OS-9 Level Two



TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

USA LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of 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. 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 Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void 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, a participating Radio Shack franchisee or a participating 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, a 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 EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

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. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY 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 a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.

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.

4/87

OS-9
Level Two
Operating System

OS-9 Level Two Operating System (software):
© 1986, Microware Systems Corporation
Licensed to Tandy Corporation.
All Rights Reserved.

All portions of this software are copyrighted and are the proprietary and trade secret information of Tandy Corporation and/or its licensor. Use, reproduction or publication of any portion of this material without the prior written authorization by Tandy Corporation is strictly prohibited.

OS-9 Level Two Operating System (manual):
© 1986, Tandy Corporation.
All Rights Reserved.

Reproduction or use of any portion of this manual, without express written permission from Tandy Corporation and/or its licensor, is prohibited. While reasonable efforts have been made in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors in or omissions from this manual, or from the use of the information contained herein.

Tandy is a registered trademark of Tandy Corporation.

OS-9 is a trademark of Microware Systems Corporation.

BASIC09 is a trademark of Microware Systems Corporation and Motorola, Inc.

10 9 8 7 6 5 4 3 2

Getting Started With OS-9

About This Manual

Using Your OS-9 Handbook

If you feel that starting a new computer *operating system* is a "scary business," relax. This handbook is designed to put you at ease when using OS-9. It is divided into two parts—each part has a different purpose.

What is in Part 1

"Part 1" of this handbook is designed to show you, step by step, how to set up and use your computer with OS-9. Follow the steps as they are described, and OS-9 is your obedient servant. The few instructions in "Part 1" are all that many OS-9 users ever need.

What is in Part 2

"Part 2" is for the more adventurous. OS-9 has an extensive repertoire of commands and functions to create and manage data and to make use of *peripherals* (devices you can connect to your computer, such as disk drives and printers). If you want to learn more about the operating system, and if you like to explore, "Part 2" is for you. You learn other useful OS-9 commands that prepare you to make use of all the functions and commands described in *OS-9 Commands*.

Contents

Part 1 / What You Need to Know About OS-9

Chapter 1 What is an Operating System? 1-1
Instructing Your Operating System 1-1
Using Application Programs and Computer Languages 1-2
Using Peripherals 1-3
Why Use OS-9? 1-4
How Much Do You Need to Know About OS-9? ..1-5

Chapter 2 How to Start and Exit Your System ...2-1
Booting OS-9 2-2
Rebooting OS-9 2-3
Exiting OS-9 2-3
Upper- and Lowercase Characters 2-4
OS-9 Error Messages 2-4

Chapter 3 What You Need to Know to Use Floppy Drives 3-1
Write Protection for Diskettes 3-2
Disk Drive Names 3-2
Making Copies of Diskettes 3-3
Formatting With One Disk Drive 3-3
Formatting With Two Disk Drives 3-4
Using the Backup Command 3-5
Making Copies With One Disk Drive 3-5
Making Copies With Two Disk Drives 3-7

Part 2 / Organization, Commands, and Keys

Chapter 4 Files and Directories 4-1
About Files 4-1
About Directories 4-1
Multiple Directories 4-4
About File and Directory Names 4-4
Examples of Filenames 4-4
About Pathlists 4-5
Anonymous Directory Names 4-6
About Device Names 4-6

Chapter 5 Commands and Keys 5-1
Typing Commands 5-1
Editing Commands 5-1
Command Parameters 5-2
Using Options 5-2
Using Commands 5-3
Accessing Commands 5-4
Commands from Disk 5-5
Changing the Execution Directory 5-6
Changing the Data Directory 5-7
Changing System Diskettes 5-7
Video Display and Keyboard Functions 5-8
Special Keys 5-8

Chapter 6 OS-9 Toolkit 6-1
Viewing Directories 6-1
Creating Directories 6-1
Deleting Directories 6-2
Displaying Current Directories 6-2
Copying Files 6-3
Deleting Files 6-4
Renaming Files 6-4
Looking Inside Files 6-5
Loading Command Modules into Memory 6-5
Listing the Command Modules in Memory 6-5
Deleting Modules from Memory 6-6
Using Other Commands 6-6

Chapter 7 Customizing Your System 7-1
Creating a New System Diskette 7-1
Monitor Types 7-8
Using Windows 7-9
Establishing a Window 7-9
Changing Window Colors 7-11
Eliminating a Window 7-12
Using Startup to Establish a Window 7-13

Index

Part 1

What You Need to Know About OS-9

Chapter 1

What is an Operating System?

OS-9 is a disk *Operating System* (that's what OS stands for). An operating system is a group of programs acting as a message center and an interpreter. Using your instructions, an operating system manages the computer's working circuits.

In fact, thinking of OS-9 as your computer manager is helpful. The boss (that's you) gives orders. OS-9 (the manager) sees they get done.

To operate OS-9 you need at least one floppy disk drive attached to your computer. OS-9 is originally configured to recognize two floppy disk drives. Later, this handbook describes how to let OS-9 know if you have more than two floppy disk drives, or if you have other hardware (printers, modems, hard disks, and so on) you want it to recognize.

Instructing Your Operating System

You give your commands to OS-9 by typing them. Because OS-9 does exactly (and only) what you tell it, your entries must be precise and have perfect *syntax* (spelling and form). You must also be sure to give OS-9 every detail it needs to perform a task.

For instance, if you told your office manager to, "Make a phone call," what can the manager do? Obviously, not much that is helpful to you. The manager must know who to call, the phone number, and what to say. OS-9 is the same. It must have all the details before it can carry out your commands properly.

To show you how to instruct your operating system, the handbook asks you to type characters, words, and lines on your keyboard. When you do, you are issuing *commands* to OS-9. Technically, a command is only one word that describes the action you want OS-9 to perform. A *command line* is a command with all of its qualifiers.

In this manual, command lines usually contain words in boxes, such as [ENTER]. These indicate keys that you press.

The manual also asks you to **press** key sequences. For instance, when asked to press [CTRL] [C], hold down the key marked CTRL, and while holding down [CTRL], press [C].

Characters that are not in boxes are typed individually. For instance, if you are asked to type the command line `format /d0` [ENTER], press each key individually ([F] [O] [R] [M] [A] [T] [ ] [/] [d] [0] [ENTER]).

If you make a mistake while typing, use [←] to move back to the error. Then retype that portion of the line.

Using Application Programs and Computer Languages

A computer *application* is a program designed to accomplish specific tasks. There are application programs to help you write letters or documents (word processors), keep a mailing list (data managers), and keep financial records (accounting packages). There are also programs to help you study for a test, play a game, play music, draw a picture, and much more.

Such application programs usually require that you use OS-9 to start your computer. A few application programs let you start directly from the application diskette. Different programs can require different procedures, and you should check your application program's documentation for specific instructions.

Application programs have special screen displays and *menus* to instruct you, or that require you to perform a particular action, such as press a key. When you are operating from an application program, that program passes your instructions to OS-9. OS-9 manages the computer's operations in the background, and its functions are invisible to you.

You can also use computer *languages* to write your own application programs. BASIC is a language. If you read the *Color Computer Disk System* manual, you already know a bit about it. There are languages you can purchase to use with OS-9 to create programs, such as assembly language, Pascal, C, and BASIC-09.

Like application programs, each language has its own startup method. The manuals that come with the languages tell you how to get them running on your Color Computer 3.

Using Peripherals

OS-9 lets you control much more than your computer's operations. It also gives you control over other hardware devices such as disk drives, a printer, modems, windows, other terminals, and so on.

Each device has a "System Name," an abbreviation preceded by a slash (/). OS-9 can only recognize a device if you type its name exactly as shown below. See Chapter 7, "Customizing Your System" for information on how to tell OS-9 what devices you want it to handle.

System Name	Description
/P	A printer connected through your computer's RS-232 port. The RS-232 port is a *serial* port, and you must have a printer with a serial connection, such as the Radio Shack® DMP 430.
/T1	A data terminal or another computer acting as a terminal, connected through the RS-232 port of your computer. If you are using another computer as a terminal, it must run a terminal program that makes it perform as a terminal.
/T2	Another data terminal or another computer acting as a terminal, connected to an optional RS-232 communications pak in a Multi-Pak Interface. If you are using another computer as a terminal, it must run a terminal program that makes it perform as a terminal.
/T3	Another data terminal or another computer acting as a terminal, connected to the optional RS-232 communications pak in a Multi-Pak Interface. If you are using another computer as a terminal, it must run a terminal program that makes it perform as a terminal.
/M1	A modem using an optional 300-baud modem pak in the optional Multi-Pak Interface. A modem allows you to communicate with other computers either directly or over phone lines.

System Name	Description
/M2	Another modem using an optional 300-baud modem pak in the optional Multi-Pak Interface.
/D0	A floppy disk drive.
/D1	Another floppy disk drive
/W, /W1, /W2, /W3 /W4, /W5 /W6, /W7	Windows that you can establish on your OS-9 system. You use CLEAR to *page* among windows you create. See "Using Windows" in Chapter 7 and *OS-9 Windowing System Owner's Manual* for information on creating windows.

Why Use OS-9?

You now know that OS-9 is an operating system for your Color Computer. You might also have heard that, in the world of computer operating systems, OS-9 is a leader. Perhaps that is why you bought it. OS-9 stands out for several reasons. Some of its strong points are:

- *File* managing capabilities.
- *Multi-user* features. With OS-9, more than one person can use the same computer at the same time.
- *Multi-tasking*. OS-9 can handle several jobs at the same time.
- *Window* functions that let you divide your display screens into sections in which you can have one or more operations running, all at the same time.
- *Input/Output* capabilities. OS-9 can communicate with TVs and monitors, disk drives, printers, and other computers.
- A sophisticated repertoire of commands.
- Sophisticated programming languages.

If you are not familiar with such terms as files, multi-user, multi-tasking, and commands, don't worry. The handbook explains these terms and more.

Programmers like OS-9 because of its powerful features. It lets them show off all of their skills. As a result, another OS-9 feature is the wide range of excellent programs that you can use with the system.

How Much Do You Need to Know About OS-9?

You might wonder how much you really need to know to use OS-9. The answer varies with your needs, and with the application programs you intend to use.

However, regardless of how you intend to use your computer, there are some OS-9 procedures you must know. For instance, you must know how to load OS-9, how to prepare diskettes to store data, and how to make copies of data or entire diskettes. This part of your handbook makes these jobs easy.

Regardless of how careful you are, there are times when things go wrong. When this happens, OS-9 displays an *error message* on the screen. This part of the manual also helps you to understand error messages and what to do about them.

How to Start and Exit Your System

Starting your computer and initializing an operating system is called *booting*. In a sense, the computer is pulling itself up by its bootstraps.

To run OS-9, Level II, you must have a Color Computer 3 with at least one floppy disk drive. Your OS-9 system diskette includes modules to support the following Color Computer hardware:

- Up to 512K RAM
- A Keyboard
- An Alphanumeric Video Display
- A Color Graphics Display
- Floppy Disk Drives (one or two)
- Joysticks (one or two)
- A Serial Printer
- An RS-232C Communications Port

If you connect a Multi-Pak Interface to your computer, and use the CONFIG utility from your BASIC09/CONFIG diskette (see Chapter 7), OS-9 can support the following devices:

- As many as two external RS-232 communications cards
- As many as two modem paks
- As many as two additional floppy disk drives

Note: The Multi-Pak Interface has four cartridge slots. A floppy disk controller must be in Slot 4. You can put modem paks, or RS-232 paks in Slots 1, 2, or 3.

Booting OS-9

Use the instructions in the *Color Computer Disk System* manual to turn on your computer system. After you do, the video screen displays a copyright message followed by the letters, OK. This is Disk Extended Color BASIC's way of telling you that it is ready to get to work. It is waiting for your commands.

To load OS-9, follow these steps:

1. Insert the OS-9 System Master diskette into Drive 0.

2. At the OK prompt, type:

   ```
   DOS [ENTER]
   ```

   OS-9 starts. If the DOS command returns a syntax error (SN? ERROR), be sure you entered the command correctly. If DOS still returns the error, check to make sure you have installed your disk cartridge properly.

3. After OS-9 displays its startup message, this prompt appears:

   ```
       yy/mm/dd hh:mm:ss
   Time?
   ```

4. Type the year, month, date, hours, minutes, and seconds in the format requested; then press [ENTER]. For instance, if the date and time is September 3, 1986, 1:22 p.m., type:

   ```
   86/09/03 13:22 [ENTER]
   ```

   Note that the time is entered in 24-hour notation and that the seconds (:SS) are optional.

   You can bypass this time and date prompt by only pressing [ENTER]. However, if you do, OS-9 cannot provide the correct date when you create and save data on disk. Also, it cannot provide the correct date and time for application programs that require them.

   After you enter the date and time, the OS-9 prompt appears and OS-9 is now in control and ready to accept a command.

   You should always keep the OS-9 System diskette in Drive 0 (/D0) while running OS-9 unless you have a hard disk containing your system files. An OS-9 System diskette is a backup copy of the OS-9 System Master diskette. The instructions for making copies are in the next chapter.

Rebooting OS-9

If you need to reboot OS-9 after the initial startup, press your computer's reset button (located at the right rear of the computer). Pressing the reset button one time causes the OS-9 boot message to reappear. The system then loads as it did originally. Be sure the System Master diskette is in Drive /D0 when you reboot.

Pressing the reset button twice returns the computer to Disk BASIC.

Exiting OS-9

In the same manner that you use OS-9 to start operations, you should use OS-9 to exit or close operations. For instance, if you are in the middle of a process, it is unwise to suddenly turn off your computer. Doing so can destroy files or garble disks.

You can usually terminate an operation by pressing [BREAK] or [CTRL] [E]. In some instances, you must let an operation complete its function before you can regain control of OS-9. If you are using an application program, that program's manual tells you how to exit the program to the OS-9 command level.

You should always be at the OS-9 command level to turn off your computer. Then follow these steps:

1. Be sure the OS-9 system prompt and cursor are displayed.

   Note: You can *turn off* the OS-9 cursor. If you or an application program has done so, the cursor does not display at the command level.

2. Take out any floppy diskettes from the disk drives, put them back in their protective envelopes, and store them in a safe place.

3. Turn off all the equipment attached to your computer such as a printer or disk drive(s); then turn off your TV or monitor. Last of all, turn off your computer and Multi-Pak Interface (if you have one). If you plug your equipment into a power strip, you can use the power strip switch to turn off all equipment at one time.

Upper- and Lowercase Characters

OS-9 can display both upper- and lowercase letters. However, you can tell it you want to use only uppercase. To do this, type:

```
tmode upc [ENTER]
```

If you do this, you cannot type lowercase letters, and the system displays all uppercase letters. To switch back to both uppercase and lowercase, type:

```
tmode -upc [ENTER]
```

Even when you are in the upper-/lowercase mode, you can switch to typing all uppercase by pressing [CTRL] [0]. Everything you type is now uppercase, but the computer can display both upper- and lowercase. Press [CTRL] [0] to switch back to upper-/lowercase.

If you want to type only one uppercase letter, hold down [SHIFT] while you press that letter.

It does not matter to OS-9 whether you type in uppercase or lowercase letters, or any combination of upper- and lowercase letters. For instance, instead of typing `TMODE UPC`, you can type `tmode upc` or `Tmode UPC`.

OS-9 Error Messages

Everyone makes a mistake now and then when typing commands. If you type something the operating system doesn't recognize, or if you ask it to do something it cannot do, it displays an *error message*. This message is a number that refers to the type of problem that OS-9 has encountered. For instance, if you type `xxxx` [ENTER] (which is nonsense to OS-9), the system displays:

```
ERROR #216
```

If you don't know the meaning of the system error number you have two options: (1) you can look up the reference in *OS-9 Commands* under Appendix A, "Error Codes" or, (2) you can type:

```
ERROR 216 [ENTER]
```

Either method shows you that Error #216 means "Path Name Not Found." OS-9 thought you wanted it to execute a command but it could not find one named `xxxx`.

Other OS-9 error messages tell you if you have used all of a disk's storage space, if the computer's memory is full, if you try to create two files with the same name, and so on.

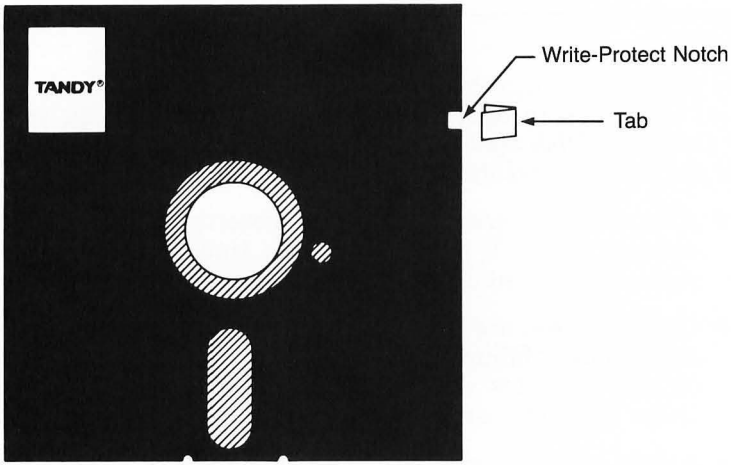
Chapter 3

What You Need to Know To Use Floppy Drives

Floppy diskettes require careful handling. You might already be familiar with how to take care of diskettes from reading your *Color Computer Disk System* manual. If not, or as a reminder, review the following points:

- Always make copies of important diskettes. The price of a diskette is small compared to the time it can take to replace destroyed data.
- Copy data you are working with regularly. If you experience a power failure while using your computer, the data on any diskettes you have in a drive can be destroyed. Other accidents can happen as well.
- Always keep the protective paper or cardboard envelope on your diskette when it is not in use.
- Your drive accesses a diskette through the oblong slot in the diskette's jacket. **Never** touch the diskette through this hole. The oil from even the cleanest hand can destroy data, making the diskette useless.
- Do not bend diskettes.
- Store diskettes away from excessive heat, dust, and any magnetic source. Even components in disk drives, video displays, TVs, and electric motors can garble the data on diskettes.
- If you must write on a diskette label after placing it on the diskette, use only a soft felt pen.
- Do not switch your computer, disk drive(s), or Multi-Pak interface on or off while you have a diskette in a disk drive.

