. Switch to 80 column mode to use the 80 column output. Switch off the 80 column mode to again use 32 column output. The mode switch is handled by the par byte of the path descriptor (described in the section of this addendum dealing with TMODE). To turn on 80 column text use TMODE type = 02. To turn it off, use TMODE type = 00. The following Alpha mode command codes are exactly the same for the 80 column display and for the 32 column display.

Alpha Mode Command Codes

Hex Control Code	Decimal Control Code	Name/Function
$01	01	**HOME**—returns the cursor to upper left corner of screen.
$02	02	**CURSOR XY**—moves the cursor to character X or line Y. Use the binary values minus 32 of the two characters following the control character as the X and Y coordinates. For example, to position the cursor at character 5 of line 10, use X = 37 and Y = 42.
$03	03	**ERASE LINE**—erases all characters on the cursor line.
$04	04	**CLEAR TO END OF LINE**—erases all characters from the current cursor position to the end of the line.
$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
$20	32	space	Cursor OFF
$21	33	!	Cursor ON . . Default color (blue)
$22	34	"	Cursor ON . . Black
$23	35	#	Cursor ON . . Green
$24	36	$	Cursor ON . . Yellow
$25	37	%	Cursor ON . . Blue
$26	38	&	Cursor ON . . Red
$27	39	'	Cursor ON . . Buff
$28	40	(	Cursor ON . . Cyan
$29	41	)	Cursor ON . . Magenta
$2A	42	*	Cursor ON . . Orange

Hex Control Code	Decimal Control Code	Name/Function
$06	06	**CURSOR RIGHT**—moves the cursor to the right one character position.
$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 home cursor.
$0D	13	**RETURN**—returns the cursor to left-most character of line.
$0E	14	**DISPLAY ALPHA**—switches the screen from graphic mode to alphanumeric mode.

Command Codes for 80-Column Card

$1F $20	31 32	**TURN OFF ATTRIBUTES**—turns off reverse video for 80-column display.
$1F $21	31 33	**TURN ON ATTRIBUTES**—turns on reverse video for 80-column display.

Graphics Mode Display

Use the graphics mode to display high-resolution two- or four-color graphics. This 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 causes the graphics screen to display and sets a current 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 continuous free memory available. (You can use MFREE to check free memory.) This memory is retained until you give the *end graphics* command, even if the program that initiated Graphics mode finishes. It's important to use the *end graphics* command to give up the display memory when you no longer need Graphics mode.

Graphics mode supports two basic formats. The two-color format has 256 horizontal by 192 vertical points (G6R mode). The four-color format has 128-horizontal by 192 vertical points (G6C mode). Two color sets are available in either mode. 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 the GRFO subroutine loaded into memory. This module is contained in the CMDS directory of the standard OS-9 system diskette. Load it with the command LOAD GRFO. You can also install the GRFO module as part of the boot operation by creating a new system diskette using the CONFIG utility described in this addendum.

Graphics Mode Selection Codes

Code	Format
00	256 x 192 two-color graphics
01	128 x 192 four-color graphics

COLOR SET AND CURRENT FOREGROUND COLOR SELECTION CODES

		Two-Color Format		Four-Color Format	
	Char	Back-ground	Fore-ground	Back-ground	Fore-ground
Color Set 1	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 1	08			Black	Black
	09			Black	Dark Green
	10			Black	Med. Green
	11			Black	Light Green
Color Set 1	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, a 6K byte display buffer is assigned. If 6K of contiguous memory isn't available, an error is returned. Follow this command with two characters specifying the graphics mode and current color/color set, respectively. This is part of CCIO and does not require GRFO.
$10	16	**PRESET SCREEN**—presets the entire screen to the color code passed by the next character. This function is processed by CCIO and does not require GRFO.
$11	17	**SET COLOR**—selects foreground color (and color set) passed by the next character, but does not change the graphics mode. This function is processed by CCIO and does not require GRFO.
$12	18	**END GRAPHICS**—disables the graphics mode and returns the 6K byte graphics memory area to OS-9 for other use. It also switches to alpha mode. This function is processed by CCIO and does not require GRFO.

Hex Control Code	Decimal Control Code	Name/Function
$13	19	**ERASE GRAPHICS**—erases all points to the background color and locates the graphics cursor at the desired position. This function is processed by CCIO and does not require GRFO.
$14	20	**HOME GRAPHICS CURSOR**—moves the graphics cursor to coordinates 0,0 (lower left-hand corner). This function is processed by CCIO and does not require GRFO.
$15	21	**SET GRAPHICS CURSOR**—moves the graphics cursor to the given X, Y coordinates. The binary value of the two characters that immediately follow are used as the X and Y values, respectively.
$16	22	**DRAW LINE**—draws a line of the current foreground color from the current graphics cursor position to the given X,Y coordinates. The binary value of the two characters that immediately follow are used as the X and Y values, respectively. The graphics cursor moves to the end of the line.
$17	23	**ERASE LINE**—operates the same as the *draw line* function, except that the line is *drawn* in the current background color, thus erasing the line.