Write Protection for Diskettes



Most diskettes have a square notch cut from one corner. This is a *write protect* notch. If you place a special adhesive tab (supplied with diskettes) over both sides of this notch, your computer can no longer write (store) data on it. This feature protects diskettes from inadvertent destruction of data.

Removing the tab again lets you write data onto the diskette.

Disk Drive Names

OS-9 has its own method of referring to your disk drives. What your *Color Computer Disk System* manual calls Drive 0, OS-9 calls Drive /D0. This is your first drive if you have more than one floppy disk drive connected to your system. Subsequent drives are named /D1, /D2, and so on.

If you have a hard disk attached to your system, OS-9 refers to it as Drive /H0. A second hard disk drive is named /H1.

Making Copies of Diskettes

Before you can store information on a diskette, you must *format* it. Formatting is the process of magnetically arranging a disk's surface so that OS-9 can store and locate information. The following steps tell you how to format a diskette. Format at least two diskettes at this time to use in making backups (copies) of your two OS-9 system diskettes. If you have other important diskettes to backup, format as many diskettes as you require.

Formatting With One Disk Drive

1. If you have not already done so, place a write-protect tab on your System Master diskette. Then, turn on and boot your computer as described in Chapter 2.

2. With the OS-9 System Master diskette in your drive, type:

   ```
   load format [ENTER].
   ```

3. Select a diskette that does not contain data or that contains data you do not want to keep. Make sure it does not have a foil tab covering the write-protect notch. Put it in your disk drive (Drive /D0) in place of your OS-9 System Master diskette and type:

   ```
   format /d0 [ENTER]
   ```

   The following prompt appears:

   ```
   COLOR COMPUTER FORMATTER
   Formatting drive /d0
   y (yes) or n (no)
   Ready?
   ```

4. Press [Y] to begin formatting. OS-9 asks you for a `Disk Name:`. Type any name, using a maximum of 32 characters. For example, you can type `s` [ENTER] to name the diskette "s."

   Next OS-9 verifies that the diskette is formatted properly. The screen shows each *track* number in hexadecimal notation during verification. A track is a concentric ring around the diskette on which information is stored.

5. When formatting is complete, OS-9 shows you the `Number of good sectors`. This number depends on the type of disk drive you are using. For a 35 track, single-sided drive, the number should be $000276 (hexadecimal 276 sectors). The OS-9 prompt and cursor reappear. Remove the newly formatted diskette from the drive, and store it in a safe place until you are ready to use it.

Format as many diskettes as you need by following Steps 3 through 5.

Formatting With Two Disk Drives

1. If your computer is off, turn it on, and boot OS-9 as outlined in Chapter 2.

2. At the system prompt (`OS9:`), type `format /d1` [ENTER]. The screen shows:

```
COLOR COMPUTER FORMATTER
Formatting drive /d1
y (yes) or n (no)
Ready?
```

3. Insert a blank disk, or one which does not contain data you want to keep, into Drive /D1, and close the latch. Be sure the diskette does not have a foil tab covering the write-protect notch. Press [Y].

4. OS-9 formats the diskette; then asks you for a `Disk Name:`. Type any name, using a maximum of 32 characters. For example, you can type `s` [ENTER] to name the diskette "s."

   Next OS-9 verifies that the diskette is formatted properly. The screen shows each *track* number in hexadecimal notation during verification. A track is a concentric ring around the diskette on which information is stored.

5. When formatting finishes, OS-9 shows you the `Number of good sectors`. This number depends on the type of disk drive you are using. For a 35-track, single-sided drive, the number should be $000276 (hexadecimal 276 sectors). The OS-9 prompt and cursor reappear. Remove the newly formatted diskette from the drive, and store it in a safe place until you are ready to use it.

Format as many diskettes as you need by following the same procedure.

Using the Backup Command

BACKUP is one OS-9 command that you can expect to use frequently. It is the command you use to make copies of your diskettes. **We strongly recommend that you now use the following instructions to make copies of your OS-9 system diskettes.** You can only copy diskettes that are created in the same type of disk drive you are using. Your OS-9 system diskettes are 35 track, single sided.

BACKUP uses two terms you need to understand. They are *source* and *destination*. A source diskette is the diskette that contains the program, file or data that you want to backup. The destination diskette is the blank formatted diskette you prepared to receive the copied data.

> **Note:** Some application programs you buy do not let you make copies of their diskettes. Check the program manual for information on protecting the data on these diskettes.

Making Copies With One Disk Drive

1. If your computer is off, turn it on, and boot OS-9 as outlined at the beginning of Chapter 2.

2. At the system prompt (`OS9:`), type:

   ```
   backup /d0 #32K ENTER
   ```

   This tells OS-9 to make a backup of the diskette in Drive /D0. The screen displays the following prompt:

   ```
   Ready to backup from /d0 to /d0
      ?:
   ```

3. Leave the System Master diskette in Drive /D0 to make a backup of it. To back up one of your other diskettes, for example the BASIC09/CONFIG diskette, remove the System Master diskette and replace it with the diskette you want to copy.

4. Press Y when you are ready to continue. The screen displays:

   ```
   Ready Destination, hit a key:
   ```

5. Replace the source diskette with the destination diskette. Then, press the space bar to continue BACKUP.

   When you back up one diskette to another, any data previously existing on the destination diskette is *overwritten* (destroyed). OS-9 gives you a chance to make sure you have inserted the proper destination diskette by displaying the message:

   ```
   DISK NAME
     is being scratched
   Ok ?:
   ```

   "Scratched" means that OS-9 is going to replace any data on the diskette with new data from the source diskette. BACKUP also gives the destination diskette the same name as the source diskette—the destination becomes a duplicate of the source.

6. Press [Y] to keep going. The screen asks you to:

   ```
   Ready Source, hit a key:
   ```

7. Remove the formatted diskette from Drive /D0, and replace it with the source diskette that contains the data you want to copy. Press the space bar.

   In a moment, a prompt asks you to:

   ```
   Ready Destination, hit a key:
   ```

8. Remove the source diskette and replace it with the destination diskette. Press the space bar.

9. Continue switching diskettes as the screen instructs you until you see:

   ```
   Sectors   copied: $0276
   Verify pass
   ```

   Followed in a moment by:

   ```
   Sectors verified: $0276
   OS9:
   ```

The diskette now in your drive, the destination diskette, is a duplicate of the source diskette. If you copied the System Master or the BASIC09/CONFIG diskette, store it in a safe place, and use the copy as your *working* diskette. Reserve the original diskette for making future backups.

Note: For computers with 512K of memory, OS-9 can backup a diskette faster if you replace #32K in Step 2 with #56K.

Making Copies With Two Disk Drives

1. If your computer is off, turn it on, and boot OS-9 as outlined at the beginning of Chapter 2.

2. At the system prompt (`OS9:`), type:

   ```
   backup /d0 /d1 #32K [ENTER]
   ```

   This tells OS-9 to make a backup of the diskette in Drive /D0.

   The screen displays the following prompt:

   ```
   Ready to backup from /d0 to /d1
   ?:
   ```

3. Leave the System Master diskette in Drive /D0 to make a backup of it. To back up one of your other diskettes, for example the BASIC09/CONFIG diskette, remove the System Master diskette and replace it with the diskette you want to copy.

4. Press [Y] when you are ready to continue.

   When you back up one diskette to another, the process *overwrites* or destroys any data previously existing on the destination diskette. OS-9 gives you a chance to make sure you have inserted the proper destination diskette by displaying the message:

   ```
   DISK NAME
     is being scratched
   Ok ?:
   ```

   "Scratched" means that OS-9 replaces any data on the destination diskette with new data from the source diskette. As well, BACKUP gives the destination diskette the same name as the source diskette—the destination becomes an exact duplicate of the source.

6. Press [Y] to keep going.

   Copying continues. When the procedure ends, you see:

```
Sectors   copied: $0276
Verify pass
```

   Followed in a moment by:

```
Sectors verified: $0276
OS9:
```

The diskette in Drive /D1 is now a duplicate of the source diskette. If you copied the System Master or the BASIC09/ CONFIG diskette, store it in a safe place, and use the copy as your *working* diskette. Reserve the original diskette for making future backups.

Note: For computers with 512K of memory, OS-9 can backup a diskette faster if you replace #32K in Step 2 with #56K.

Part 2

Organization, Commands, and Keys

Chapter 4

Files and Directories

Before you can use OS-9 extensively, you need to know how the system organizes and stores data on disk. The information in this section is true for both floppy diskettes and hard disks. However, because of the greater storage capacity of a hard disk, it is of particular importance to hard disk users.

About Files

Consider the information stored on disks to be of two basic types, programs and data. A program is *code* that causes your computer to execute a task. Data is information that a program uses or that a program creates.

All the information that OS-9 stores on disks, whether program or data, is stored in units called *files*. Whenever a program creates a file, OS-9 defines a portion of your disk to store it. It keeps the location of the file in a special list (called a *directory*), also located on the disk, so that it knows where to find your program or data the next time you want it.

About Directories

A directory is a storage space for filenames, other directory names, or both.

After you format a disk, it contains one directory called the ROOT directory. However, a disk can have many directories. For instance, besides the ROOT directory, your System Master diskette contains the CMDS and SYS directories. The ROOT and CMDS directories are especially important to you.

When you boot OS-9, you automatically begin operation from these two directories. The ROOT directory becomes your current *data directory* and the CMDS directory becomes your current *execution directory*.

Whenever you ask OS-9 to store a file on a diskette, it automatically stores it in the current data directory (the ROOT directory), unless you tell it otherwise. If you ask OS-9 to execute a command or program, it automatically looks for that command or program in the execution directory (the CMDS directory), unless you tell it otherwise.

Every OS-9 directory can also contain other directories, called *subdirectories*. For instance, SYS, and CMDS are established as subdirectories of the ROOT directory. Put in chart form, your ROOT directory with its subdirectories looks like this:

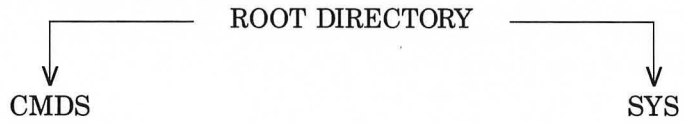


Figure 4.1

But there are also files in the ROOT directory, OS9Boot and Startup are two. The full ROOT directory might look like this:

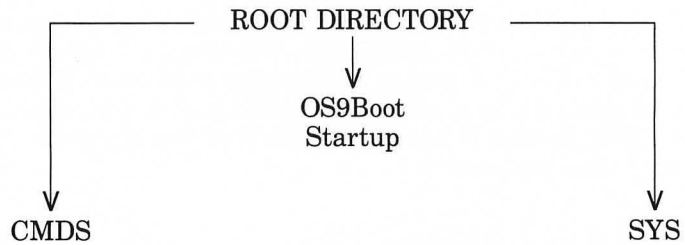


Figure 4.2

You can create another subdirectory of the ROOT directory if you want. For instance, if you created a directory named FAMILY, the chart of the ROOT directory looks like this:

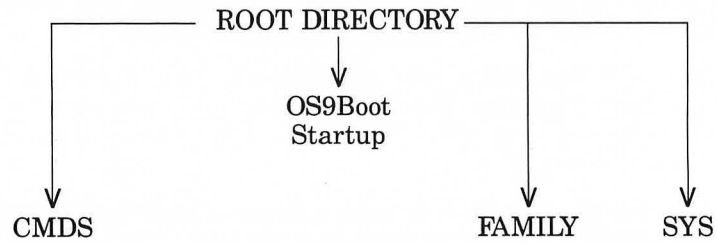


Figure 4.3

After you create the FAMILY directory, you can also create other directories in it. Suppose you create two subdirectories named PLEASURE and WORK. The chart organization is as follows:

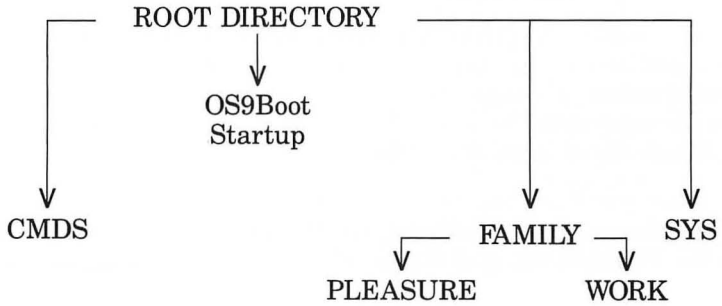


Figure 4.4

The directories you create also can hold files. If you created three files each in the PLEASURE and WORK directories, the chart might look like this:

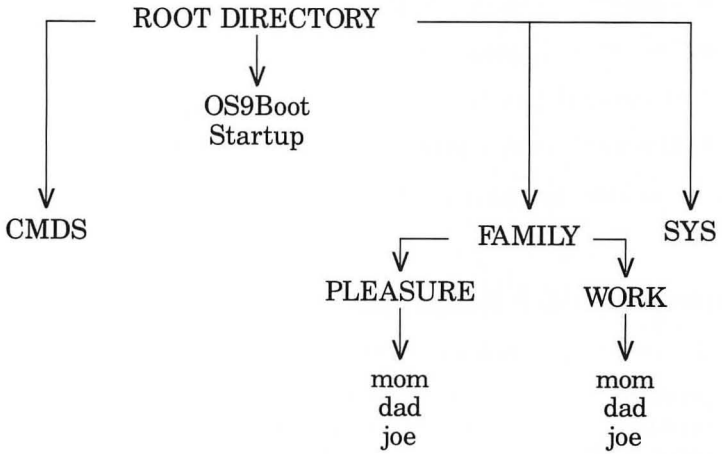


Figure 4.5

You can continue to create files and subdirectories in any or all of your disk's directories until you fill the disk's storage space.

Multiple Directories

There is nothing wrong with storing all your files in the ROOT directory. Doing so makes it easy to access them because they are always in your data directory.

However, creating multiple directories makes it easy to keep your data organized when you have many files, or if more than one person is using the same disk. Such a multiple-directory organization is especially helpful when using hard disks, which can store hundreds of individual files.

Also, when you have multiple directories, you can store files having the same name in different directories without conflict, such as in the PLEASURE and the WORK directories of Figure 4.5.

About File and Directory Names

The file and directory names shown so far consist only of letters of the alphabet, but you can use other characters and symbols in a file or directory name as long as each name begins with a letter. The following is a complete list of acceptable characters:

- Uppercase letters: (A-Z)
- Lowercase letters: (a-z)
- Decimal digits: (0-9)
- The underscore character (_) and the period (.)

You can include as many as 29 characters in a file or directory name.

Examples of Filenames

The following are samples of filenames that OS-9 can recognize:

mydata	samfile
mydata1	Dollar_gifts
records.srt	help.file
XXX.xx	file#1.txt
prog1.bas	program.sourcecode
prog2.bas	program.opcode

Examples of invalid filenames are:

his*hers .....because * is not a valid character for names
.DATA .....because the name does not begin with a letter
COST+INT .....because + is not a valid character for names
100_dollar_gifts ...because names cannot begin with a digit

About Pathlists

Because you can organize OS-9 disks into multiple levels, you need a way to tell the system where to find directories and files. The directions you give are called *pathlists*.

A pathlist is exactly what its name implies—a path (or route) to the device, directory, or file you want to access. For instance, if you are in the ROOT directory (see Figure 4.5) and want to look at the contents of a file in the WORK directory, you must tell OS-9 how to get there. The pathlist from the ROOT directory to the Dad file is `family/work/dad`. OS-9 expects you to separate the junctions of pathlists with slashes. To look at the contents of Dad, you type:

```
list family/work/dad [ENTER]
```

Because you are accessing a file on the current disk, you do not need to specify a drive name. Because every disk contains a ROOT directory, and all other directories and files branch from it, ROOT is always implied in a pathlist. If Figure 4.5 represented the diskette in Drive /D1, the pathname to the Dad file would be `/d1/family/work/dad`.

Depending on the location of the directory or file you want to access, a full pathlist need not contain any more than the name of a drive, the name of a directory, or the name of a file. For instance, the complete pathlist from the ROOT directory of Figure 4.5 to the Startup file is `startup`. To look at the contents of Startup, type:

```
list startup [ENTER]
```

Anonymous Directory Names

To save time, or if you do not know a full pathlist, you can refer to the current directory, or to a higher-level directory, using an *anonymous* name, or name substitute, as follows:

- One period (.) refers to the current directory
- Two periods (..) refer to the *parent* of the current directory (the next highest-level directory).
- Three periods (...) refer to the directory two levels up, and so on.

You can use an anonymous directory name in place of a pathlist or as the first name in a pathlist. Some examples are:

`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.

Anonymous names can refer to either execution or data directories, depending on the context in which you use them.

About Device Names

In the same manner that OS-9 has names for its commands, it also has names for its devices. These names are abbreviations of actual device names. For instance, instead of typing Disk Drive 0 to refer to your first disk drive, you only need to type `/D0`. To refer to your printer, type `/P`. OS-9 windows are named /W through /W7.

All of OS-9's device names are preceded by a slash—this is how OS-9 can tell you are referring to a device rather than a directory or file. When you purchase your System Master diskette, OS-9 is configured to recognize two disk drives, a printer, and one terminal port. For information on how to configure your system to recognize other devices, see Chapter 7.

Commands and Keys

You already put OS-9 to work with commands such as FORMAT and BACKUP. In these cases the manual told you exactly what to do to accomplish a very specific task. If you want to strike out on your own, you should know some additional background information.

Typing Commands

As explained earlier, some OS-9 files are programs. You tell OS-9 to execute these programs by typing the program (file) name and pressing [ENTER]. You are then issuing a command to OS-9. That's all a command is, the name of a program for the system to execute. The following are some rules about commands:

- You can enter a command whenever the screen displays the system prompt (`OS9:`).
- A command consists of one word, the command name. A *command line* consists of one or more command names and their associated *parameters* and *modifiers*. Parameters and modifiers are special information you include with a command that provide necessary data for the command to operate, or that affect the command's operation.
- A command line can have a maximum of 198 characters including any combination of upper- or lowercase letters. To execute a command, press [ENTER]. For example, to clear the screen, type:

```
display 0c [ENTER]
```

Editing Commands

OS-9 is very particular about the commands you type. If you make any mistake, OS-9 either does not understand (and tells you so with an error message) or does the wrong thing.

If you see that you made a mistake before you press [ENTER], you have two choices: (1) use [←] or [CTRL] [H] to move the cursor to the mistake, and retype that portion of the line, or (2) press [CTRL] [X] or [SHIFT] [←] to erase the line you are typing, and start over.

Command Parameters

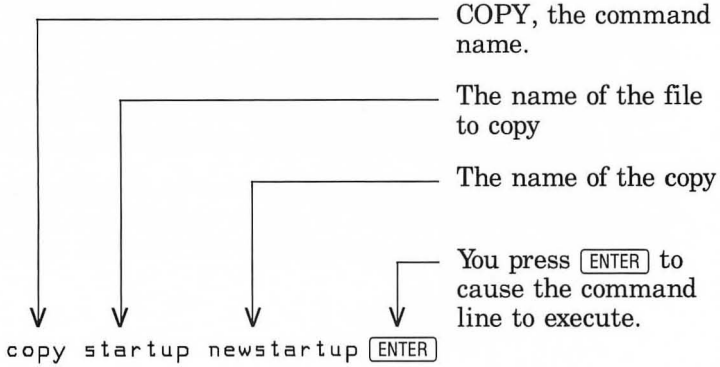
You can follow a command name with one or more parameters that give OS-9 more specific instructions. For example, in the command line:

```
list file1 [ENTER]
```

LIST is the name of the command that displays the contents of a text file. *File1*, the specified parameter, is the name of the file that you want displayed.

> **Note:** In a command line, always use spaces to separate parameters from their command, and from each other. Parameters cannot contain spaces. Chapter 6 discusses parameters for each OS-9 command.

Some commands have more than one parameter. For instance, COPY requires two parameters: the name of the file being copied, and the name of the new file you want COPY to create. If you want to copy a file called Startup, and call the copy Newstartup, your command line reads:



Using Options

Command lines can also contain another type of parameter, called an *option*. An option changes the way a command performs. For instance, the command DIR, without parameters, shows the name of all files in the current data directory.

However, if you add the E option as a parameter to the command, like this:

```
dir e [ENTER]
```

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.

To display complete information about each file in SYS, type:

```
dir sys e [ENTER]
```

Using Commands

As described in Part 1, OS-9 acts in much the same manner as an office manager. It looks after the operation of your computer and equipment. Because OS-9 is only a manager, it expects you to make the necessary decisions.

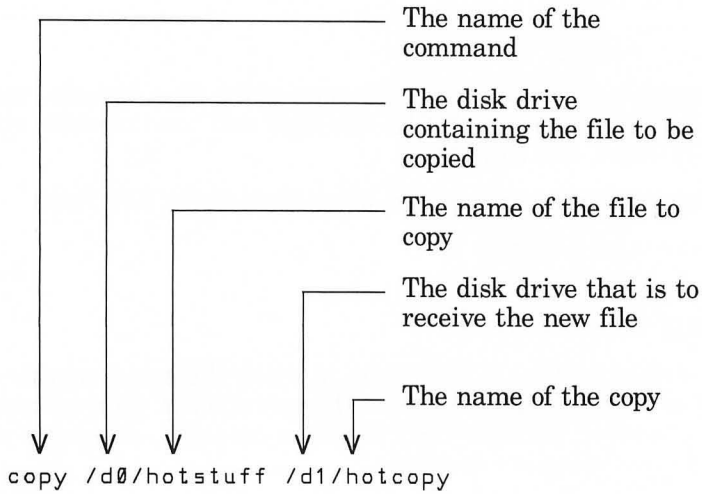
For example, suppose you have an important file named Hotstuff that you want to copy. Before giving it to your office manager (OS-9), you must make executive decisions, such as:

- Do you want the copy on disk, paper, or the computer screen?
- If you want the copy on disk, which disk?
- If you want the copy on the same disk, what name do you want to give the second copy so OS-9 is not confused?
- If you want the copy on the computer screen, do you want the display to pause when it fills the screen?

You make the decisions, OS-9 manages the job. For instance, if your decision is to copy Hotstuff from one diskette to another, you might type the following command line:

```
copy /d0/hotstuff /d1/hotcopy [ENTER]
```

This is how OS-9 sees your command:



This command line tells OS-9 to copy a file named Hotstuff from your floppy disk Drive /D0 to a second floppy Drive /D1. The file copy is given the new name, Hotcopy.

You only need to know the name of the file you want to copy, on which disk it is located, and the disk on which you want the new copy. OS-9 manages the operation for you.

Accessing Commands

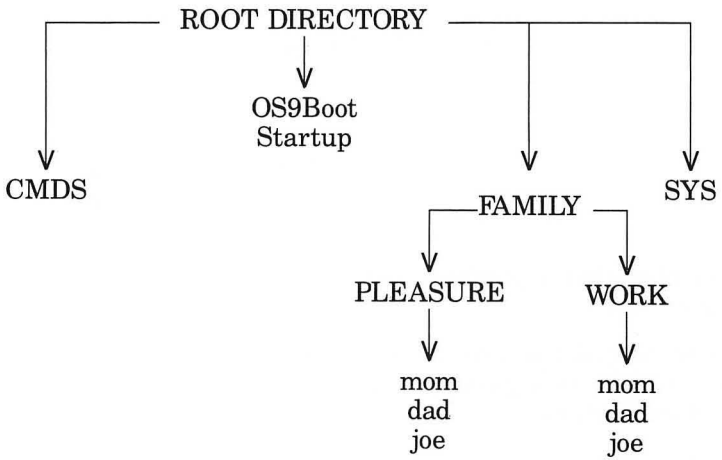
OS-9 has two ways to access commands. Some commands reside on a disk. When you type the command name and press ENTER, OS-9 must look on the disk, load the program into the computer's memory, and then execute it.

Other commands are loaded into your computer's memory at startup, or you can load them into memory later. When you call a command that is in memory, it is executed immediately. There is no delay while OS-9 finds it on disk.

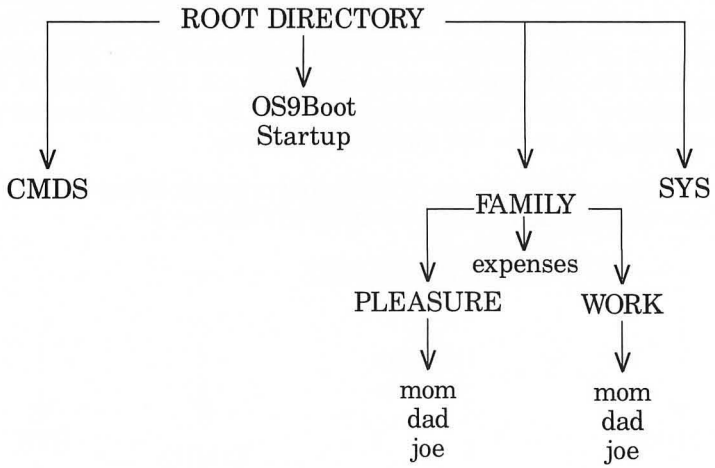
Commands from Disk

When you give OS-9 a command that it cannot find in memory, it looks for the command in the current execution directory. If it cannot find it there, it checks the current data directory. If it still cannot find it, the system issues Error Message #216, Path Name Not Found. If the command you want executed is in a directory other than the current directory, you must tell OS-9 where to find it. Remember, when initialized, OS-9 sets the CMDS directory of the system disk to be the execution directory.

For instance, suppose you booted your system using a diskette configured like the example we used in Chapter 4:



When the system starts, the ROOT directory is the data directory, and the CMDS directory is the execution directory. Now, suppose you had a program named Expenses in the family directory:



(Remember that a program and a command are really the same thing.)

You can now access (use) the expenses program in two ways. One way is to specify a pathlist from the ROOT directory to execute Expenses, such as:

```
/d0/family/expenses ENTER
```

Another way is to change the execution directory.

Changing the Execution Directory

To change the execution directory to the FAMILY directory, type:

```
chx /d0/family ENTER
```

Or specify a pathlist relative to the current execution directory, such as:

```
chx ../family ENTER
```

To execute the Expenses program, you now only need to type `expenses ENTER`.

However, after you change the execution directory, to use a command in the COMMANDS directory, you must tell OS-9 where to find it. For example, to format a new diskette in Drive /D1, type:

```
/d0/cmds/format /d1
```

Changing the Data Directory

Suppose that the Expenses program keeps track of work and pleasure expenses for Mom, Dad, and Joe. Unless you tell OS-9 otherwise, it looks for data files in the current data directory, the ROOT directory. To tell OS-9 to look for data files in the PLEASURE directory, type:

```
chd family/pleasure [ENTER]
```

The slash between FAMILY and PLEASURE tells the system that PLEASURE is a branch of FAMILY. Subordinate directories and files are always separated from their parent in this way.

Now, when Expenses needs data, it knows to look in the PLEASURE directory.

Changing System Diskettes

Although it is preferable to leave the system diskette in place while the system is running, particularly with multiuser systems, there might be times when you need to use another diskette. Only remove the current diskette when the screen displays the OS-9 prompt, followed by the cursor. If you do remove the system diskette and begin to use another one, use the CHD and CHX commands to tell OS-9 where you want to be located on the new diskette. (For directions, see Chapters 2 and 6.) Those commands set both directory pointers, data and execution, for the new diskette.

While using a program or command, do not remove a diskette and insert another unless the program or command asks you to. You can lose data, or entire files, if you do.

Video Display and Keyboard Functions

OS-9 has many features that expand the capability of the Color Computer's video display and keyboard.

- The video display has upper-/lowercase, screen pause, graphics functions, and 80 column displays if you have a monitor connected.
- The [ALT] key provides an alternate key function. Holding down [ALT] while pressing another key sets the *high order bit* of the character pressed. That is, it adds 128 to the normal ASCII value produced by that key. Holding down [ALT] while pressing any other key produces a graphics character on the standard VDG screen. If you are using windows, [ALT] lets you produce international characters. (See *OS-9 Windowing System Owner's Manual* for more information).
- The keyboard has an auto-repeat function. Holding down a key causes the character to repeat until you release the key. This function operates properly only when the disk drives are not in use by a program.
- You can deal with the video display and keyboard together as though they are a file. You can receive input from the keyboard and send output to the video screen using the device name /TERM.

Special Keys

The following keys and key sequences have special significance to OS-9.

[ALT]	Produces graphic characters on a standard VDG screen or international characters with windows. Press [ALT] *char* (where *char* is a keyboard character).
[CTRL]	A control key.
[BREAK] or [CTRL][E]	Stops the current program execution.
[←] or [CTRL][H]	Moves the cursor to the left one space.

CTRL —	Generates an underscore character.
CTRL ,	Generates a left brace ({).
CTRL .	Generates a right brace (}).
CTRL 3	Generates a tilde (~).
CTRL /	Generates a backslash (\ ).
CTRL BREAK	Performs an ESCAPE function, and sends an end-of-file message to a program receiving keyboard input. To be recognized, CTRL BREAK must be the first thing typed on a line.
SHIFT BREAK **or** CTRL C	Performs a CONTROL C function by interrupting the video display of a program. The program runs as a background task.
CLEAR *	Selects the next video window.
SHIFT CLEAR *	Selects the previous video window.
	* You must have established windows for this function key to have any effect. See "Using Windows" in Chapter 7.
CTRL CLEAR	Toggles the *keyboard mouse* on and off. The keyboard mouse uses the arrow keys and the two function keys (F1 and F2) to simulate an external mouse. When keyboard mouse is on, the normal functions for the arrow and function keys is suspended.
SHIFT ← or CTRL X	Deletes the current line.
CTRL 0	Activates or deactivates the shift lock function.
CTRL 1	Generates a vertical bar (|).
CTRL 7	Generates an up arrow (^).
CTRL 8	Generates a left bracket ([).

CTRL 9	Generates a right bracket ( ]).
CTRL A	Redisplays the last line you typed and positions the cursor at the end of the line, but does not process the line. Press ENTER to process the line, or edit the line by backspacing. If you edit, press CTRL A again to display the edited line.
CTRL D	Redisplays the current command line.
CTRL W	Temporarily halts video output. Press any key to resume output.
ENTER	Performs a carriage return or executes the current command line.

Chapter 6

OS-9 Toolkit

You now know about a number of OS-9 commands that can help you set up and use your computer system. There are many more commands available. This chapter contains information about a few of the most helpful commands. Becoming acquainted with these makes it easy for you to use other commands and functions. *OS-9 Commands* contains more information and a complete reference to all OS-9 commands (including those you have already discussed).

Viewing Directories

To look at your disk directories use the DIR command. For example, to view the contents of the current data directory, type:

```
dir [ENTER]
```

If your data directory contains more filenames than can display on the screen at one time, the display pauses. Press the space bar to cause additional files to scroll onto the screen.

You can also view your execution directory in a similar manner. This time you must include the command option, x. Type:

```
dir x [ENTER]
```

If you want to look at a directory on a disk drive other than the current drive, specify a complete path for OS-9 to follow, including the disk drive name. For example:

```
dir /d0/FAMILY/WORK [ENTER]
```

Creating Directories

Before you can store data in a directory other than the ROOT directory, you must create that directory with MAKDIR. For instance, to create a FAMILY directory on your Drive /D0 diskette, type:

```
makdir /d0/FAMILY [ENTER]
```

Deleting Directories

You can also delete directories you create. **When you delete a directory you also delete any files or subdirectories it contains; so use this command with caution.** To delete a directory, follow these steps.

1. Use DIR to view the contents of the target directory and any of its subdirectories.
2. Copy any files you want to keep into a directory outside of the directory you want to delete.
3. Type:

   ```
   deldir dirname [ENTER]
   ```

   where *dirname* is the name of the directory you want to delete.

   The screen shows:

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

4. You now have three options:

   a. To again confirm the contents of the directory before you delete it, press [l] [ENTER].

   b. To initiate the deletion process, press [d] [ENTER].

   c. To quit the process and leave the directory intact, press [q] [ENTER].

If you try to delete directories other than the ones you create, OS-9 might display Error #214, No Permission (you do not *own* the directory or have write permission for it). For information on handling such directories, see the ATTR command in *OS-9 Commands*.

Displaying Current Directories

There are times when you need to know the names of your current data and execution directories. The PWD and PXD commands make this possible. To determine your current data directory, type:

```
pwd [ENTER]
```

The command displays the path from the ROOT directory to the current data directory. For instance, if your current data directory is PLEASURE (see Figure 4.5 in Chapter 4) the display is:

```
/D0/FAMILY/PLEASURE
```

To discover your current execution directory, type:

```
pxd [ENTER]
```

The screen might display:

```
/D0/CMDS
```

A standard convention of OS-9 is to capitalize directory names. If you follow this convention when creating directories, you can always tell which files are directories at a glance.

Copying Files

COPY, like BACKUP, provides file security. If something happens to one file, you can use a copy. Also, you might want to copy a command or program to use in more than one directory, or you might want to use the same data on more than one computer.

Suppose you are in the PLEASURE directory of a diskette configured as in Figure 4.5. Your execution directory is the FAMILY directory, where you are using the Expenses program. Because the FAMILY directory does not contain any OS-9 commands, you have to change the execution directories whenever you want to use them.

You can make your work easier by copying the Expenses program to the CMDS directory. To do this, first make the CMDS directory your data directory by typing:

```
chd /d0/CMDS [ENTER]
```

Then copy the Expenses file to the CMDS directory by typing:

```
copy /d0/FAMILY/expenses expenses [ENTER]
```

Now, Expenses is in the CMDS directory, and you do not need to change the execution directory to FAMILY to use it.

Likewise, if the ROOT directory is your data directory, and you want to copy the Mom file from the WORK directory to the ROOT directory, type:

```
copy family/work/mom mom [ENTER]
```

You can copy any file between directories and between disks. To do so, you must provide the COPY command with a pathlist for the location of the original file and for the destination of its copy.

Deleting Files

You can delete files in any directory using the DEL command, such as:

```
del myfile [ENTER]
```

You can delete a file in the current execution directory by using the −x parameter. For instance, to delete Myprogram from the current execution directory, type:

```
del -x myprogram [ENTER]
```

If the file you want to delete is in a directory other than the current data directory or the current execution directory, you must specify the full pathlist to the file. For instance, suppose you are in the ROOT directory of a diskette configured as Figure 4.5. To delete the Joe file in the WORK directory, type:

```
del family/work/joe [ENTER]
```

If the file you want to delete is on a drive other than your current drive, include the drive name in your pathlist, such as:

```
del /d1/family/work/joe [ENTER]
```

If you attempt to delete a file you did not create, OS-9 might display Error #214, No Permission. For information on deleting such files see the ATTR command in *OS-9 Commands*.

Renaming Files

OS-9 lets you change the names of files. Suppose Joe leaves home, and you now want to keep track of expenses for Sue. To change the name of the Joe file to Sue, type:

```
rename family/pleasure/joe  sue [ENTER]
```

Looking Inside Files

LIST is a command that lets you examine files that consist of text characters. For instance, to view the Dad file from the WORK directory, you might type:

```
list family/work/dad [ENTER]
```

The contents of the file appears on the screen.

If you use LIST to display a file that is not a text file, it produces a meaningless display.

Loading Command Modules into Memory

When using OS-9, you might notice that some commands begin execution immediately, while others require access to the disk drive before they execute. The OS-9 commands you need most often load into memory at startup, so they are available for immediate use. If you plan to frequently use a command that is not in memory, you can *load* it.

For instance, the DSAVE command lets you copy an entire directory from one disk to another. To place the DSAVE module into your computer's memory, first be sure your execution directory is the CMDS directory, then type:

```
load dsave [ENTER]
```

Now you can use DSAVE as many times as you want, without waiting for OS-9 to find it on disk.

Listing the Command Modules in Memory

At startup, OS-9 loads into memory the commands you use most often. If you are not sure whether a command already resides in memory, you can check using the MDIR command. To display a directory of the modules in your computer's memory, type:

```
mdir [ENTER]
```

A list of all the modules in your computer's memory appear on the screen. The names you see are of modules OS-9 uses to boot and handle system operations and the commands it loads into memory when you boot the system.