Hex Control Code	Decimal Control Code	Name/Function
$18	24	**SET POINT**—sets the pixel at point X,Y to the current foreground color. The binary values of the two characters that immediately follow are used as the X and Y values, respectively. The graphics cursor moves to the point set.
$19	25	**ERASE POINT**—operates the same as the *draw point* function, except the point is *drawn* in the current background color, thus erasing the point.
$1A	26	**DRAW CIRCLE**—draws a circle of the current foreground color around the current graphics cursor (as the center point) using the radius obtained from the binary value of the next character.
$1C	28	**ERASE CIRCLE**—operates the same as the *draw circle* function, except that the circle is drawn in the current background color, thus erasing the circle.
$1D	29	**FLOOD FILL**—*paints* the current 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 fill is called for the first time it requests allocation for a 512 byte stack for the fill routine. This memory is not returned until you terminate graphics with the *end graphics* command.

Note: Hexadecimal codes are for compatibility with the OS-9 Display command.

Get Status Commands

The Color Computer I/O driver includes OS-9 *get status* commands that return information about the keyboard, display, and joysticks. All are accessed using the assembly language system calls listed. You can access the first two calls using the BASICO9 Graphics Interface Module.

Get Display Status:

Calling Format:	lda #1	(path number)
	ldb #SS.DStat	(Getstat code $12)
	os9 I$GSTT	(call OS-9)

Passed: nothing

Returns: X = address of graphics display memory
Y = graphics cursor address ⌐
A = color code of pixel at cursor address

Note: This command is available through the BASICO9 Graphics Interface Module.

Get Joystick Values:

Calling Format:	lda #1	(path number)
	ldb $SS.Joy	(Getstat code $13)
	os9 I$GSTT	(call OS-9)

Passed: X = 0 for right joystick; 1 for left joystick

Returns: X = selected joystick x value (0-63)
Y = selected joystick y value (0-63)
A = $FF if fire button on; $00 if off

Note: This command is available through the BASICO9 Graphics Interface Module.

Get Alpha Display Status:

Calling Format:	ldb #1	(path number)
	ldb #SS.AlfaS	(Getstat code $1C)
	os9 I$GSTT	(call OS-9)
Passed:	nothing	
Returns:	A = caps lock status	
	$00 = lowercase	
	$FF = uppercase	
	X = address of buffer in memory	
	Y = address of cursor in memory	

Get Alpha Cursor Status:

Calling Format:	lda #1	(path number)
	lda #SS.Cursr	(Getstat code $25)
	os9 I$GSTT	(call OS-9)
Passed:	nothing	
Returns:	A = character under cursor	
	X = x position of cursor (column)	
	Y = y position of cursor (row)	

Get Screen Size:

Calling Format:	lda #1	(path number)
	ldb #SS.ScSiz	(Getstat code $26)
	os9 ISGSTT	(call OS-9)
Passed:	nothing	
Returns:	X = number of columns on screen	
	Y = number of rows on screen	

Get Key Sense Information:

Calling Format:	lda #1	(path number)
	ldb #SS.KySns	(Getstat code $27)
	os9 I$GSTT	(call OS-9)

Passed:	nothing
Returns:	A = status byte with bits set indicating a key was depressed on last keyboard scan.

0 = bit not set, key NOT down

1 = bit set, key down

Bit   Key
0 = shift key
1 = control/Clear key
2 = alt/(α key
3 = up arrow key
4 = down arrow key
5 = left arrow key
6 = right arrow key
7 = spacebar

Note: To enable this command, you must enable the Key Sense mode using the Enable/Disable Key Sense set status command.

Set Status Commands

The Color Computer I/O driver includes OS-9 *set status* commands that lets you alter the keyboard or display status.

Enable/Disable Key Sense Ability:

Calling Format:	lda #1	(path number)
	ldb #SS.KySns	(Setstat code $27)
	ldx #KySnsVal	(0 to disable, 1 to enable)
	os9 I$SETSTT	(call OS-9)

Passed:	x = 0 to disable key sense mode
	1 to enable key sense mode
Returns:	nothing

Allocate Additional Graphics Buffers

Calling Format:	lda #1	(path number)
	ldb #SS.AAGBf	(SetStat code $80)
	os9 I$SETSTT	(call OS-9)

Passed:	nothing
Returns:	X = address of buffer
	Y = buffer number (0, 1, 2)

Select Graphics Buffer

Calling Format:	lda #1	(path number)
	ldb #SS.SLGBf	(SetStat code $81)
	ldx #Mode	(get mode)
	ldy #BUFNUM	(get buffer number)
	os9 I$SETSTT	(call OS-9)

Passed: X = mode:

    0 = select buffer as current use buffer

    > 0 = select current buffer and cause it to be displayed beginning at next clock interrupt.

Y = buffer number (0, 1 or 2)

Returns: X = mode, unchanged

Y = buffer number, unchanged

Display Control Codes Condensed Summary

1st Byte Dec	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
04	04			Erase to End of line
05	05	Cursor Code		Alter Cursor
06	06			Cursor right
08	08			Cursor left
09	09			Cursor up
10	0A			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

Appendix D Update

Keyboard Control Functions

Key Definitions for Special Functions and Characters

KEY COMBINATION	CONTROL FUNCTION OR CHARACTER
`@` or `ALT`	Alternate key—sets the high order bit on a character. Press `@`*CHAR* or `ALT`*CHAR*. If your keyboard does not have `ALT` use `@`.
`CTRL` or `CLEAR`	Used as a control key `CTRL`. Some Color Computers use `CLEAR`.
`BREAK` or `CTRL` `E`	Stops the program currently executing.
`CTRL` `—`	Generates an underline (__) character. The underline character displays as a left arrow (`←`).
`CTRL` `,`	Generates a left brace ({) character. The left brace character displays as a left bracket ([) in reverse video.
`CTRL` `.`	Generates a right brace (}) character. The right brace character displays as a right bracket (]) in reverse video.
`CTRL` `#`	Generates a tilde (˜) character. The tilde displays as a hyphen (-) in reverse video.
`CTRL` `/`	Generates a reverse slash (\) character.
`CTRL` `BREAK`	Generates an end-of-file (EOF). This sequence is the same as `ESC` on a standard terminal.

Key	Description
SHIFT @	Generates the (a character.
← 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/lower case shift lock function.
CTRL 1	Generates a vertical bar (|) character. The vertical bar character displays as an exclamation point (!) in reverse video.
CTRL 7	Generates an up arrow or caret (^) character.
CTRL 8	Generates a left bracket ([) character.
CTRL 9	Generates a right bracket (]) character.
CTRL A	Repeats the previous command line.
CTRL D	Redisplays the current command line on the video display.
CTRL W	Temporarily halts output to the screen display. Press any key to resume output.
CLEAR ?	Generates a backslash (/) character.

Appendix E Update

Hard Disk Support

OS-9 Version 02.00.00 supports the use of 15 or 35 meg
hard disks through a device driver called CCHDisk.
CCHDisk lets you use one or more hard disk drives and the
optional Hard Disk Controller Cartridge in Slot 3 of the
Multi-Pak Interface. The descriptors for the two types of hard
disks are contained within the CONFIG utility as:

h0__15	Drive 0 15 meg hard disk
h1__15	Drive 1 15 meg hard disk
h0__35	Drive 0 35 meg hard disk
h1__35	Drive 1 35 meg hard disk

When the SYSGO module starts the system, it attempts to
change to the hard disk, if one is attached. If you have a
hard disk and the appropriate driver and descriptors, your
execution directory is /H0/CMDS and your data directory is
/H0.

The device descriptor for the hard disk is different than for
floppy diskettes. The following charts provide the data for the
Color Computer Hard Disk descriptors and disk drive units:

Descriptor for Color Computer Hard Disk	
IT.DTP	= 1 (RBF)
IT.DRV	= drive number 0 - 3
IT.STP	= step rate, see table
IT.TYP	= %100000000 Hard disk flag for bit 7
IT.DNS	= unused
IT.CYL	= number of cylinders (see chart)
IT.SID	= number of heads (see chart)
IT.VFY	= write verify bit 0 = on : >0 = off

Descriptor for Color Computer Hard Disk		
IT.sct	= sectors/track	(see chart)
IT.TOS	= sectors/track 0	(see chart)
IT.ILV	= sector interleave	
IT.SAS	= segment allocation size (CoCo = 1)	
IT.TFM	= not used, DMA only	
IT.Exten	= not used, level II only	
ITSToff	= not used, level II only	

Step Rates for WD1010 Controller	
Code	Rate
0	35 us
1	0.5 ms
2	1.0 ms
3	1.5 ms
4	2.0 ms
5	2.5 ms
6	3.0 ms
7	3.5 ms
8	4.0 ms
9	4.5 ms
10	5.0 ms
11	5.5 ms
12	6.0 ms
13	6.5 ms
14	7.0 ms
15	7.5 ms

Cylinder/Head chart for TANDY Hard Disks					
Disk	Heads	Cylinders	TPI	SPT	Pre-Comp
15 meg	6	306	345	32	128
35 meg	8	512	512	34	256

Six bytes are added to the end of the hard disk descriptor that are not copied to the path descriptor. They are used at initialization to set up the parameters for the specific drive type. Therefore, if you have more than one hard disk drive, they must be of the same type as the first drive. These bytes start at offset $25 in the device descriptor and are described in the following chart.

Offset	Function
$25	Write Pre-Comp cylinder number /4 (see chart)
$26	The number of digits in the cylinder address, DIVA value for the driver
$27	Normalized divisor for the division routine, DIVY value
$29	DIVU value, MSB = mask for quotient LSB = shift value for remainder
$2B	Format gap value

Most values are used for the division routines to convert LSN's from RBF into a cylinder number, sector number, and head number.

Color Computer I/O Devices

Several new input/output devices are available for the Color Computer operating under OS-9. The OS-9 system diskette contains drivers for the standard devices and the CONFIG/BOOT diskette contains drivers and descriptor modules for other devices. This section provides information about the devices available using OS-9.

SCF-Type Devices

SCF-type devices are those that operate on character oriented data and deal with only one character at a time. Full information on the SCF is found in the *Sequential Character File Manager* chapter in the *OS-9 Technical Information* manual. The SCF-type devices supported include:

TERM—for the keyboard video display
P—for a serial printer
T1, T2, and T3—for external terminals
M1 and M2—for external modems
SSC—for a voice synthesizer

Parity and Baud Rates

Under OS-9 Version 02.00.00 and higher, external serial devices P, T1, T2, T3, M1, and M2 have the capability of changing their baud rates, parity, stop bits, and word length parameters any time they are in use. Such changes are handled by SCF, through the SS.ComSt SETSTAT. Two bytes in the device descriptor hold information about parity and baud rates for these devices. These bytes can initialize similar bytes in the path descriptor.