Deleting Modules from Memory

After you load a module into memory, you can also delete it. The process is called *unlinking*. To delete the DSAVE command from memory, type:

```
unlink DSAVE [ENTER]
```

Some modules might require unlinking more than once, depending on the number of times they were linked.

Do not attempt to unlink modules that you did not install in memory with the LOAD command.

Using Other Commands

OS-9 has nearly 50 commands and functions. This chapter has mentioned only a few. Not only are there other commands available through OS-9, several of the commands presented here have additional options.

The guidelines you learned in this handbook provide the background you need to make use of OS-9's many other capabilities.

By referring to *OS-9 Commands* you can learn how to create files, create *procedure files* to accomplish complicated tasks, send information to your printer, transfer data between devices, execute more than one task at the same time, and much more.

Customizing Your System

Your OS-9 operating system is originally configured in a certain way. For instance, it is set up to recognize two floppy disk drives, but no hard drives. It is set up to recognize a printer or one extra terminal. It does not recognize a modem. It assumes that you only want 32 characters across your computer's display screen. It provides all of the OS-9 commands.

Using the CONFIG utility from the BASIC09/CONFIG diskette that came with your OS-9 package, you can create system diskettes that match the computer system you have. Before proceeding further, be sure you have a working copy of the BASIC09/CONFIG diskette and a blank, formatted diskette. You can use the instructions in "Making Copies of Diskettes" in Chapter 3 to create a working copy of the BASIC09/CONFIG diskette and to create a blank, formatted diskette.

Creating a New System Diskette

To create a new system diskette **make sure you have a newly formatted diskette on hand**, then follow these steps:

1. Take out the System Master diskette, and replace it with the BASIC09/CONFIG diskette. Type:

```
chx /d0/cmds [ENTER]
chd /d0 [ENTER]
config [ENTER]
```

   The first question the screen asks is:

```
HOW MANY DRIVES DO YOU HAVE:
 1 - ONE DRIVE ONLY
 2 - TWO OR MORE DRIVES
    SELECTION [1,2]
```

2. If you're using a single-drive system, press [1]. If you have more than one drive, press [2].

   If you indicated that you have two or more drives, CONFIG prompts:

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

and

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

Type the appropriate drive name (/D0, /D1, etc.) at each prompt.

3. OS-9 informs you that it is:

```
BUILDING DESCRIPTOR LIST
    .... PLEASE WAIT
```

OS-9 is putting together a list of the various devices you might want to use with your computer. When it finishes, it shows you the list:

```
                CONFIG
   ARROWS - UP/DOWN/MORE/BACK
S - SEL/UNSEL H - HELP D - DONE
  ITEM              SEL
-------------------------------
  →   P
      T1
      T2
      T3
      M1
      M2
      PIPE
      DØ_35S          X
      D1_35S
      D2_35S
```

To view the rest of this menu, press [→]. Now the screen shows:

```
                CONFIG
    ARROWS - UP/DOWN/MORE/BACK
S - SEL/UNSEL H - HELP D - DONE
  ITEM              SEL
--------------------------------
  →   D3_35S
      DDD0_35S
      D0_40D
      D1_40D
      D2_40D
      DDD0_40D
      D1_80D
      D2_80D
```

4. You can choose the various devices you plan to use with your computer from this list. To choose a device, use [↑] or [↓] to move to the device. The → shows the device you've chosen. Then, press [S] (for Select) to display an X in the SEL ("Selected") column. Pressing [S] again cancels the selection.

   You can move back and forth between the first and second screens by pressing either [→] (from the first screen) or [←] (from the second screen). Here's a short description of each device listed on this screen. To display helpful information about a device, position the → on its line in the list, and press [H] for Help. Then, press the space bar to make the help information disappear. The devices on this screen are:

   P — A printer that connects to the RS-232 serial port on your computer.

   T1 — A terminal using the standard RS-232 port (in addition to your main computer display).

   T2 — A terminal using the optional RS-232 communications pak in Slot 1 of the Multi-pak Interface. T2 supports a full baud rate range. Use T2 in addition to your main computer display alone, or in addition to your main computer display and a "T1" type terminal.

T3	Another terminal using the optional RS-232 communications pak in Slot 2 of the Multi-pak Interface.
M1	A modem using an optional 300 baud modem pak.
M2	A modem using an optional 300 baud modem pak.
PIPE	Lets you use the PIPE utility in OS-9 (a utility that takes the information a program puts out and uses it as input data in another command).
D0_35S	Floppy Disk Drive /D0, single sided, 35 tracks.
D1_35S	Floppy Disk Drive /D1, single sided, 35 tracks.
D2_35S	Floppy Disk Drive /D2, single sided, 35 tracks.
D3_35S	Floppy Disk Drive /D3, single sided, 35 tracks.
DDD0_35S	Default Disk Drive /DD using Drive /D0, single sided, 35 tracks. Select one default drive — the drive where you keep your system diskette.
D0_40D	Floppy Disk Drive /D0, double sided, 40 cylinders.
D1_40D	Floppy Disk Drive /D1, double sided, 40 cylinders.
D2_40D	Floppy Disk Drive /D2, double sided, 40 cylinders.
DDD0_40D	Default Disk Drive /DD using Drive /D0, double sided, 40 cylinders. Select one default drive — the drive where you keep your system diskette.
D1_80D	Floppy Disk Drive /D1, double sided, 80 cylinders.
D2_80D	Floppy Disk Drive /D2, double sided, 80 cylinders.

You must select a "D0" device as your first disk drive—use D1, D2, and D3 devices for additional floppy disk drives. Select the drive that matches the drives you have on your system. If you are not sure, check with your supplier. To use extra terminals and modems, you must connect them via a Multi-Pak Interface.

5. As you finish choosing among the devices on the first screen, press [→] to display another screen of devices:

6. When you finish selecting devices, press [D] for Done. The screen asks:

```
ARE YOU SURE (Y/N) ?
```

7. Now's your chance to change your mind. Press [N] if you want to reselect your devices. If you're sure about the devices you selected, press [Y].

   The next part of the CONFIG process appears on the screen:

```
                CONFIG

          SELECT TERM DESCRIPTOR

          1 - TERM_VDG
          2 - TERM_WIN
                H - HELP
       SELECTION [1,2]
```

8. These are Color Computer terminal I/O subroutine modules you can use. For a 32 character display, select 1 (TERM_VDG). In order to have OS-9 windows and an 80 column display, select 2 (TERM_WIN).

   Note: You can use TERM.WIN with a TV rather than a monitor but it is difficult, if not impossible, to see characters on an 80-column window. When you later create text windows, select 40-column displays.

If you select 2 (Term_Win), CONFIG presents you with another menu of choices. This time, the display looks like this:

```
                CONFIG
   ARROWS - UP/DOWN/MORE/BACK
S - SEL/UNSEL H - HELP D - DONE
       ITEM              SEL
------------------------------
          W                X
          W1               X
          W2
          W3
          W4
          W5
          W6
          W7
```

This list represents the pre-established windows you can open for use with OS-9. The next section in this chapter tells you how to open and use windows. For now, if you expect to open windows in which you can run mulitple tasks, select these items for your new diskette. (See "Using Windows" later in this Chapter.)

9. After you select the modules you want to use, press [D]. As it did when you selected devices, the screen asks `ARE YOU SURE (Y/N) ?` Press [Y] if you're finished. Or, press [N] to keep working on this screen.

   OS-9 creates a file called Bootlist in Drive /D0's ROOT directory, using the information you've provided so far. It lets you know what it's up to by displaying:

```
BUILDING BOOT LIST
  .... PLEASE WAIT
```

   Then, the screen asks:

```
SELECT CLOCK MODULE:
 1 - 60 HZ (AMERICAN POWER)
 2 - 50 HZ (EUROPEAN POWER)
    SELECTION [1,2]
```

10. Press [1] if you live in the United States, Canada, or any other country that uses 60Hz electrical power. If you live in a country that uses 50Hz electrical power, press [2].

11. CONFIG is ready to begin creating your customized System Master diskette. If you have one drive, the screen tells you that the DESTINATION diskette is your blank, formatted diskette and that your SOURCE diskette is the BASIC09/ CONFIG diskette. Place your formatted diskette in the drive, and press [C]. You'll swap between the formatted diskette and the BASIC09/CONFIG diskette several times.

    If you have a two-drive system, place a formatted diskette in Drive /D1, and press the space bar. The screen tells you that OS-9 is:

```
GENERATING NEW BOOT
   .... PLEASE WAIT
```

12. Following the boot file generation, a menu lets you select the commands you want to include on your system diskette. You have the following choices: none; the full set of commands; or a set consisting of commands you choose individually. The menu looks like this:

```
                CONFIG

        DO YOU WISH TO ADD
    [N]O COMMANDS, STOP NOW
    [F]ULL COMMAND SET
    [I]NDIVIDUALLY SELECT
    [H] RECEIVE HELP
SELECTION [N,F,I,H]
```

    Most people like to choose the individual commands they want to use. For the time being, press [F] to include the full set. Later, you can create another custom diskette that has only the commands you need.

13. Do one of the following:

    a. If you have one drive, the screen asks you to place your formatted diskette in Drive /D0. Do so, and press the space bar. Next, you'll place your "uncustomized" backup of the System Master diskette in Drive /D0. Swap the two diskettes as the screen asks you to. When the CONFIG program finishes, the `OS9:` message reappears. You now have a brand new, customized copy of the System Master diskette.

    b. If you have more than one drive, the screen asks you to place your system diskette in Drive /D0. CONFIG continues and in a few minutes, finishes its work. The `OS9:` message reappears, and you have a customized copy of the System Master diskette in Drive /D1.

14. Label the diskette so that you can distinguish between your working copy of the System Master diskette and the custom copy.

Monitor Types

OS-9 lets you set your system for different monitor types. The monitor options are for a RGB color monitor, a composite color monitor or TV, or a monochrome monitor or TV. To set your system for a particular monitor type, enter one of the following commands, or add it to your system Startup file:

Monitor Type	Command
RGB	montype r
Composite	montype c
Monochrome	montype m

Therefore, to set your system for a composite monitor, type:

```
montype c [ENTER]
```

To save typing the command each time you start OS-9, put it in the Startup file in the ROOT directory of your system diskette.

If your system disk does not have an existing Startup file:

Create one by typing:

```
build startup [ENTER]
montype r [ENTER]
[ENTER]
```

If your system disk already has a Startup file:

First rename the Startup file by typing:

```
rename startup oldstart [ENTER]
```

Then create a file that contains the new command, such as:

```
build newstart [ENTER]
montype r [ENTER]
[ENTER]
```

Now combine the two files into a new Startup file:

```
merge oldstart newstart > startup [ENTER]
```

Use DEL to delete oldstart, newstart, or both, or leave them on your disk for future use.

Using Windows

If the window descriptors (W, W1, W2, W3, W4, W5, W6, W7) and the graphics interface and driver, GrfInt and GrfDrv, are in memory, OS-9 lets you set up windows on your display screen.

> **Note:** GrfInt and the window descriptors must be loaded as part of the boot operation. Your System Master diskette does this.

Once you have initialized windows, you can then move among them, initiating different tasks in each. You can even have different processes showing on different portions of your display screen at the same time.

Another advantage of using windows is that you can choose windows that give you displays of 40 or 80 columns across the screen, rather than only 32. However, unless you have a monitor connected to your computer, rather than a television, you might be unable to read the screen.

Establishing a Window

You can establish one or more windows after booting OS-9, or you can include the window creation process in OS-9's Startup file. Startup is a file containing commands you want your system to execute during startup.

To establish a window from the OS-9 prompt, type:

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

In this example, *number* represents the window number to initialize. After you type these commands, you can select the window by pressing [CLEAR]. To return to the original screen, press [CLEAR] again.

The default values for the window descriptors /W1 through /W7 are:

Window device name	Text size in columns	Window's physical size Starts at:	Size:
/W1	40	0,0	27,11
/W2	40	28,0	12,11
/W3	40	0,12	40,12
/W4	80	0,0	60,11
/W5	80	60,0	19,11
/W6	80	80,0	80,12
/W7	80	0,0	80,24

Note: To initialize Windows /W2 and /W3, you must be operating from Window /W1. To create Windows /W5 and /W6, you must be operating from Window /W4.

The "Starts at" column, indicates the position on the screen of the top left corner of the window. On the screen grid, coordinates 0,0 are located at the top left corner.

The "Size:" column indicates the number of characters across each window and the number of character lines in each window.

Therefore, Window 1 displays 40 column text, begins in the top left corner of the screen, extends right for 27 characters and down for 11 lines. Window 5 displays 80 column text, begins at the top of the screen, 60 columns from the left, extends 19 columns to the right and 11 lines down.

Note that the coordinates for each window are based on the text size of the screen. Therefore, Window 1 (based on 40 column text) ends at column 27, while Window 5 (based on 80 column text) begins at column 60.

Using the information in the previous chart, you can now establish any, or all, of the seven windows.

> **Note:** You cannot establish all of the windows unless your computer has 512 kilobytes of memory.

For instance, to set up a full screen, 80-column window, type:

```
shell i=/w7& [ENTER]
```

After a short pause, the screen displays a message, such as:

```
&004
```

This means that OS-9 has opened a path to your new window and started a shell on the window with the process identification of 04. To move to the window, press [CLEAR]. Your 32-column screen vanishes and you are now in Window 7. You can type commands or run programs from here in the same manner as before.

To set up three windows on the same screen, type these commands, then use [CLEAR] to move among the windows:

```
iniz w1 w2 w3 [ENTER]
shell i=/w1& [ENTER]
shell i=/w2& [ENTER]
shell i=/w3& [ENTER]
```

If you want, and your computer has enough memory, you can run different processes in all of the windows.

Changing Window Colors

Perhaps you don't like the color of the screen in one or more of your windows. You can change it using the display command. The following charts show you all of the colors available for the screen background, text, and border.

Background Code = 33
Text Code = 32
Border Code = 34

Color Codes

Codes	Color
00 or 08	White
01 or 09	Blue
02 or 0A	Black
03 or 0B	Green
04 or 0C	Red
05 or 0D	Yellow
06 or 0E	Magenta
07 or 0F	Cyan

To change a color, type DISPLAY 1b, followed by the background, text, border, or foreground code followed by a color code. Then, press [ENTER].

For instance, if you are in Window 7, you can change the background color to red, by typing:

```
display 1b 33 04 [ENTER]
```

Change the text color to black by typing:

```
display 1b 32 02 [ENTER]
```

To put a white border around the screen, type:

```
display 1b 34 00 [ENTER]
```

You can also type all the codes on one line, like this:

```
display 1b 33 04 1b 32 02 1b 34 00 [ENTER]
```

Pick the colors you want for each window, and change them using DISPLAY.

Eliminating a Window

In the command to establish windows (shell i=/w*number*&), "i" tells SHELL that the process being created is *immortal*. This means that you can only terminate it from the window in which it resides.

To kill a window in which you have established a shell, press CLEAR until the window you want appears on the screen. Type:

```
ex [ENTER]
```

Now press CLEAR to move to another window in which a shell is running. Then use DEINIZ to deinitialize that window. For instance, if the window you want to eliminate is Window 1, type:

```
deiniz w1 [ENTER]
```

Using Startup To Establish A Window

If you intend to use a window whenever you start OS-9, for instance if you want to use an 80 column screen, put the appropriate commands in the Startup file. This file must be located in the ROOT directory of your system disk.

If your system diskette already has a Startup file:

First rename the existing Startup file, such as:

```
rename startup oldstart [ENTER]
```

Then put your new commands into a temporary file. To initialize window Number 7 (80 columns, full screen) with white text on a black background, type:

```
build tempstart
iniz w7 [ENTER]
shell i=/w7& [ENTER]
display 1b 32 00 1b 33 02 1b 34 02 0c > /w7 [ENTER]
[ENTER]
```

Now combine your new commands with the original Startup file by typing:

```
merge oldstart tempstart > startup [ENTER]
```

You can remove the Tempstart file by typing `del tempstart` [ENTER], or you can leave it in your ROOT directory for future use.

If Startup does not already exist:

Create it by typing:

```
build startup [ENTER]
iniz w7 [ENTER]
display 1b 32 00 1b 33 02 1b 34 02 0c > /w7 [ENTER]
shell i=/w7& [ENTER]
[ENTER]
```

Now, after you boot OS-9, press [CLEAR] to operate in an 80-column, black and white screen.

Index

adding commands 7-7
ALT 5-8
anonymous directory names 4-6
application 1-2
    diskette 1-2
    programs 1-2, 3-5
arrow keys 1-2, 5-8, 5-9
ASCII value 5-8
auto-repeat, keyboard 5-8

backslash character 5-9
backup 3-5
    diskettes 3-3
    files 6-3
BASIC09/CONFIG diskette 3-5, 3-7
bit, high order 5-8
booting OS-9 2-2
brace 5-9
bracket character 5-9, 5-10
BUILD 7-8, 7-13

care of diskettes 3-1
carriage return 5-10
changing
    directories 5-6
    the system diskette 5-7
character
    ASCII 5-8
    backslash 5-9
    brace 5-9
    tilde 5-9
    underscore 5-9
    up arrow 5-9
    valid 4-5
    vertical bar 5-9
clock module 7-6
CMDS directory 4-1, 5-5
colors, window 7-11, 7-12
command 1-1, 6-1
    accessing 5-4
    adding 7-7
    editing 5-1
    loading 6-5
    line, 1-1, 1-2, 5-1
    using spaces 5-2
    mistakes 5-1
    modules, listing 6-5
    option 5-2
    parameters 5-2
    process, 5-10
    typing 5-1
communications pak 1-3
composite monitor 7-8
computer, turning off 2-3, 3-1
CONFIG 7-1
configuring your system 7-1
contents of directories 6-1
control key 5-8
CONTROL-C 5-9
copies
    with one drive 3-5
    with two drives 3-7
COPY 5-2, 5-4
copying
    diskettes 3-3, 3-5
    files 6-3
CTRL-BREAK 5-9
CTRL-C 1-1
CTRL-H 5-1
CTRL-X 5-1
current execution directory 5-5
cursor, move 5-8
customizing your system 7-1

data 4-1
    directory, changing 5-7
    files 4-1
    terminal 1-3
    types 4-1
    storing 4-1
date 2-2
DEINIZ 7-13
deleting
    lines 5-9
    directory files 6-2
    files 6-4
    memory modules 6-6
descriptors, window 7-10
destination diskette 3-6
device names 4-6, 7-3
DIR 5-2

directory 4-1
    changing 5-6
    changing the data 5-7
    CMDS 4-1
    contents 6-2
    creating 6-1
    current execution 5-5
    deleting 6-2
    display 6-2
    finding 4-5
    multiple 4-4
    pathlist 4-5
    ROOT 4-1, 4-5
    SYS 4-2
    viewing 6-1
directory names 4-4
    anonymous 4-6
    displaying 6-2
disk drive 1-4
    names 3-2,
disk name 3-3, 4-6
diskette
    backup 3-5
    copying 3-3
    formatting 3-3, 3-4
    handling 3-1
    removing 5-7
    track 3-3
    BASIC09/CONFIG 3-5
    changing the system 5-7
    destination 3-6
    source 3-6
    system 7-1
    write protect 3-2
DISPLAY 7-12
display
    directory 6-2
    directory names 6-2
    file 5-2
    video 5-8
drive names 3-2
DSAVE 6-5
duplicate diskette 3-6, 3-7

editing commands 5-1
end-of-file 5-9
entering commands 5-1
error message 1-5, 2-4
ESCAPE function 5-9
examples of filenames 4-4
execution directory 5-5
exit OS-9 2-3
extended DIR 5-3

files 4-1
    copying 6-3
    deleting 6-4
    display 5-2
    finding 4-5
    names 4-4
    pathlist 4-5
    renaming 6-4
    Startup 7-8
    viewing 6-5
filename
    examples 4-4
    legal characters 4-4
    finding 4-5
    directories 4-5
formatting
    diskette 3-3, 3-4
    with two drives 3-4

graphics
    characters 5-8
    interface 7-9

halt video output 5-10
handling diskettes 3-1
hard disk name 3-2
hardware 1-3
high order bit 5-8

initialize diskette 3-3, 3-4
INIZ 7-10
interface, graphics 7-9
international characters 5-8

key, control 5-8
keyboard
    mouse 5-9
    auto-repeat 5-8
keys, arrow 5-8, 5-9
kill a window 7-13

languages 1-2
last line, redisplay 5-10
left brace 5-9
left bracket 5-9
legal characters, filenames 4-4

length, command line 5-1
line
    command 5-1
    delete 5-9
    redisplay 5-10
listing command modules 6-5
loading commands 6-5
lowercase letters 2-4, 5-1, 5-8

memory modules 6-5
    deleting 6-6
menus 1-2
messages, error 2-4
mistakes, command 5-1
modem 1-3, 1-4
modifier 5-1
module, clock 7-6
modules
    in memory 6-5
    window 7-6
monitor 5-8, 7-8
    monochrome 7-8
mouse, keyboard 5-9
move cursor 5-8
multi-pak interface 1-3, 2-1
multi-tasking 1-4
multi-user 1-4
multiple directories 4-4

names
    anonymous 4-6
    device 4-6
    directory 4-4
    disk drive 3-2
    file 4-4
    hard disk 3-2
    of devices 7-3

one drive copies 3-5
operating system 1-1
option, command 5-2
OS-9, starting 2-1
output, halt video 5-10

parameter 5-1
    command 5-2
pathlist 4-5, 4-6
periods, anonymous names 4-6
peripherals 1-3
process command 5-10
program 4-1
    application 3-5
    execution, stopping 5-8
    files 4-1
    name 5-1
protect diskettes 3-2

quit OS-9 2-3

rebooting OS-9 2-3
redisplay
    current line 5-10
    last line 5-10
removing
    diskettes 5-7
    windows 7-12
renaming files 6-4
reset button 2-3
RGB monitor 7-8
right brace 5-9
ROOT directory 4-1, 4-2, 4-5
route to files 4-5
RS-232 1-3, 2-1

scratched 3-6, 3-7
screen, VDG 5-8
sector 3-4
select window 5-9
serial port 1-3
SHELL 7-11
shift lock 5-9
size of windows 7-10
slash
    in device names 4-6
    in pathlist 4-5, 5-7
SN error 2-2
source diskette 3-6
spaces in a command line 5-2
special keys 5-8
starting
    OS-9 2-1
    your computer 2-1
Startup 7-8
    file 7-13
stopping program execution 5-8
storing
    data 4-1
    diskettes 3-1
subdirectory 4-2
substitute names 4-6

syntax error 2-2
SYS directory 4-2
system
    customizing 7-1
    devices 7-3
    diskette 7-1
    name 1-3

TERM 5-8
TERM_VDG 7-5
TERM_WIN 7-5
terminal 1-3
text file 5-2
tilde 5-9
time 2-2
track 3-3, 3-4
turning off your computer 3-1
two drive
    copies 3-7
    formatting 3-4
types
    of data 4-1
    of monitors 7-8

underscore character 5-9
up arrow 5-9
uppercase 2-4, 5-1, 5-8

valid character 4-4, 4-5
VDG screen 5-8
vertical bar 5-9
video
    display 5-8
    output, halt 5-10
    window, select 5-9
viewing files 6-5

window 1-4, 7-9
    40-column 7-5
    80-column 7-5
    colors 7-11, 7-12
    descriptors 7-10
    eliminating 7-12
    establishing 7-9
    modules 7-6
    names 4-6
    path 7-11
    size 7-10
    establish 7-13
write protect 3-2, 3-3

OS-9
Commands
Reference

OS-9™ Level Two Operating System
©1983, 1986 Microware Systems Corporation.
Licensed to Tandy Corporation.
All Rights Reserved.

OS-9 Commands:
©1986 Tandy Corporation
and Microware Systems Corporation.
All Rights Reserved.

Reproduction or use, without express written permission from Tandy Corporation or Microware Systems Corporation, of any portion of this manual is prohibited. While reasonable efforts have been taken in preparation of this manual to assure its accuracy, neither Tandy Corporation nor Microware Systems Corporation assumes any liability resulting from any errors in or omissions from this manual, or from the use of the information contained herein.

Contents

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

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

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

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

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

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

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

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

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

Appendices

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

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

C OS-9 Keyboard Codes C-1

D OS-9 Keyboard Control Functions D-1

Index

Introduction

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

The Kernel

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

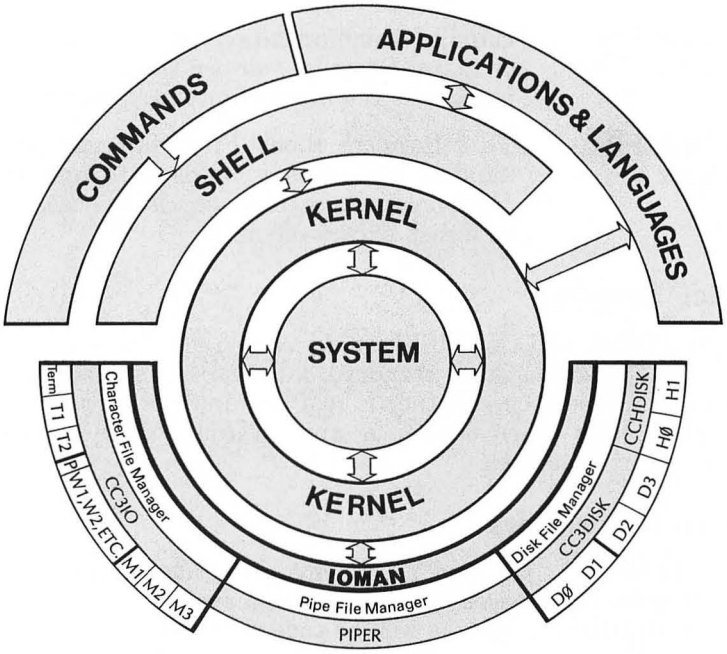


Figure 1
Term is your keyboard and video.
T1 and T2 are additional terminals.
P is a printer.
M1, M2, and M3 are modems.

The Input/Output Manager

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

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

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

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

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

Device Drivers

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

Device Descriptors

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

The Shell

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

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

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

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

Going On

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

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

The OS-9 File System

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

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

Input/Output Paths

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

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

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

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

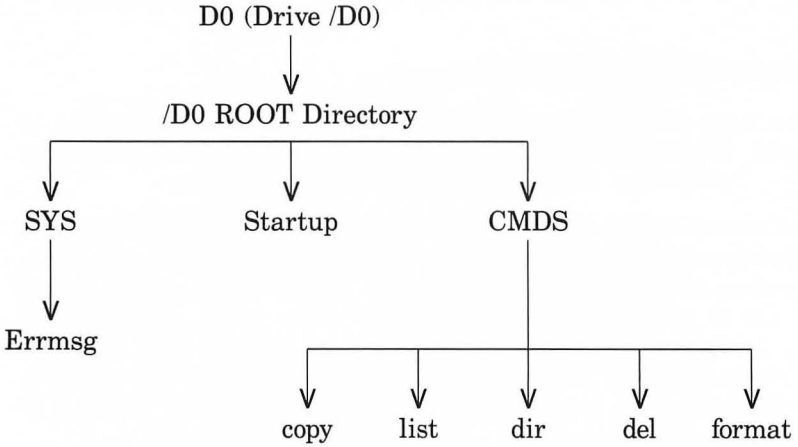
- Programs operate correctly regardless of the I/O devices selected.
- Programs are highly portable from one computer to another, even when the computers have different types of I/O devices.
- You can redirect I/O to alternate files or devices when you run a program, without having to alter the program.

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

Disk Directories

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

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



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

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

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

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

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

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

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

Subdirectories

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

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

Disk Files

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

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

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

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

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

Sectors

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

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

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

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

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

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

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

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

Text Files

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

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

Use LIST to examine the content of text files.

Random-Access Data Files

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

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

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

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

Procedure Files

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

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

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

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

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

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

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

Executable Program Module Files

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

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

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

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

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

Miscellaneous File Use

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

The File Security System

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

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

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

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

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

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

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

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

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

Public execute permission — Anyone can execute the file.

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

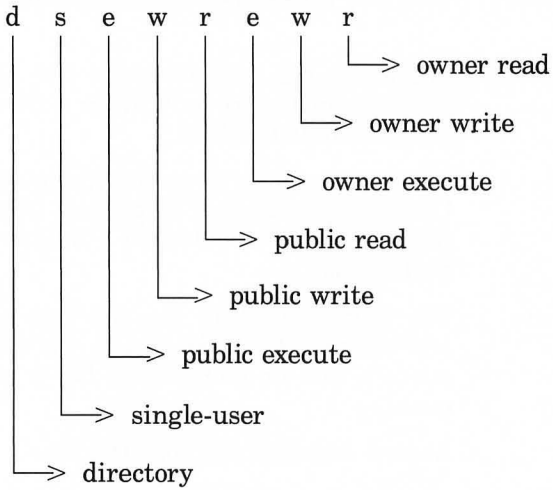
Examining and Changing File Attributes

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

```
Directory of . 10:20:44

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

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



For example, the first file shows:

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

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

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

```
attr file2 ENTER
```

A possible screen display is:

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

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

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

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

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

Record Lockout

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

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

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

Device Names

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

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

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

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

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

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

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

Advanced Features of the Shell

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

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

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

More About Command Line Processing

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

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

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

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

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

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

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

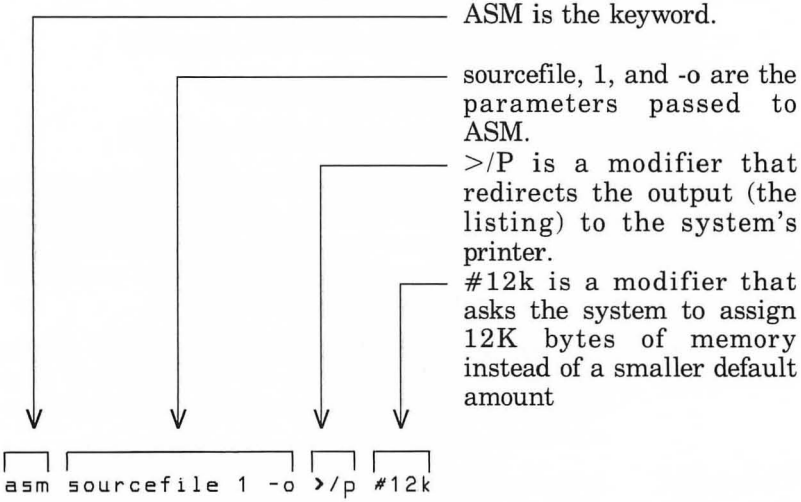
Once the shell identifies the keyword, it processes any modifiers. It then assumes the remaining text consists of parameters, which it passes to the program being called.

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

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

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

In this example:



Command Modifiers

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

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

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

Execution Modifiers

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

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

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

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

The following two examples have identical results:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

You can now view the contents of Savelisting by typing:

```
list /d1/savelisting [ENTER]
```

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

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

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

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

```
basic09 <>>>/w1 [ENTER]
```

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

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

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

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

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

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

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

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

Here are two more examples of commands using the semicolon:

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

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

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

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

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

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

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

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

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

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

```
&007
OS9:
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Command Grouping

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

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

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

or:

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

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

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

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

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

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

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

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

Shell Procedure Files

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

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

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

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

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

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

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

```
shell mailsequence [ENTER]
```

or

```
mailsequence [ENTER]
```

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

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

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

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

Built-in Shell Commands and Options

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

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

The built-in commands and their functions are:

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

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

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

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

```
ex [ENTER]
```

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

Running Compiled Intermediate Code Programs

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

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

```
basic09 adventure [ENTER]
```

or:

```
adventure [ENTER]
```

or:

```
runb adventure [ENTER]
```

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

Multiprogramming and Memory Management

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

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

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

Processor Time Allocation and Timeslicing

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

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

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

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

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

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

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

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

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

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

Process States

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

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

- **Waiting**—Applies to processes that the system suspends until another process terminates. This state allows coordination of sequential process execution. The shell, for example, is in the waiting state during the execution of a command it has initiated.

- **Sleeping**—Applies to a process suspending itself for a specified time, or until receipt of a *signal*. (Signals are internal messages that coordinate concurrent processes.) This is the typical state of processes waiting for input/output operations.

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

Creation of Processes

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

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

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

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

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

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

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

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

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

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

In summary, each new process has:

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

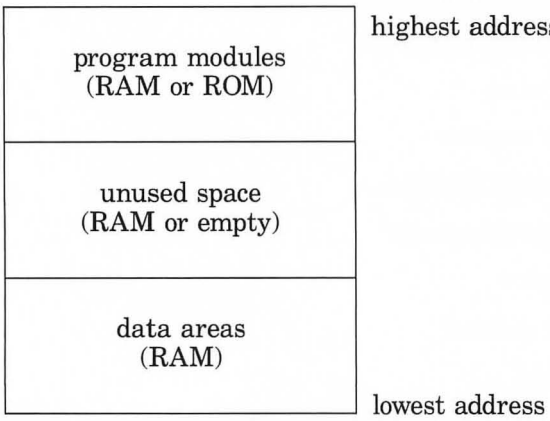
Basic Memory Management Functions

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

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

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

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



Loading Program Modules into Memory

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

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

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

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

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

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

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

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

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

Deleting Modules From Memory

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

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

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

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

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

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

Loading Multiple Programs

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

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

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

Chapter 5

Useful System Information and Functions

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

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

File Managers, Device Drivers, and Descriptors

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

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

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

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

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

The Sys Directory

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

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

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

The Startup File

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

The CMDS Directory

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

Making New System Diskettes

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

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

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

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

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

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

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

Technical Information for the RS-232 Port

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

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

```
xmode /p ENTER
```

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

Stop Bits

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

Word Length

Word Length	Value to Use
7 Bits	20
8 Bits	0

Baud Rate

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

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

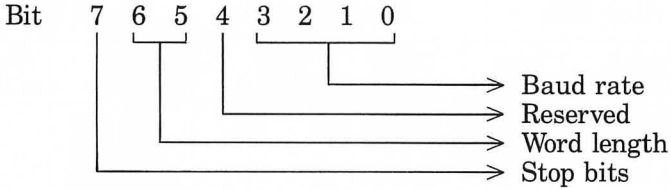
0 + 20 + 2 = 22

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

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

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

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



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

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

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

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

Parity

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

MDM Kill

Kill Switch	Value to Use
On	10
Off	0

Not Ready Delay

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

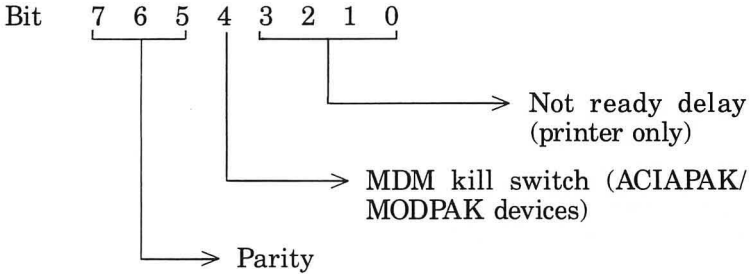
Select a value from each chart, and add them together to get a final TYPE value. For instance, to select even parity, MDM kill off, and a not ready delay of 10 seconds, select these values and add them:

60 + 0 + A = 6A

To set the new values, type:

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

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



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

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

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

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

System Command Descriptions

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

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

Organization of Entries

Each command entry includes:

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

Command Syntax Notations

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

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

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

```
build filename [ENTER]
```

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

```
build myfile ENTER
```

Pressing ENTER executes the command.

Common variables are:

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

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

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

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

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

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

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

Command Summary

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

The following list is a summary of these commands:

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

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

ATTR

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

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

Parameters:

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

Options:

The file permission abbreviations you can use are:

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

Notes:

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

Examples:

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

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

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

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

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

```
attr datalog ENTER
```

BACKUP

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

Function: Copies all data from one disk to another.

Parameters:

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

Options:

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

Notes:

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

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

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

  ```
  backup [ENTER]
  ```

  The screen display and your input are:

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

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

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

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

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

The screen display and your responses are as follows:

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





```
Ready Destination, hit a key: [ENTER]
Sectors copied: $0276
Verify pass
Sectors verified: $0276
```

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

Examples:

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

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

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

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

BUILD

Syntax: **build *filename***

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

Parameters:

filename	The name of the file you are creating.