A driver can then use these bytes to change its operating parameters through the SETSTAT call. TMODE and XMODE give you access to these bytes for examination or change. The two bytes are used as follows:

Baud Byte—supplies information on baud rate, word length and stop bits
IT.BAU—for device descriptor
PD.BAU—for path descriptor

Bits 0-3   Baud rate code:
              0 = 110
              1 = 300
              2 = 600
              3 = 1200
              4 = 2400
              5 = 4800
              6 = 9600
              7 = 19.2K

             Note: on some devices, all values might not be obtainable or alterable.

Bit 4   Reserved

Bits 5-6   Word length code:
             00 = 8 bit
             01 = 7 bit

Bit 7   Stop bit code:
             00 = 1 stop bit
             01 = 2 stop bits

Parity Byte—supplies information on parity.
        IT.TYP—for the device descriptor
        PD.PAR—for the path descriptor

Bits 0-3   Reserved

Bit 4   Modem support feature (T1, T2, T3, M1, M2 only)

Bits 5-7   Parity code:
             XX0 = no parity (X = 1 or 0)
             101 = MARK parity transmitted, no parity check
             111 = SPACE parity transmitted, no parity check

The true ASCIA devices (T2, T3, M1, and M2) also support these additional parity codes:

001 = ODD parity transmitted and received
011 = EVEN parity transmitted and received

Reserved bytes can be used differently by different devices. They are noted in their individual sections.

TERM

Uses .... Keyboard/video display
Modules Required ..... TERM
CCIO
Additional I/O subroutine modules

Features include:

Lowercase and 80 column   TERM does not use the baud byte and has a different use for the parity byte. It does not handle data communications through a specific port, but directly reads the keyboard and writes to the video display:

Bit 0 Is used as a flag for true lowercase on compatible machines
Bit 1 Is used to distinguish display size. 0 = 32 columns, 1 = 80 columns through an external 80 column card.

Display Screen

SCF devices now have full support for varying display screen sizes. The device descriptor has two bytes for this purpose, IT.COL and IT.ROW. These two bytes contain information for the number of columns and rows available for display. You can use an OS-9 GetStat call to access this information. The standard display screen format is 32 columns. An optional text display is available for 80 column text using a monitor and an 80 column cartridge in a Multi-Pak Expansion Interface. The CO80 I/O subroutine module handles the text output for this card. Many of the OS-9 utilities can now adapt to either 32 or 80 column output.

While the 32 column display module is standard, and included in the standard OS-9 system boot, the modules for both 32 and 80 column displays are available through the CONFIG utility. If you want to use a monitor and an 80 column display with the 80 column cartridge, create a boot that uses the TERM80 and CO80 modules with CONFIG. Should you wish, you can use CONFIG to restore the 32 column standard display.

Graphics

Basic graphics routines that do not require drawing on the graphics screen are handled directly by CCIO. All drawing routines pass to GRFO for processing.

The graphics module, GRFO, resides in the CMDS directory of your system diskette. You can load GRFO into memory using the LOAD command, or you can include it in the system boot file using CONFIG. GRFO is needed only if you wish to use the built-in OS-9 draw commands for lines, points, circles and fills. For more information on the alpha and graphics modes, see Appendix B of this addendum.

Following is a table summarizing the I/O subroutine modules supported:

Name	Function
CO32	32 column text and semi-graphics output to a television display
CO80	80 column text output to a monitor using an optional 80 column cartridge
GRFO	Graphics output to television display

Keyboard Character Generation

The Color Computer keyboard can generate all ASCII characters as well as semigraphic characters using [ALT] or [@]. The CCIO also features key repeat and type-ahead capabilities. Type-ahead is only functional when disk drives are not being accessed. Other keyboard features include a terminal bell (Character 07) which sounds through the television speaker, and support for true lowercase capability on Color Computers having this feature.

RS-232 External Terminal

Device descriptor T1 provides serial communication capability through the RS-232 serial port at the rear of the Color Computer. You can use this port to support terminal speeds as high as 300 baud. Data transfer rates above 300 baud are

unreliable. If data transmission proves unreliable at 300 baud, use the TUNEPORT utility to adjust the T1 driver for the capabilities of your computer.

T1 supports changeable baud rates, parity (no parity, MARK, and SPACE), word length, and stop bits. The default settings are:

- 300 baud
- no parity
- 8 data bits
- 1 stop bit

Use XMODE or TMODE utilities to configure the port of other parameters.

T1 is set for 80 column displays. Utilities that adapt to different display sizes, display in 80 columns to an attached terminal.

The T1 and RS232 modules are on the standard OS-9 system diskette.