Notes:

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

```
build newfile [ENTER]

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

- To view the newly created file, type:

```
list newfile [ENTER]
```

The screen displays:

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

Examples:

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

  ```
  build small_file [ENTER]
  ```

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

  ```
  build /p [ENTER]
  ```

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

  ```
  build <mytext /t1 [ENTER]
  ```

CHD
CHX

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

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

Parameters:

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

Notes:

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

Examples:

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

```
chd /d1/programs [ENTER]
```

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

```
chx .. [ENTER]
```

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

```
chx binary_files/text_programs [ENTER]
```

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

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

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

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

CMP

Syntax: **cmp *filename1 filename2***

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

Parameters:

filename1	are the files to compare.
filename2	

Notes:

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

Examples:

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

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

Following is a sample screen display:

```
Differences

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

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

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

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

The screen display might be:

```
Differences

None ...

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

COBBLER

Syntax: **cobbler *devname***

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

Parameters:

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

Notes:

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

Examples:

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

```
cobbler /d0 ENTER
```

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

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

CONFIG

Syntax: config

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

Notes:

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

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

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

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

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

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

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

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

Examples:

The following steps take you through the complete CONFIG process:

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

   ```
   config [ENTER]
   ```

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

   If you specify one drive, continue with Step 3.

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

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

   Type `/d0` [ENTER]

   A display now asks you to:

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

   Type `/d1` [ENTER]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPY

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

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

Parameters:

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

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

Options:

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

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

Notes:

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

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


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

Examples:

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

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

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

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

DATE

Syntax: **date [t]**

Function: Displays the current date.

Options:

t	Causes the time to appear with the date.

Notes:

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

  ```
  setime [ENTER]
  ```

  A possible screen display and your responses follows:

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

  date [ENTER]

  August 22, 1986

  date t [ENTER]

  August 22, 1986 14.20.20
  ```

Examples:

- To display the system date and time, type:

  ```
  date t [ENTER]
  ```

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

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

DCHECK

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

Function: Checks a disk's file structure.

Parameters:

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

Options:

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

Notes:

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

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

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

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

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

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

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

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

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

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

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

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

- DCHECK builds its disk allocation map in a file called *pathlist*/DCHECK*pp*0, where *pathlist* is specified by the -w option and *pp* is the process number in hexadecimal. Each bit in this bitmap file corresponds to a cluster of sectors on the disk. If you use the -p option, DCHECK creates a second bitmap file (*pathlist2*/DCHECKpp1) that has a bit set for each cluster DCHECK finds as "previously allocated" or "in file structure but not in allocation map." DCHECK then makes another pass through the directory structure to determine the pathlists for these questionable clusters. You can save the bitmap work files by specifying the -m option on the command line.

- For best results, DCHECK should have exclusive access to the disk being checked. Otherwise, the command might be fooled by a change in the disk allocation map while DCHECK is building a bitmap file. DCHECK cannot process disks with more than 39 levels of directories.

- -p causes DCHECK to make a second pass through the file structure and print pathlists for clusters that are not in the allocation map but are allocated or existing in a file structure.

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

Examples:

- The following two examples demonstrate DCHECK sessions:

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

A sample screen display might be:

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

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

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

A sample screen display might be:

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

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

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

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

DEINIZ

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

Function: Deinitializes and detaches a device.

Parameter:

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

Notes:

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

Example:

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

```
deiniz w1 ENTER
```

DEL

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

Function: Deletes the file(s) specified.

Parameter:

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

Option:

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

Notes:

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

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

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

```
dir /d1 [ENTER]

directory of /d1 14.29.46
myfile     newfile

del newfile [ENTER]
dir /d1 [ENTER]

directory of /d1 14.30.37
myfile
```

Examples:

- To delete files named Text_program and Test_program, type:

```
del text_program test_program ENTER
```

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

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

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

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

DELDIR

Syntax: **deldir *dirname***

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

Parameter:

dirname	The pathlist to the directory you want to delete.

Notes:

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

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

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

  Pressing [D] starts the deletion process.

  Pressing [Q] cancels the command.

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

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

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

DIR

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

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

Parameters:

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

Options:

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

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

Examples:

- To display the current data directory, type:

  `dir` ENTER

- To display the current execution directory, type:

  `dir x` ENTER

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

  `dir x e` ENTER

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

```
dir .. ENTER
```

- To display a directory named NEWSTUFF, type:

```
dir newstuff ENTER
```

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

```
dir e ENTER
```

The screen might display:

```
Directory of . 16:50:12

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

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

```
dir ENTER
```

The screen might display:

```
Directory of . 16:50:37

OS9Boot     CMDS     SYS     startup
DEFS
```

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

```
dir e [ENTER]

Directory of  . 16:52:04

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

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

```
dir [ENTER]


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

DISPLAY

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

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

Parameters:

hex	A list of one or more hexadecimal numbers.

Notes:

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

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

Examples:

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

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

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

```
display 07 [ENTER]
```

DSAVE

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

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

Parameters:

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

Options:

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

Notes:

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

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

Examples:

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

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

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

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

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

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

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

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

ECHO

Syntax: **echo** ***text***

Function: Echoes text to the screen.

Parameters:

text	The character or characters you type.

Notes:

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

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

Examples:

- To display a message on the screen, type:

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

- To echo text to the console, type:

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

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

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

ERROR

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

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

Parameters:

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

Notes:

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

Example:

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

```
error 215 216 [ENTER]
```

The screen displays:

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

EX

Syntax: **ex *filename***

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

Parameters:

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

Notes:

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

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

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

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

Example:

- To run BASIC09 without a resident shell, type:

```
ex basic09 ENTER
```

FORMAT

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

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

Parameters:

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

Options:

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

Notes:

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

  ```
  tmode -pause ENTER
  ```

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

  When formatting finishes, type:

  ```
  tmode pause ENTER
  ```

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

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

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

Examples:

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

```
format /d1 [ENTER]
```

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

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

- To format hard Disk /H0, type:

```
tmode -pause [ENTER]
format /h0 [ENTER]
tmode pause [ENTER]
```

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

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

FREE

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

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

Parameters:

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

Notes:

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

Examples:

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

  ```
  free [ENTER]
  ```

  The screen is similar to this:

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

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

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

A sample screen display is:

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

HELP

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

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

Parameters:

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

Notes:

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

Examples:

- To obtain help for the BACKUP command, type:

```
help backup [ENTER]
```

  The screen displays:

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

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

```
help me [ENTER]

me: no help available
```

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

```
help help [ENTER]
```

  The screen displays:

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

IDENT

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

Function: Displays header information for memory modules.

Parameters:

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

Options:

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

Notes:

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

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

Examples:

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

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

The screen might display:

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

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

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

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

The display might include:

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

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

INIZ

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

Function: Initializes the specified device driver.

Parameters:

devname The name of one or more devices to initialize.

Notes:

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

Examples:

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

```
iniz p t2 ENTER
```

KILL

Syntax: **kill *procID***

Function: Terminates the process specified by *procID*.

Parameters:

procID	The ID number of the process to kill.

Notes:

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

Examples:

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

```
kill 5 ENTER
```

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

```
procs [ENTER]

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

kill 3 [ENTER]
procs [ENTER]

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

LINK

Syntax: **link *modname***

Function: Locks a previously loaded module into memory.

Parameters:

modname	The name of the memory module to link.

Notes:

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

Examples:

- To lock the Edit module into memory, type:

```
link edit ENTER
```

LIST

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

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

Parameters:

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

Notes:

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

Examples:

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

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

  The ampersand makes the printing job a concurrently executed task.

- To list three files to the screen, type:

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

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

```
list /term >/p [ENTER]
```

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

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

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

list animals ENTER
```

The screen displays:

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

LOAD

Syntax: **load *pathlist***

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

Parameters:

pathlist	Specifies the module to load.

Notes:

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

Examples:

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

```
mdir [ENTER]
```

The screen display is similar to the following:

```
   Module Directory at 12:49:52
REL         Boot        OS9p1       OS9p2       Init
IOMan       RBF         CC3Disk     D0          D1
DD          SCF         CC3IO       VDGInt      GrfInt
TERM        W           W1          W2          W3
W4          W5          W6          W7          ACIAPAK
T2          PRINTER     P           PipeMan     Piper
Pipe        Clock       CC3Go       CC3HDisk    H0
Shell       Copy        Date        DeIniz      Del
Dir         Display     Echo        Iniz        Link
List        Load        MDir        Merge       Mfree
Procs       Rename      Setime      Tmode       Unlink
Basic09     GrfDrv

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

The screen displays:

```
   Module Directory at 12:51:12

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

MAKDIR

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

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

Parameters:

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

Notes:

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

Examples:

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

```
makdir /d1/STEVE/PROJECT [ENTER]
```

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

```
makdir DATAFILES [ENTER]
```

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

```
makdir ../SAVEFILES [ENTER]
```

MDIR

Syntax: **mdir [e]**

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

Options:

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

Notes:

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

Examples:

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

```
mdir e ENTER
```

The 80-column display of MDIR e is:

```
   Module Directory at 03:03:53

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

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

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

- The 32-column display of MDIR is:

```
      Module Directory at 03:06:49

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

```
1 6594   1AA 11   1 .     1 CC3Go
6    0   5F2 11   1 r     3 Shell
6  5F2   2DC 11   1 r     0 Copy
6  8CE    FD 11   1 r     0 Date
6  9CB    76 11   1 r     0 DEIniz
6  A41    A5 11   1 r     0 Del
6  AE6   365 11   1 r     0 Dir
6  E4B    84 11   1 r     0 Display
6  ECF    22 11   1 r     0 Echo
6  EF1    6A 11   1 r     0 Iniz
6  F5B    2C 11   1 r     0 Link
6  F87    4F 11   1 r     0 List
6  FD6    24 11   1 r     0 Load
6  FFA   2F1 11   1 r     1 MDir
6 12EB    68 11   1 r     0 Merge
6 1353   1EB 11   1 r     0 Mfree
6 153E   319 11   1 r     0 Procs
6 1857   11D 11   1 r     0 Rename
6 1974   118 11   1 r     0 Setime
6 1A8C   301 11   1 r     0 Tmode
6 1D8D    2D 11   1 R     0 Unlink
```

MERGE

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

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

Parameters:

filename Specifies the files to combine.

Notes:

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

Examples:

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

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

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

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

MFREE

Syntax: mfree

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

Notes:

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

Examples:

- Type this command:

  ```
  mfree [ENTER]
  ```

  The screen shows a display similar to this:

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

MODPATCH

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

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

Parameters:

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

Options:

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

Notes:

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

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

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

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

Examples:

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

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

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

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

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

```
modpatch w180 [ENTER]
```

Next, initialize W1:

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

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

MONTYPE

Syntax: **montype** *type*

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

Parameters:

Parameters:

type A single letter indicating the monitor type:

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

Notes:

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

Example:

To set your system for an RGB monitor, type:

```
montype r ENTER
```

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

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

Next, append the file to Startup. Type:

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

Delete the temp file:

```
del temp [ENTER]
```

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

```
rename Startup Startup.old
```

Then rename Startup.new:

```
rename Startup.new Startup
```

OS9GEN

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

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

Parameters:

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

Options:

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

Notes:

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

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

PROCS

Syntax: **procs [e]**

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

Options:

e	Causes PROCS to display the processes of all users.

Notes:

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

Examples:

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

```
procs e [ENTER]
```

Following is a possible 32-column display of PROCS:

```
Id  PId  User#  Pty  Age  Sta
 Sigl  Mem    StPtr   Primary
============================
  2   1      0   128 128 $80
    0     3    $78E2   Shell
  3   6      0   128 128 $80
    0    16    $74B2   Basic09
  4   2      0   128 128 $80
    0     6    $05F3   Procs
  5   0      0   128 128 $80
    0     3    $6FB2   Shell
  6   0      0   128 129 $80
    0     3    $68E2   Shell
```

Following is a possible 80-column display of PROCS:

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

PWD
PXD

Syntax: **pwd**
pxd

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

Notes:

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

Examples:

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

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

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

```
pwd [ENTER]
```

  The screen displays the data directory path:

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

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

```
chd .. [ENTER]
pwd [ENTER]

/D1/STEVE/TEXTFILES
```

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

```
chd .. [ENTER]
pwd [ENTER]

/D1/STEVE
```

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

```
pxd [ENTER]

/D0/CMDS
```

RENAME

Syntax: **rename *pathlist filename***

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

Parameters:

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

Notes:

- You must have write permission for the file.

Examples:

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

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

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

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

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

```
dir [ENTER]
```

The screen displays:

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

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

The screen now shows:

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

SETIME

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

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

Parameters:

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

Options:

Specifying seconds in the new time entry is optional.

Notes:

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

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

Examples:

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

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

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

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

SETPR

Syntax: **setpr *procID number***

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

Parameters:

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

Notes:

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

Examples:

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

```
setpr 8 250 ENTER
```

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

```
procs [ENTER]
```

Following is a sample screen display:

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

setpr 3 255 [ENTER]
procs [ENTER]

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

SHELL

Syntax: **shell *arglist***

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

Parameters:

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

Notes:

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

  ```
  shell dir e [ENTER]
  ```

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

  ```
  dir e [ENTER]
  ```

The following two commands also have identical effects:

```
shell x [ENTER]

x [ENTER]
```

- The shell command separators are:

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

- The Shell command modifiers are:

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

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

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

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

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

TMODE

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

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

Common uses include changing baud rates and control key definitions.

Parameters:

pathnum — One of the standard path numbers:

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

paramlist — One of the following options.

Options:

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

-upc — Displays both upper- and lowercase characters.

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

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

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

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

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

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

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

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

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

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

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

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

- 000 = no parity
- 101 = MARK parity transmitted, no checking
- 111 = SPACE parity transmitted, no checking
- 011 = even parity (available only with the external ACIA pak and Modpak devices)
- 001 = odd parity (available only with the external ACIA pak and Modpak devices)

Bit 4 selects auto-answer modem support features.

1 = on

0 = off

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

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

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

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

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

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

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

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

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

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

Notes:

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

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

Examples:

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

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

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

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

TUNEPORT

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

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

Parameters:

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

Options:

-s=	Sets a new delay loop value.

Examples:

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

  ```
  tuneport /p ENTER
  ```

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

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

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

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

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

UNLINK

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

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

Parameters:

modname One or more modules you want to unlink.

Options:

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

Notes:

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

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

Examples:

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

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

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

```
mdir [ENTER]
```

A possible screen display is:

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


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

The new screen display is:

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

WCREATE

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

Function: Initializes and creates a window.

Parameters:

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

Options:

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

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

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

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

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

-? Produces a help message for the command.

Examples:

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

```
wcreate /w1 -s=2 0 0 80 24 7 4 1 ENTER
```

- To create two windows (/w2 and /w3) on a 640 x 192 graphics screen in which /w2 is the upper left of the display and /w3 is the right half of the display, first use build to create an input file:

```
build wfile ENTER
? /w2 -s=07 0 0 40 12 7 4 1 ENTER
? /w3 40 0 40 24 4 7 ENTER
? ENTER
```

Then, create the windows using Wfile as input:

```
wcreate -z < wfile ENTER
```

- You can use the -z option to create windows in your system startup file. For example, the following startup file sets up several windows, along with the usual SETIME.

```
* lock the shell in memory and set the time
  link shell
setime < /1

* create the new windows
wcreate -z
* set up an 80-column full window for /w1
/w1 -s=2 0 0 80 24 7 4 1
* set up a 40 column full window for /w2
/w2 -s=1 0 0 40 24 7 4 1
* set up /w3 and /w4 as halves of a
*640 x 192 display
/w3 -s=7 0 0 40 24 7 4 1
/w4 40 0 40 24 4 7
* the following blank line terminates input
* from wcreate

* get the graphics fonts loaded
merge sys/stdfonts > /w1
```

Now, when the system boots, it has four windows defined, besides TERM. As shown, you can use an asterisk as the first character on a line in order to allow comments in the file.

XMODE

Syntax: **xmode** ***devname*** **[*paramlist*]**

Function: Displays or changes the initialization parameters of any SCF-type device such as the video display, printer, RS-232 port, and others. XMODE automatically adjusts its output for 32- or 80-column displays.

Common uses include changing baud rates and control key definitions.

Parameters:

pathnum	The device name to change, such as /term, /w7, /t2, and so on.
paramlist	One of the following options.

Options:

upc	Displays uppercase characters only. Lowercase characters automatically convert to uppercase.
-upc	Displays both upper- and lowercase characters.
bsb	Causes a backspace to erase characters. Backspace characters echo as a backspace-space-backspace sequence. This setting is the system default.
-bsb	Causes backspace not to erase. Only a single backspace echoes.
bsl	Enables *backspace over a line*. Deletes lines by sending backspace-space-backspace sequences to erase a line (for video terminals). This setting is the system default.
-bsl	Disables *backspace over a line*. To delete a line, you must print a *new line* sequence (for hard-copy terminals).

echo	Input characters *echo* on the terminal. This setting is the system default.
-echo	Turns off the echo default.
lf	Turns on the auto line feed function. Line feeds automatically echo to the terminal on input, and they output carriage returns. The auto line feed setting is the system default.
-lf	Turns off the auto line feed default.
null = *n*	Sets the null count—the number of null ($00) characters transmitted after carriage returns for the return delay. The value *n* is in decimal. The default is 0.
pause	Turns on the screen pause. This setting suspends output when the screen fills. See the *pag* parameter for a definition of screen size. Resume output by pressing the space bar. This setting is the system default.
-pause	Turns off the screen pause mode.
pag = *n*	Sets the length of the video display page to *n* (decimal) lines. This setting affects the *pause* mode.
bsp = *h*	Sets the backspace character for input. The value *h* is in hexadecimal. The default is 08.
del = *h*	Sets the delete line character for input. The value *h* is in hexadecimal. The default is 18.
eor = *h*	Sets the end-of-record (carriage return) character for input. This setting requires a value in hexadecimal. The default is 0D.
eof = *h*	Sets the end-of-file character for input. The value *h* is in hexadecimal. The default is 1B.
reprint = *h*	Sets the reprint line character. The value *h* is in hexadecimal.
dup = *h*	Sets the character to duplicate the last input line. The value *h* is in hexadecimal. The default is 01.

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

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

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

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

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

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

000 = no parity

101 = MARK parity transmitted, no checking

111 = SPACE parity transmitted, no checking

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

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

Bit 4 selects auto-answer modem support features.

1 = on

0 = off

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

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

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

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

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

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

Bits 0-3 determine the baud rate
Bit 4 is reserved for future use
Bits 5-6 determine the word length:
  00 = 8 bits
  01 = 7 bits
Bit 7 determines the number of stop bits:
  0 = 1 stop bit
  1 = 2 stop bits.

See "Technical Information for the RS232 Port" in Chapter 5 for further information.

xon=*h* Sets the character to be used as a signal for resuming transmission of data after an xoff signal is received. Default is 0 (not active).

xoff=*h* Sets the character to be used for stopping data transmission. Default is 0 (not active).

Notes:

- XMODE is similar to TMODE, but there are differences. TMODE operates only on open paths, so its effect is temporary. XMODE updates the device descriptor. Its change persists as long as the computer is running, even if you or the system repeatedly open and close the paths to the device.
- If you use XMODE to change parameters and the COBBLER program to make a new system diskette or to remake the boot tracks on the current system diskette, the process permanently changes the parameters on the new system diskette.
- XMODE requires that you specify a device name. If you do not specify parameters, XMODE displays the present value for each parameter. You can use any number of parameters, separating them with spaces or commas.

Examples:

- The following command sets the term (video) for upper- and lowercase, the null count to 4, the backspace character value to 1F hexadecimal, and turns on the screen pause function.

```
xmode /term -upc null=4 bse=1F pause ENTER
```

Macro Text Editor

Overview

The OS-9 Macro Text Editor is a powerful, easy-to-learn text-preparation system. Use it to prepare text for letters and documents or text to be used by other OS-9 programs, such as the assembler and high-level languages. The text editor includes the following features:

- Compact size
- Capability of having multiple read and write files open at the same time
- All OS-9 commands usable inside the text editor
- Adjustable workspace size
- Repeatable command sequences
- Edit macros (special utility functions)
- Multiple text buffers
- Powerful commands

The Macro Text Editor is about 5 kilobytes in size and requires at least 2K bytes of free RAM to run.

Text Buffers

As you enter text, the editor places it in a temporary storage area called a text buffer. A text buffer acts as a scratch pad for saving text that you can manipulate with various edit commands. The Macro Text Editor can use multiple text buffers, one at a time.

A buffer in use is called the *edit buffer*. Edit also has another default buffer called the *secondary buffer*. As well, you can create additional buffers up to the capacity of your computer's memory.

Edit Pointers

The Macro Text Editor has an edit *pointer* that identifies your position in the buffer, in a manner similar to holding your place in a book with your finger.

The pointer is invisible to you, but Edit commands can reposition it and display the text to which it points. Each buffer has its own edit pointer, and you can move from buffer to buffer without losing your place in any of them.

Entering Commands

The Macro Text Editor is interactive. This means you and the editor carry on a two-way conversation. You issue a command, and the editor carries out the command and displays the result. When you are through making changes, you can save your edited file, then press [Q] [ENTER] to quit editing.

When the editor displays `E:` on the screen, it is waiting for you to type a command. You type a line that includes one or more commands, then press [ENTER]. Edit carries out the commands and again displays `E:`.

If you enter more than one command on a line, separate the commands with a space. If, however, a space is the first character on a line, the editor considers the space to be an insert command and not a separator.

Correct a typing error by backspacing over it or by deleting the entire line. Note, you cannot correct a line after pressing [ENTER].

Control Keys

You can use the same special control keys with Edit that you use with OS-9. See Appendix D for a complete listing of these keys. Following is a list of some of the control keys that are especially useful with Edit:

Control Key(s)	Function
[CTRL] [A]	Repeats the previous input line.
[CTRL] [C]	Terminates the editor and returns to command entry mode.
[CTRL] [D]	Displays the current input on the next line.
[CTRL] [H] or [←]	Backspaces and erases the previous character.

Control Key(s)	Function
[Q] [ENTER]	Interrupts the editor and returns to command entry mode.
[CTRL] [W]	Temporarily halts the data output to your terminal so that you can read the screen before the data scrolls off. Output resumes when you press any other key.
[CTRL] [X] or [SHIFT] [←]	Deletes the line.
[CTRL] [BREAK]	Terminates the editor, and returns to command entry mode.

Command Parameters

There are two types of edit parameters, "numeric" and "string."

Numeric Parameters. Numeric parameters specify an amount, such as the number of times to repeat a command or the number of lines affected by a command. If you do not specify a numeric parameter, the editor uses the default value of one. Specify all other numeric parameters in one of the following ways.

- Enter a positive decimal integer in the range 0 to 65,535. For example:

  0
  10
  5250
  65532
  31

- Enter an asterisk (*) as a shorthand for *all* (all the way to the beginning, all the way to the end, all of the lines, and so on). To the editor, an asterisk means infinity. Use the asterisk to specify all remaining lines, all characters, or repeat forever.
- Use a numeric variable. (See "Parameter Passing" later in this chapter.)

String Parameters. String parameters specify a single character, group of characters, word, or phrase. Specify string parameters in either of the following ways.

- Enclose the group of characters with delimiters (two matching characters). You can use any characters, but they must match. If one string immediately follows another, separate the two with a single delimiter that matches the others. For example:

```
"string of characters"
/STRING/
: my name is Larry :
"first string"second string"
/string 1/ string 2/
```

- Use a string variable. (See "Using Macros" later in this chapter.)

Syntax Notation

Syntax descriptions indicate what to enter and the order in which to do it. The command name is first; type it exactly as shown. Follow the command name with the correct parameters. Enter each as it is described in the section on parameters.

The syntax descriptions for each command use the following notations:

n = numeric parameter
str = string parameter
␣ = space character. When you see ␣, press the space bar.
text = one or more characters terminated by pressing [ENTER]

Getting Started

From the OS-9 prompt, start Edit by typing:

```
edit [ENTER]
```

Enter a command when the screen shows `E:`.

You can quit Edit at any time by pressing [Q] [ENTER]. The Q command terminates the editor and returns you to the OS-9 Shell, which responds with the `OS9:` prompt.

Following is a list of ways you can start the editor, including the effect of each. The examples call a file that already exists *oldfile*. They call a file to be created *newfile*.

EDIT

OS-9 loads the editor and starts it. The command does not establish an initial read or write files, but you can perform text file operations by opening files after the editor is started.

EDIT *newfile*

OS-9 loads the editor and starts it, creating the file called *newfile*. *Newfile* is the initial write file. There is no initial read file. However, you can open files to read later.

EDIT *oldfile*

OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called SCRATCH as the initial write file. When you end the edit session, OS-9 deletes *oldfile* and renames SCRATCH to *oldfile*. This gives the appearance of *oldfile* being updated.

Note: The two OS-9 utilities DEL and RENAME must be present on your system if you wish to start the editor in this manner.

EDIT *oldfile newfile*

OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates *newfile*—the initial write file. The terms *oldfile* and *newfile* refer to any properly constructed OS-9 pathlist.

Edit Commands

Displaying Text

L*n*

Lists (displays) the next *n* lines, starting at the current position of the edit pointer. The edit pointer position does not change.

```
l [ENTER]
```

displays the current line. If the edit pointer is not at the beginning of the line, only the portion of the line to the right of the edit pointer shows on the screen.

```
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 the *n* lines that precede the edit pointer. The position of the edit pointer does not change. For example:

```
x [ENTER]
```

displays any text on the current line that precedes the edit pointer. If the edit pointer is at the beginning of the line, the command displays nothing.

```
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

[CTRL] [7] or [↑] on an external terminal

Moves the edit pointer to the beginning (first character) of the text buffer. The screen shows the up arrow when you hold down [CTRL] and press [7]. For example,

[CTRL] [7] [ENTER]

moves the edit pointer to the beginning of the buffer.

/

Moves the edit pointer to the end (last character) of the buffer. For example,

/ [ENTER]

moves the edit pointer past the end of the buffer.

[ENTER]

Moves the edit pointer to the beginning of the next line and displays it. Use this command to go through text one line at a time. You can look at each line, correct any mistakes, and then move to the next line.

+n Moves the edit pointer either to the end of the line or forward *n* lines and displays the line. Entering a value of zero moves the edit pointer to the end of the current line. For example:

```
+0 ENTER
```

Entering a value other than zero moves the pointer forward *n* lines and displays the line. For example,

```
+ ENTER
```

moves the pointer to the next line and displays the line. This command performs the same function as ENTER.

```
+10 ENTER
```

moves the pointer ahead 10 lines and displays the line.

```
+* ENTER
```

moves the edit pointer to the end of the buffer.

-n Moves the edit pointer either to the beginning of the line or backward *n* lines. For example:

```
-0 ENTER
```

moves the edit pointer to the beginning of the line and displays the line. Entering a value other than zero moves the edit pointer back *n* lines. For example,

```
- ENTER
```

moves the edit pointer back one line and displays the line.

```
-5 ENTER
```

moves the edit pointer back five lines and displays the line.

```
-* ENTER
```

moves the edit pointer to the beginning (top) of the buffer and displays the first line.

>*n*

Moves the edit pointer to the right *n* characters. Use this command to move the edit pointer to some position in the line other than the first character. For example,

> [ENTER]

moves the edit pointer to the right one character.

>25 [ENTER]

moves the edit pointer to the right 25 characters.

>* [ENTER]

moves the edit pointer to the end of the buffer.

<*n*

Moves the edit pointer to the left *n* characters. Use this command to move the edit pointer to some position in a line other than the first character. For example:

< [ENTER]

moves the edit pointer to the left one character.

<10 [ENTER]

moves the edit pointer to the left 10 characters.

<* [ENTER]

moves the edit pointer to the beginning of the buffer.

Inserting and Deleting Lines

␣*text* Preceding text lines with a space inserts the text as a new line ahead of the edit pointer. The position of the edit pointer does not change. For example,

```
␣Insert this line [ENTER]
```

inserts the line.

```
␣Line one [ENTER]
␣Line two [ENTER]
␣Line three [ENTER]
```

inserts three lines.

I*n str* Inserts a line of *n* copies of the specified string immediately before the position of the edit pointer. The position of the edit pointer does not change. For example,

```
i40/*/ [ENTER]
```

inserts a line containing 40 asterisks. You can also use the "I" command to insert a line containing a single copy of the string. This function is important when you want to use a macro to insert lines, since the space bar cannot be used within a macro. For example,

```
i"Line to insert" [ENTER]
```

inserts the line.

D*n*

Deletes (removes) *n* lines from the edit buffer, starting with the current line. This command displays the lines to be deleted. For example:

d ENTER

deletes the current line, regardless of the position of the edit pointer, and displays it.

d4 ENTER

deletes the current line and the next three lines.

d* ENTER

deletes everything from the current line to the end of the buffer.

K*n*

Kills (deletes) *n* characters, starting at the current position of the edit pointer. This command displays all deleted characters. For example,

k ENTER

deletes the character at the edit pointer.

k4 ENTER

deletes the character at the current position of the edit pointer and the next three characters.

k* ENTER

deletes everything from the current position of the edit pointer to the end of the buffer.

E*n str*

Extends *n* lines by adding a string to the end of each line. E extends a line, displays it, and then moves the pointer past it. For example,

```
e/this is a comment/ [ENTER]
```

adds the string "this is a comment" to the end of the current line and moves the edit pointer to the next line.

```
e3/xx [ENTER]
```

adds the string xx to the end of the current line and the next two lines. It moves the pointer past these lines.

U

Unextends (deletes) the remainder of a line from the current position of the pointer. Use U to remove extensions, such as comments, from a line. For example,

```
u [ENTER]
```

deletes all the characters from the current position of the pointer up to the end of the current line.

For some practice in using the commands that display text, manipulate the edit pointer, and insert and delete lines, turn to Sample Session 1 in this chapter.

Searching and Substituting

S*n string*

Searches for the next *n* occurrences of *string*. When Edit finds an occurrence, it displays the line and moves the edit pointer to the line. If Edit does not find the string or if all the occurrences have been found, the edit pointer does not move. For example,

```
s/my string/ ENTER
```

searches for the next occurrence of "my string".

```
s3"strung out" ENTER
```

searches for the next three occurrences of "strung out".

```
s*/seek and find/ ENTER
```

searches for all occurrences of "seek and find" between the edit pointer and the end of the text.

C*n string1 string2*

Changes the next *n* occurrences of *string1* to *string2*. When Edit finds *string1*, it moves the edit pointer past it and changes *string1* to *string2*, then it displays the updated line. If it does not find *string1* it displays "NOT FOUND." If all the occurrences have been found, the edit pointer does not move. For example,

```
c/this/that/ ENTER
```

changes the next occurrence of "this" to "that".

```
c2/in/out/ ENTER
```

changes the next two occurrences of "in" to "out".

```
c*!seek and find!sought and
found! ENTER
```

changes all occurrences of "seek and find" that are between the edit pointer and the end of text to "sought and found".

A*n* Sets the SEARCH/CHANGE anchor to Column *n*. To find a string that begins in a specific column, set the anchor to the column position before using the search command to find it. If you do not include a value for *n*, Edit assumes Column 1. For example:

```
a ENTER
```

finds a string only if it begins in Column 1.

```
a20 ENTER
```

finds a string only if it begins in Column 20. If you use the A command to set the anchor, this setting remains in effect only for the current command line. After Edit executes the command, the anchor automatically returns to its normal value of zero.

For some practice in using the commands that search and substitute, turn to Sample Session 2 in this chapter.

Miscellaneous Commands

T*n* Tabs (moves) the edit pointer to Column *n* of the current line. If *n* exceeds the line length, this command extends the line with spaces. For example,

```
t ENTER
```

moves the edit pointer to Column 1 of the current line.

```
t5 ENTER
```

moves the edit pointer to Column 5 of the current line.

.SHELL *command line*

Lets you use any OS-9 command from within the editor. The remainder of the command line following .SHELL passes to the OS-9 Shell for execution. For example,

```
.shell dir /d1 ENTER
```

calls the OS-9 Shell to display the directory of D1.

```
.shell basic09 ENTER
```

starts BASIC09.

```
.shell edit oldfile newfile ENTER
```

restarts the editor.

M*n*

Adjusts the amount of memory available for buffers and macros. If the workspace is full and the editor does not allow you to enter more text, increase the workspace size. If you need only a small amount of the available workspace, decrease the workspace size so that other OS-9 programs can use the memory. For example,

```
m5000 ENTER
```

sets the workspace size to 5000 bytes.

```
m10000 ENTER
```

sets the workspace size to 10000 bytes.

Before leaving Edit, you can increase the workspace. This decreases the time the editor takes to copy the input file to the output file, because the editor can read and write more data at one time. Edit changes memory in 256-byte pages. For the M command to have any effect, a new workspace size must differ from the current size by at least 256 bytes. The M command does not let you deallocate any workspace that Edit needs for buffers or macros.

.SIZE

Displays the size of the workspace and the amount that has been used. For example:

```
.size
  521     15328
```

521 is the amount of workspace Edit uses for buffers and macros. 15328 is the amount of available memory.

Q

Ends editing and returns to the OS-9 Shell. If you specified files when you started, Edit writes the text in Buffer 1 to the initial write file (specified when you start Edit). Next it copies the remainder of the initial input file (specified when you start Edit) to the initial write file. The editor then terminates, and control returns to the OS-9 Shell.

V*mode*

Turns the verify mode on or off. Edit always starts with the verify mode on. Therefore, the editor displays the results of all the commands for which verify is appropriate. If you do not want to see the results of commands, turn off the verify mode by specifying 0 (zero) for *mode*. To turn verify back on, specify any non-zero number. For example,

```
v0 [ENTER]
```

turns off the verify mode.

```
v2 [ENTER]
```

turns on the verify mode.

```
v13 [ENTER]
```

turns on the verify mode.

If the verify mode is on when you switch to a macro, it remains on. If you turn off verify while in the macro, it is restored when you return to the editor.

Manipulating Multiple Buffers

.DIR Displays the directory of the editor's buffers and macros. For example:

```
BUFFERS:
$       0 (secondary buffer)
*       1 (primary buffer)
        5 (another buffer)

MACROS:

    MYMACRO
    LIST
    COPY
```

B*n* Makes buffer *n* the primary buffer. When you switch from one buffer to another, the old one becomes the secondary buffer, and the new one becomes the primary buffer. For example,

```
b5 [ENTER]
```

makes Buffer 5 the primary buffer. If Buffer 5 does not exist, Edit creates it.

P*n* Puts (moves) *n* lines into the secondary buffer. This command removes the lines from the primary buffer, starting at the position of the edit pointer, and inserts them into the secondary buffer before the current position of the edit pointer. It displays the text that is moved. For example,

```
p [ENTER]
```

moves one line to the secondary buffer.

```
p5 [ENTER]
```

moves five lines to the secondary buffer.

```
p* [ENTER]
```

moves all lines that are between the current position of the edit pointer and the end of text to the secondary buffer.

G*n* Gets (moves) *n* lines from the secondary buffer. This command takes the lines from the top of the secondary buffer and inserts them into the primary buffer before the current position of the edit pointer. Edit then displays the moved lines. When used with the P command, G moves text from one place to another. For example,

```
g ENTER
```

gets one line from the secondary buffer.

```
g5 ENTER
```

gets five lines from the secondary buffer.

```
g* ENTER
```

gets all lines from the secondary buffer.

For some practice in using miscellaneous commands and the commands that manipulate multiple buffers, turn to Sample Session 3 in this chapter.

Text File Operations

This section of the manual describes the group of commands related to reading and writing OS-9 text files.

.NEW Gets new text. Use .NEW when editing a file that is too large to fit into the editor's workspace. .NEW writes out all lines that precede the current line, then appends an equal amount of new text to the end of the buffer.

.NEW always writes text to the initial output file (created when you start the editor) and always reads text from the initial input file (specified when you start the editor).

If you have finished editing the text currently in the buffer, you can "flush" this text and fill the buffer with new text by moving the edit pointer to the bottom of the buffer and then using the .NEW command. For example:

```
/.new ENTER
```

If you wish to retain part of the text that is already in the buffer, move the edit pointer to the first line you wish to retain and then type `.new`. This command "flushes" all lines that precede the edit pointer. It then tries to read in new text that is the same size as the portion flushed out.

.READ ***str*** Prepares an OS-9 text file for reading. *str* specifies the pathlist. For example.