T2

Use	Interrupt driven serial port
Modules Required	T2
	ACIAPAK
Additional Hardware	RS232 Communications Cartridge
	Multi-Pak Expansion Interface
	External Terminal

The T2 device descriptor uses the optional RS232 Communications Card installed in Slot 1 of the Multi-Pak Expansion Interface. The associated device driver is named ACIAPAK. This port can support a terminal at data transfer rates as high as 19.2K baud. T2 expects an 80 column terminal and uses an 80 column display mode. T2 provides the following features:

- changeable baud rates, parity, word length, stop bits, and X-ON/X-OFF protocol.

- default settings of:
  - 1200 baud
  - no parity
  - 8 data bits
  - 1 stop bit

Use TMODE or XMODE to change these parameters to match your terminal.

The T2 and ACIAPAK modules are not on the standard OS-9 system diskette, but you can include them on a system diskette using CONFIG. For full information on the hardware specifications and pin-out configuration of the communications card, see the owner's manual.

T3

Use .... Virtual Interrupt driven terminal port
Modules required T3
                        MODPAK
Additional hardware .... RS232 Communications Cartridge
                         Multi-Pak Expansion Interface

Device descriptor T3 provides for a terminal port using the optional RS232 Cartridge in Slot 2 of the Multi-Pak Expansion Interface. The T3 port uses the MODPAK driver, which is a virtual interrupt driven driver (see VIRQ in this addendum). You can use T3 in conjunction with T2. The characteristics of T3 are:

- 300 baud rate only
- no parity
- 8 data bits
- 1 stop bit

The T3 and MODPAK modules are not included on the standard OS-9 system diskette, but you can add them to a system diskette using CONFIG.

P

Use .... Serial printer port
Modules required ...... P
                        PRINTER
Additional hardware .... Printer

Device descriptor P uses the PRINTER device driver to pro-
vide serial printer support through the port at the rear of the
Color Computer. The port can support a serial printer at data
transfer rates as high as 9600 baud. If transmission on your
printer is unreliable, use the TUNEPORT utility to adjust the
driver for your printer.

P supports changeable baud rates, parity (no parity, MARK,
and SPACE), word length, and stop bits. Default settings are:

- 600 baud
- no parity
- 8 data bits
- 1 stop bit
- 80 column output

The PRINTER driver uses the following reserved bits of the
parity byte:

Bits 0-3   Timeout value for the printer. The Ready
            status of the printer is checked at Init/
            Open time and an error is reported
            immediately. On subsequent writes to
            the printer the timeout value is counted
            down and if the device is not ready when
            it reaches zero, the error is returned.

P and PRINTER modules are contained on the standard OS-9
system diskette.

M1

Use .... Interrupt driven modem port
Modules required M1
                          ACIAPAK

The M1 device descriptor provides future enhancements to the Color Computer using Slot 1 of the Multi-Pak Expansion Interface. Both M1 and T2 use the ACIAPAK driver and thus cannot operate simultaneously. M1 supports a 300 baud modem for data communications. It assumes the screen size is 80 columns.

M1 supports changeable parity, word length and stop bits through the XMODE utility. The default settings are:

- 300 baud only
- no parity
- 8 data bits
- 1 stop bit

The ACIAPAK driver also supports communications devices such as auto answer modems. The Data Carrier Detect line is monitored and a transition from MARK to SPACE causes an E$HangUp error from a read request, and all M1 opened paths are marked for killing by SCF. So, if disconnection occurs, all processes started are killed and the next caller does not begin where the previous caller ended.

The *kill* feature is controlled through bit 4 of the PD.PAR byte in the path descriptor or IT.PAR of the device descriptor. Setting bit 4 enables the kill feature, clearing bit 4 disables the feature. Bit 4 also enables the shutdown of RTS and DTR at the close of the last path. If bit 4 is cleared, RTS and DTR are disabled at *terminate* rather than at *close*. (See INIZ for information on using INIZ to keep a device from terminating.)

M1 and ACIAPAK are contained on the CONFIG/BOOT diskette, and are not on the standard OS-9 system diskette. For information on setup and hardware characteristics of the modem cartridge, see the owner's manual for the cartridge.

M2

Use .... Virtual Interrupt driven modem
                    port
Modules required M2
                    MODPAK

Device driver M2 provides for future enhancements of the
Color Computer through Slot 2 of the Multi-Pak Expansion
Interface. The M2 port uses the virtual interrupt driven
MODPAK driver (see VIRQ in this addendum). M2 supports
a 300 baud modem and can be used in conjunction with the
T2 serial port which requires the RS232 Cartridge in Slot 1
of the Multi-Pak.

M2 supports changeable parity, word length, and stop bits
through the XMODE utility. The default settings are:

- 300 baud only
- no parity
- 8 data bits
- 1 stop bit

The M2 and MODPAK modules are not included on the stan-
dard OS-9 system diskette but you can add them to a system
disk using CONFIG.

SSC

Use .... Speech/Sound output port
Modules required ...... SSC
                    SSCPAK
Additional hardware .... Speech/Sound Cartridge
                    Multi-Pak Expansion
                    Interface

The SSC device descriptor supports the optional Speech/
Sound Cartridge from Slot 2 or 3 of the Multi-Pak Expansion
Interface. Speech or sound output is received through the tel-
evision or monitor speaker.

Use your SSC device in much the same manner as a printer (P). Use I/O redirection to send output to the Speech/Sound cartridge instead of the video display, or open a path to the Speech/Sound Cartridge to write to it. Because the Speech/Sound cartridge uses the same hardware as the joystick ports, you cannot use the joysticks and the cartridge simultaneously. If you do, speech or sound is interrupted by a high pitched noise from the joysticks. However, you can alternate between using speech/sound output and joystick output without difficulty.

The SSC and SSCPAK modules are not contained on the standard OS-9 system diskette, but you can add to a system diskette using CONFIG.

NIL

```
Use ................ Device for abandoning output
Modules required ...... NIL
                        NILDRV
Additional hardware .... none
```

The NIL device descriptor and the NILDRV driver support a null device. Using the device descriptor and driver lets you abandon output from a process or enter nothing as input to a process. For example, if you run a background process that sends messages to the screen, and you do not want them displayed, you can redirect the standard output and standard error path to the NIL device.

NIL and NILDRV are not included on the standard OS-9 system diskette, but you can add them to a system diskette using CONFIG.

RBF-Type Devices

RBF-type devices are those that handle data in large blocks of bytes. See the *Random Block File Manager* chapter of the *OS-9 Technical Information* manual for information on RBF. RBF-type devices supported by OS-9 are floppy disk drives, D0-D3 and hard disk drives, H0-H3.

D0-D3

Uses . . . . Floppy disk devices
Modules required . . . . . . D0 (D1-D3 are optional)
                           CCDISK
Additional hardware . . . . Floppy disk drives

Device descriptors D0-D3 are modules that support floppy disk drives using the CCDisk device driver. If you are using a Multi-Pak Expansion Interface, put the disk controller cartridge in Slot 4 with the Multi-Pak switch set to 4.

OS-9 requires a minimal system of one floppy disk drive, D0. The standard OS-9 system diskette includes the D0 and D1 descriptors and the CCDisk driver and supports a two-drive system. If you have additional floppy disk units, use CONFIG to add the descriptors, D2 or D3, to the boot file.

H0-H1

Use . . . . Hard disk devices
Modules required . . . . . . H0 (H1 for second hard disk)
                           CCHDisk
Additional hardware . . . . 15 meg or 35 meg hard disk drives
                           Color Computer Hard Disk
                           Interface Kit
                           Multi-Pak Expansion
                           Interface

Device descriptors H0 and H1 are modules that support hard disk drives using the CCHDisk device driver. Locate the Hard Disk Interface cartridge in Slot 3 of the Multi-Pak Interface Expansion with the floppy disk controller in Slot 4.

When your OS-9 system initializes, it searches for a device /H0. If it is found, the execution directory is set to /H0/CMDS, and the data directory is set to /H0. The H0, H1, and CCHDisk modules are not found on the standard OS-9 system diskette, but you can add them to a system diskette

using CONFIG. Hard disk device descriptors are named H0—
15, H1—15, H0—35, and H1—35. Information on setting
up your hard disk drives is contained in the Color Computer
Hard Disk Interface Kit manual.

PIPEMAN Devices

PIPEMAN is the file manager for the OS-9 "pipe" facility,
as described in the *Advanced Features of the Shell* chapter of
the *OS-9 Commands* manual. Pipes let data transfer between
processes by sending the output of one process to the input
of another. There is only one device supported by
PIPEMAN.

PIPE

Use .... Create a pipeline between two
                     processes
Modules required ...... Pipe
                     Piper
Additional hardware .... None

The device descriptor PIPE supports OS-9 pipelines in con-
junction with the PIPER driver. Use an exclamation point (!)
to tell the SHELL to open a pipeline between two processes.
The standard output of the first process becomes the standard
input of the second process. Information on the use of pipes
is found in *Advanced Features of the Shell* in the *OS-9 Com-
mands* manual. Both PIPE and PIPER are contained on the
standard OS-9 system diskette.

Sample Driver

```
00001                                    nam     Nil Device
00002                                    ttl     Driver Module
00003
00004                                    use     defsfile
00005     0001              LEVEL equ   1
00006                 * ifp1
00007                                    use     ..../defs/os9defs
00434                                    opt     1
00435
00436                                    use     ..../defs/
                                                 os9sysdefs.li
00550                                    opt
00551                                    use     ..../defs/os9iodefs
00663                                    opt     1
00664                                    use     ..../defs/os9rbfdefs
00863                                    opt     1
00864                                    use     ..../defs/os9scfdefs
01001                                    opt     1
01002                                    use     ..../defs/systype
01171                                    opt     1
01172
01173                                    use     ../cc.ass/sysdefs.cc
01174
01175     000A              CPUType      set     Color
01176     0001              COCOType     set     Original
          0000              ClocType     set     0
01178     0002              PwrLnFrq     set     Hz60 Set  frequency
                                                 for U.S.
01179     FF40              DPortset     $FF40   Disk       controller
                                                 address
01180     0002              ACIAType     set     ACIA6551
01181     FF68              A.T2         set     $FF68 6551  ACIAPak
                                                 Address
01182     FF6C              A.M          set     $FF6C   Modem   Pak
                                                 Address
01183                 * endc
01184
01185                 ************************************************
01186                 *
01187                 *   Nil Driver Module
01188                 *
01189     00E1              type         set     Drivr+Objct
01190     0081              revs         set     ReEnt+1
01191     0000  87CD002D    mod          ndrv.end,ndrv.nam,
                                         type,revs,ndrv.ent,
                                         V.User
```

```
01192    000D    03              fcb       Updat.   mode
01193    000E    4E696C44 ndrv.nam fcs      "NilDrv"
01194            * edition date   comments
01195            *   3 83/02/11 - change read to return EOF
01196    0014    03              fcb       3     edition number
01197
01198    0015 5F ndrv.ent clrb              initialize
01199    0016 39          rts
01200    0017 12          nop
01201    0018            bra      ndrv.eof return EOF
         200C
01202    001A 12          nop
01203    001B 5F          clrb             write
01204    001C 39          rts
01205    001D 12          nop
01206    001E 5F          clrb             getstat
01207    001F 39          rts
01208    0020 12          nop
01209    0021 5F          clrb             putstat
01210    0022 39          rts
01211    0023 12          nop
01212    0024 5F          clrb             terminate
01213    0025 39          rts
01214
01215    0026 53 ndrv.eof comb             set carry
01216    0027            ldb      #E$EOF   get error codes
         C6D3
01217    0029 39          rts
01218
01219    002A FEAA14      emod
01220    002D    ndrv.end equ      *
01221
01222                    ttl      Device Descriptor Module
```

```
01223
01224
01225            *************************
01226            *
01227            *Nil Device Descriptor Module
01228            *
01229    00F1    type     set      Devic+Objct
01230    0081    revs     set      ReEnt+1
```

```
01231    0000 87CD0039    mod       ndsc.end,ndsc.nam,type,revs,
                                    nilmgr,nildrv
01232    000D 03          fcb       UPDAT.    mode
01233    000E 000000      fcb       0,0,0     null port address
01234    0011 18          fcb       ndsc.nam-*-1  option    byte
                                                       count
01235    0012 00          fcb       DT.SCF    Device type: SCF
01236
01237                * DEFAULT PARAMETERS
01238
01239    0013 00          fcb       0         cass=UPPER and lower
01240    0014 00          fcb       0         backspace=BS,SP,BS
01241    0015 00          fcb       0         delete=bkspace over
                                                       line
01242    0016 00          fcb       0         auto echo on
01243    0017 00          fcb       0         auto line feed on
01244    0018 00          fcb       0         null count
01245    0019 00          fcb       0         end of page pause on
01246    001A 18          fcb       24        lines per page
01247    001B 08          fcb       C$BSP     backspace character
01248    001C 18          fcb       C$DEL     delete line char
01249    001D 0D          fcb       C$CR      end of record char
01250    001E 1B          fcb       C$EOF     end of file char
01251    001F 04          fcb       C$RPRT    reprint line char
01252    0020 01          fcb       C$RPET    dup last line char
01253    0021 17          fcb       C$PAUS    pause char
01254    0022 03          fcb       C$INTR    keyboard   interrupt
                                                       char
01255    0023 05          fcb       C$QUIT    keyboard quit char
01256    0024 08          fcb       C$BSP     backspace echo char
01257    0025 07          fcb       C$BELL    line overflow char
01258    0026            fcb       0,0       undefined init/baud
         0000                                         rate
01259    0028            fcb       0         no echo device
         0000
01260    002A  ndsc.nam fcs        "Nil"     device name
         4E69EC
01261    002D  nilmgr   fcs        "SCF"     file manager
         5343C6
01262    0030  nildrv   fcs        "NilDrv"  device driver
         4E696C44
01263
01264    0036 E07D20     emod                Module CRC
01265    0039  ndsc.end equ        *
01266
01267                     end

                   000000 error(s)
                   000000 warning(s)
                   $0066 00102 program bytes generated
                   $03C3 00963 data bytes allocated
                   $2875 10357 bytes used for symbols
```

RADIO SHACK, A Division of Tandy Corporation

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

AUSTRALIA	BELGIUM	FRANCE	U. K.
91 Kurrajong Avenue	Rue des Pieds d'Alouette, 39	BP 147-95022	Bilston Road Wednesbury
Mount Druitt, N.S.W 2770	5140 Naninne (Namur)	Cergy Pontoise Cedex	West Midlands WS10 7JN

To OS-9 users. . .

The new Version 01.01.00 extends the capabilities of your OS-9 Operating System by adding new function codes to provide you with more information about the memory, the cursor, and the size of your screen. It also enhances your RS/232 cartridge with new Comcard support and allows you to patch your clock module to 50 hz. The following pages describe the procedures for using the new version.

These added features reduce the total available memory by only seven pages. In the 01.01.00 version with the new features added, you still have 62 pages of free space available and much greater control of your computer.

Please note the changes to your OS-9 Technical Information, Commands, and Program Development Manuals which are set forth in the following pages.

Pages 99 – 102 should read as follows:

OS-9 I$Getstt   103F 8D

Function:

Returns the status of a file or device.

This is a "wildcard" call. It is used to handle device parameters that:

- Are not the same for all devices
- Are highly hardware-dependent
- Must be user-changeable

The exact operation of the GET Status system call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's parameters for such functions as backspace character, echo on/off. The Get Status call is commonly used with Set Status call.

The Get Status function codes that are currently defined are listed in the "Uses of Get Status" section below.

Entry Conditions:

      A = path number
      B = function code
         (Other registers depend upon function code)

Exit Conditions:

    Depends upon function code.

    If error:

     CC = C bit set
      B = error code

Uses of Get Status

   Function codes 7 through 127 are reserved for future use.

   Function codes 128 through 255 and their parameter-passing conventions are user-definable. The function code and register stack are passed to the device driver.

The following function codes are defined: $00, $01, $02, $05, $06, $12, $13, $1c, $25, $26. The parameter-passing conventions for these function codes are given below.

SS.OPT (Function code $00): Reads the option section of the path descriptor and copies it into the 32-byte area pointed to by Register X.

Use this code to determine the current settings for editing functions such as echo and auto line feed. For a complete description of the status packet, see the section on path descriptors.

Entry Conditions:
   A = path number
   B = $00
   X = address of place to put a 32-byte status packet

Exit Conditions:
     Status packet

     If error:
  CC = C bit set
   B = error code

SS.READY (Function code $01): Tests for data available on SCFMAN-supported devices.

Entry Conditions:
   A = path number
   B = $01

Exit Conditions:
     If ready:
  CC = C bit clear
   B = $F6 (E$NOTRDY)

     If error:
  CC = C bit set
   B = error code

SS.SIZE (Function code $02): Gets the current file size (RBFMAN-supported devices only).

Entry Conditions:
   A = path number
   B = $02

Exit Conditions:
- X = ms 16 bits of the current file size
- U = ls 16 bits of the current file size

If error:
- CC = C bit set
- B = error code

SS.POS (Function code $05): Gets the current file position (RBFMAN-supported devices only).

Entry Conditions:
- A = path number
- B = $05

Exit Conditions:
- X = MS 16 bits of the current file position
- U = LS 16 bits of the current file position

If error:
- CC = C bit set
- B = error code

SS.DSTAT (Function code $12): Returns the display status.

Entry Conditions:
- A = path number
- B = $12

Exit Conditions:
- X = address of the graphics display memory
- Y = graphics cursor address; x = MSB, y = LSB
- A = color code of the pixel at the cursor address

SS.Joy (Function code $13): Returns the joystick values.

Entry Conditions:
- A = path number
- B = $12
- X = 0 (right joystick), or
- X = 1 (left joystick)

Exit Conditions:
    X = selected joystick x value (0-63)
    Y = selected joystick Y value (0-63)
    A = $FF (if the fire button is on), or
    A = $00 (if the fire button is off)

SS.Alfas (Function code $1c): Returns information about alpha screen memory.

Entry Conditions:
    A = path number
    B = $1C

Exit Conditions:
    A = caps lock status
        00 = lower case
        $FF = upper case
    X = address of buffer in memory
    Y = address of cursor in memory

SS. Cursr (Function code $25): Returns cursor information from alpha screen.

Entry Conditions:
    A = path number
    B = $25

Exit Conditions:
    A = character under cursor
    X = x position of cursor (column)
    Y = y position of cursor (row)

SS.ScSiz (Function code $26): Returns size of screen (default from descriptor)

Entry Conditions:
    A = path number
    B = $26

Exit Conditions:
    X = number of columns on screen
    Y = number of rows on screen

Note: Currently these are simply the values in the device descriptors following the XOFF byte. Descriptors may be patched to return different values.

OS-9 COMMANDS MANUAL

Add the following information to page 15.

New Comcard Support

The ACIAPAC driver for the expansion MultiPac interface #26-3024 is included on the master disk, along with a device descriptor named "/T2." The Com Board RS/232 cartridge #26-2226 must be plugged into slot #1 of the MultiPac. The disk controller is plugged into slot #4 of the MultiPac. X mode and T mode will allow /T2 to use XON/X-OFF. To use them make sure your MultiPac cartridge selector switch is in position #4.

> Note: When connecting an RS-232C device, check the device's owner's manual for pin out descriptions.

> The Com Board's (#26-2226) Data Set Ready (DSR), Data Carrier Detect (DCD), and Clear to Send (CTS) RS-232C handshake lines require active signals.

Add the following codes to the Display Control codes on page 132.

Cursor ON-OFF (All numbers are in decimal.) Displaying a code of 05 followed by a qualifier code changes the cursor.

Lead in	C#	Char	Function
04			Clear to end of line
05	32	(space)	Cursor OFF
05	33	(!)	Cursor ON .. Default color (Blue)
05	34	(")	Cursor ON .. Black
05	35	(#)	Cursor ON .. Green
05	36	($)	Cursor ON .. Yellow
05	37	(%)	Cursor ON .. Blue
05	38	(&)	Cursor ON .. Red
05	39	(")	Cursor ON .. Buff
05	40	(( )	Cursor ON .. Cyan
05	41	( ))	Cursor ON .. Magenta
05	42	(*)	Cursor ON .. Orange
11			Clear to end of screen

OS-9 PROGRAM DEVELOPMENT MANUAL

Add the following information to page 161.

A command procedure to patch the resident version of the clock module to 50 hz has been included on the master disk. To use this command, first make a backup of the system master. On the new backup copy type "CLOCKPATCH.COM". After you have completed the procedure, reboot the system.

Thank You!
Radio Shack
A Division of Tandy Corporation

875-9555