```
.read "myfile" ENTER
```

closes the current input file and opens "myfile" for reading.

You can specify an empty pathlist. For example,

```
.read "" ENTER
```

closes the current input file and restores the initial input file (specified when you start the editor) for reading.

An open file remains attached to the primary buffer until you close the file. You can have more than one input file open at any time by using the .READ command to open them in different buffers.

To read these files, switch to the proper buffer, and then use the R command to read from that buffer's input file. To close a file, you must be in the same buffer where the file was opened.

.WRITE *str* Opens a new file for writing. The *string* specifies the pathlist for the file you wish to create. For example,

```
.write "newfile" ENTER
```

closes the current write file and creates one called "newfile". You can specify an empty pathlist. For example:

```
.write "" ENTER
```

closes the current write file and restores the initial write file (specified when you start the editor).

.WRITE attaches a new write file to the primary buffer that remains attached until you close the file. You can have more than one write file open by using .WRITE to open them in different buffers. To write these files, switch to the proper buffer. To close a file, you must be in the same buffer where the file was opened.

R*n* Reads (gets) *n* lines of text from the buffer's input file. It displays the lines and inserts them before the current position of the edit pointer. For example,

```
r ENTER
```

reads one line from the input file.

```
r10 ENTER
```

reads 10 lines from the input file.

```
r* ENTER
```

reads the remaining lines from the input file.

If a file contains no more text, the screen shows the `*END OF FILE*` message.

W*n* Writes *n* lines to the output file, starting with the current line. It displays all lines that are deleted from the buffer. For example,

```
w [ENTER]
```

writes the current line to the output file.

```
w5 [ENTER]
```

writes the current line and the next four lines to the output file.

```
w* [ENTER]
```

writes all lines from the current line to the end of the buffer to the output file.

For some practice in using the commands that read and write OS-9 text files, turn to Sample Session 4 in this chapter.

Conditionals and Command Series Repetition

When a command cannot be executed, the editor sets an internal flag, and the screen shows *FAIL*. For example, if you try to read from a file that has no more text, the editor sets the fail flag. A set fail flag means that the editor cannot execute any more commands until Edit encounters one of the following:

- The end of a command line typed from the keyboard.
- The end of the current loop. Any loops that are more deeply nested are skipped. (See the repeat command.)
- A colon (:) command. Since loops nested deeper than the current level are skipped, any occurrences of : that are in a more deeply nested loop are also skipped.

Following are the commands and conditions that set the fail flag:

<	Trying to move the edit pointer beyond the beginning of the edit buffer.
>	Trying to move the edit pointer beyond the + end of the buffer.
S,C	Not finding a string that was searched for.
G	No text left in the secondary buffer.
R	No text left in the read file.
P,W	No text left in the primary buffer.

If you specify an asterisk for the repeat count on these commands, Edit does not set the fail flag, because an asterisk usually means continue until there is nothing more to do. The following commands explicitly set the fail flag if some condition is not true.

.EOF	Tests for end-of-file. .EOF succeeds if there is no more text to read from a file. Otherwise, it sets the fail flag.
.NEOF	Tests for not end-of-file. .NEOF succeeds if there is text to read from the file. Otherwise, it sets the fail flag.
.EOB	Tests for end-of-buffer. .EOB succeeds if the edit pointer is at the end of the buffer. Otherwise, it sets the fail flag.
.NEOB	Tests for not end-of-buffer. .NEOB succeeds if the edit pointer is not at the end of the buffer. Otherwise, it sets the fail flag.
.EOL	Tests for end-of-line. This test succeeds if the edit pointer is at the end of the line. Otherwise, it sets the fail flag.
.NEOL	Tests for not end-of-line. .NEOL succeeds if the edit pointer is not at the end of the line. Otherwise, it sets the fail flag.
.ZERO *n*	Tests for zero value. .ZERO succeeds if *n* equals zero. Otherwise, it sets the fail flag.

.STAR *n* Tests for star (asterisk). .STAR succeeds if *n* equals 65,535 ("*"). Otherwise, it sets the fail flag.

.STR *str* Tests for string match. .STR succeeds if the characters at the current position of the edit pointer match the string. Otherwise, it sets the fail flag.

.NSTR *str* Tests for string mismatch. .NSTR succeeds if the characters at the current position of the edit pointer do not match the string. Otherwise, it sets the fail flag.

.S Exits and succeeds. This is an unconditional exit from the innermost loop or macro. The fail flag clears after the exit.

.F Exits and fails. This is an unconditional exit from the innermost loop or macro. The fail flag sets after the exit.

[*commands*]*n* Repeats the *commands n* times. Left and right brackets form a loop that repeats the enclosed *commands n* times. (The loop must be repeated at least once.) If you enter the loop command from the keyboard, it must all be on one line. If it is part of a macro, however, it can span several command lines. For example,

```
[ l ] 5 ENTER
```

repeats the L command five times.

Note: This is not the same as L5, which executes the L command only once and has 5 as its parameter.

[+]* Displays lines starting with the next line up to the end of the buffer and moves the edit pointer to the end of the buffer.

This command repeats until the operation reaches the end of the buffer. Then, when the command tries to move the edit pointer past the end of the buffer, Edit sets the fail flag, terminates the loop, then clears the fail flag.

: *commands* Executes the commands following the colon based on the state of the fail flag. For example:

FAIL FLAG CLEAR	Skips all commands that follow the colon (:) up to the end of the current loop or macro.
FAIL FLAG SET	Clears the fail flag, and executes the commands that follow the colon (:).

Below is a command line that deletes all lines that do not begin with the letter A.

```
CTRL 7 [ .neob [ .str"A" + : d ]
] * ENTER
```

↑ moves the edit pointer to the beginning of the buffer. The outer loop tests for the end of the buffer and terminates the loop when it is reached.

The inner loop tests for A at the beginning of the line. If there is an A, the + command is executed. Otherwise, it executes the D command.

Below is a command that searches the current line for "find it". If the command finds the text, it displays the line. Otherwise, the command line fails and the screen shows * FAIL *.

```
[ .eol v0 -0 v .f : .str"find it"
-0 .s : [>] ]* ENTER
```

.EOL V0 -0 V .F tests to determine if the edit pointer is at the end of the line. If it is, Edit turns off the verify mode to prevent -0 from displaying the line. Then it turns verify back on, and .F ends the loop.

If the edit pointer is not at the end of the line, the .STR command searches for "find it" at the current position of the edit pointer. If it is at the end of the line, Edit executes the -0 .S commands. This execution moves the edit pointer back to the beginning of the line, displays the line, and terminates the loop. Otherwise, the > command moves the edit pointer to the next position in the line.

The brackets prevent the command from failing and terminating the main loop if the end of the buffer is reached.

Edit Macros

Edit macros are commands you create to perform a specialized or complex task. For example, you can replace a frequently used series of commands with a single macro. First, save the series in a macro. Then each time you need it, type a period followed by the macro's name and parameters. The editor responds as if you had typed the series of commands.

Macros consist of two main parts, the header and the body. The header gives the macro a name and describes the type and order of its parameters. The body consists of any number of ordinary commands. (Except for a space character and [ENTER], you can use any command in a macro).

Note: Macros cannot create new macros.

To create a macro, first define it with the .MAC command. Then enter the header and body in the same manner as you enter text into an edit buffer. When you are satisfied with the macro, close its definition by pressing [Q] [ENTER]. This command returns you to the normal edit mode.

Macro Headers. A macro header must be the first line in each macro. It consists of a name, and a "variable list" that describes the macro's parameters, if there are any. The name consists of any number of consecutive letters and underline characters. Following are possible macro names:

```
del_all
trim_spaces
LIST
CHANGE_X_TO_Y
```

Although you can make a macro name any length, it is better to keep it short, because you must spell it the same way each time you use it. You can use upper- and lowercase letters or a mixture.

Using Macros. Like other commands, you can give parameters to macros so that they are able to work with different strings and with different numbers of items. Macros are unable to use parameters directly. Instead, Edit passes the parameters on to the commands that make up the macro.

To pass the macro's parameters to these commands, use the variable list in the macro header to tell each command which of the macro's parameters to use. Each variable in the variable list represents the value of the macro parameter in its corresponding position. Use the corresponding variable wherever the parameter's value is needed.

The two types of variables are numeric and string. A numeric variable is a variable name preceded by the # character. A string variable is a variable name preceded by a $ character. Variable names, like macro names, are composed of any number of consecutive letters and underline characters. Examples of numeric variables are:

```
#N
#ABC
#LONG_NUMBER_VARIABLE
```

Examples of string variables are:

```
$A
$B
$STR
$STR_A
$lower_case_variable_name
```

The function of the edit macro below is the same as that of the S command, to search for the next *n* occurrences of a string.

The first line of the macro is the macro header. It assigns the macro's name as SRCH. It also specifies that the macro needs one numeric parameter (#N) and one string parameter ($STR). The entire body of the macro is the second line. This example passes both of the macro's parameters to the S command, which does the actual searching.

```
SRCH #N $STR
S #N $STR
```

Here is an example of how to execute this macro:

```
.SRCH 15 "string" [ENTER]
```

In the next example, the order of the parameter is reversed. Therefore, when executing the macro, use the reverse order. The macro structure is:

```
SRCH $STR #N
S #N $STR
```

Specify the parameters for the "S" command in the proper order since it is only the "SRCH" macro that is changed. The following example shows how to execute this macro. The order of the parameters corresponds directly to the order of the variables in the variable list.

```
.SRCH "string" 15 [ENTER]
```

Macro Commands

Although macro editing has the same functions as text editing, the macro mode also includes some special commands. The macro commands you can use are as follows:

! *text* Places comments inside a macro. Ignores the remainder of the line following the ! command. This command lets you include, as part of a macro, a short description of what it does. Comments can help you remember the function of a macro. For example:

```
!
<^>! Move the pointer to the top of the
     buffer.
I*!  Display all lines of text.
!
```

In this example there are four comments. Two are empty, and two describe the commands that precede them.

.macro name Executes the macro specified by the name following the period (.). For example:

```
.mymacro [ENTER]
.list 0 [ENTER]
.trim " " [ENTER]
.merge " file_a " file b b" [ENTER]
```

.MAC *str* Creates a new macro or opens the definition of an existing one so that it can be edited. To create a new macro, specify an empty string. For example,

```
.mac"" [ENTER]
```

creates a new macro and puts you into the macro mode.

The screen shows M: instead of E: when the editor is in the macro mode. To edit a macro that already exists, specify the macro's name. For example,

```
.mac "mymacro" [ENTER]
```

opens the macro "MYMACRO" for editing.

When a macro is open, edit it, or enter its definition with the same commands you use in a text buffer. After you edit the macro, press [Q] [ENTER] to close its definition and return to the edit mode. The first line of the macro must begin with a name that is not already used in order to close the definition and return to Edit.

.SAVE *str1 str2*

Saves macros on an OS-9 file. *Str1* specifies a list of macros to be saved. Separate the macro names with spaces. *Str2* specifies the pathlist for the file on which you want to save the macros. For example:

```
.save "mymacro"myfile" [ENTER]
```

saves the macro "MYMACRO" on the file "MYFILE".

```
.save "maca macb macc"mfile" [ENTER]
```

saves the macros "MACA," "MACB," and "MACC" on the file "MFILE".

.SEARCH *n str*

Searches the text file buffer for the specified string. When a match is found, it stops and displays that line. The *n* option permits a search for the *nth* occurrence of a string match. This command is the same as S *n str*.

.LOAD *str* Loads macros from an OS-9 file. As each macro loads, Edit verifies that no other macro already exists with the same name. If one does, the macro with the duplicate name does not load, and Edit skips to the next macro on the file. Edit displays the names of all macros it loads. For example,

```
.load "macrofile" ENTER
```

loads the macros in the file called MACROFILE.

```
.load "myfile" ENTER
```

loads the macros in the file called MYFILE.

.DEL *str* Deletes the macro specified by the string. For example,

```
.del "mymacro" ENTER
```

deletes the macro called MYMACRO.

```
.del "list" ENTER
```

deletes the macro called LIST.

.DIR Displays the current edit buffer area. All edit buffers and macros currently in memory are displayed.

.CHANGE *n str1 str2* Changes the occurrence of *str1* to *str2*. The *n* option permits *n* occurrences of *str1* to be changed to *str2*.

Q Ends a macro edit session and returns you to the normal edit mode. For example:

```
Search_and_Delete #N $STR
!This example MACRO is used to
!check
!the string at the beginning of
!an #N number of lines. If the
!string matches, it will delete
!that line from the text buffer
!file.
!
!NOTE: The way the editor
!processes a MACRO causes it to
!see any parameters in the outer
!loop first. Thus, the #N
!parameter is processed before
!the STR parameter.
!
[^]             !Move to start of
                !edit buffer
[               !start of outer loop
.neob           !test for buffer end
[               !start of inner loop
.nstr $str      !test for not string
                !match
+               !go to next line if
                !no match
:               !if flag clear skip
                !next command
D               !delete line if flag
                !set
]               !end of inner loop
]#N             !end of outer loop
! End of Macro
```

For practice in using macro commands, turn to Sample Session 5 in this chapter.

Sample Session 1

Clear the buffer by deleting its contents.

You Type: [CTRL][7] D* [ENTER]
Screen Shows: ^D*

Insert three lines into the buffer. Begin each line with a space, which is the command for inserting text.

You Type:
```
 MY FIRST LINE [ENTER]
 MY SECOND LINE [ENTER]
 MY THIRD LINE [ENTER]
```
Screen Shows:
```
MY FIRST LINE
MY SECOND LINE
MY THIRD LINE
```

Move the edit pointer to the top of the text. The editor always considers the first character you type a command.

Note: [CTRL][7] always shows ^ on the screen. Typing -* also moves the edit pointer to the beginning of a buffer.

You Type: [CTRL][7] [ENTER]
Screen Shows: ^

List (display) the first line you inserted into the buffer.

You Type: L [ENTER]
Screen Shows:
```
L
 MY FIRST LINE
```

Display the first two lines you inserted into the buffer.

You Type: L2 [ENTER]
Screen Shows:
```
L2
 MY FIRST LINE
 MY SECOND LINE
```

Move to the next line and display it.

You Type: [ENTER]
Screen Shows: MY SECOND LINE

Move to the next line and display it.

You Type: [ENTER]
Screen Shows: MY THIRD LINE

Using L, display text beginning at the position of the edit pointer.

You Type: L [ENTER]
Screen Shows:
```
L
 MY THIRD LINE
```

Insert a line into the buffer.

Note: In the next sample you see that the insert comes before the current position of the edit pointer.

You Type: □INSERT A LINE [ENTER]
Screen Shows: INSERT A LINE

The following command line consists of more than one command. [CTRL][7] [ENTER] moves the edit pointer to the top of the text. L displays the text, and the asterisk (*) following L indicates that text is displayed through to the end of the buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 MY FIRST LINE
 MY SECOND LINE
 INSERT A LINE
 MY THIRD LINE
```

Show the position of the edit pointer.

You Type: L [ENTER]
Screen Shows:
```
L
 MY FIRST LINE
```

Move the edit pointer forward two lines and display the lines.

You Type: +2 [ENTER]
Screen Shows:
```
+2
 INSERT A LINE
```

Display all lines from the edit pointer to the end of the buffer.

You Type: L* [ENTER]
Screen Shows:
```
L*
 INSERT A LINE
 MY THIRD LINE
```

Move the edit pointer to the end of the buffer.

You Type: / [ENTER]
Screen Shows: /

Determine if the edit pointer is at the end of text. Since the screen shows no more lines, the edit pointer is at the end-of-text.

You Type: L* [ENTER]
Screen Shows: L*

Insert two more lines.

You Type:
```
□FIFTH LINE [ENTER]
□LAST LINE [ENTER]
```
Screen Shows:
```
FIFTH LINE
LAST LINE
```

Move the edit pointer back one line, and display the line.

You Type:
```
-2 [ENTER]
```
Screen Shows:
```
-2
 FIFTH LINE
```

Move the edit pointer back two lines, and display the line.

You Type:
```
-3 [ENTER]
```
Screen Shows:
```
-3
 MY SECOND LINE
```

Move the edit pointer three characters to the right and display the remainder of the line.

Note: You must put spaces between commands.

You Type:
```
>3 L [ENTER]
```
Screen Shows:
```
>3 L
 SECOND LINE
```

Display the characters that precede the edit pointer on the current line.

You Type:
```
X [ENTER]
```
Screen Shows:
```
X
 MY
```

Move the edit pointer to the end of the current line.

You Type:
```
+0 [ENTER]
```
Screen Shows:
```
+0
```

Determine if the edit pointer is at the end of the line. It is, since the screen shows no lines.

You Type:
```
L [ENTER]
```
Screen Shows:
```
L
```

Display the characters that precede the edit pointer on the current line.

You Type:
```
X [ENTER]
```
Screen Shows:
```
X
 MY SECOND LINE
```

Move the edit pointer back to the beginning of the current line.

You Type: -0 [ENTER]
Screen Shows:
```
-0
MY SECOND LINE
```

Determine if the edit pointer is at the beginning of the line. Since the screen shows no lines, the pointer is at the beginning.

You Type: X [ENTER]
Screen Shows:
```
X
```

Go to the beginning of the text.

You Type: [CTRL][7] [ENTER]
Screen Shows:
```
^
```

Insert a line of 14 asterisks.

You Type: I14"*" [ENTER]
Screen Shows:
```
I14"*"
 **************
```

Insert an empty line.

You Type: I"" [ENTER]
Screen Shows:
```
I""
```

Move to the top of the text, and display all lines in the buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 **************

 MY FIRST LINE
 MY SECOND LINE
 INSERT A LINE
 MY THIRD LINE
 FIFTH LINE
 LAST LINE
```

Move the edit pointer forward two lines.

You Type: +2 [ENTER]
Screen Shows:
```
+2
 MY FIRST LINE
```

Extend the line with XXX.

You Type: E" XXX" [ENTER]
Screen Shows:
```
E" XXX"
 MY FIRST LINE XXX
```

Display the current line.

Note: The previous E command moved the edit pointer to the next line.

You Type: L [ENTER]
Screen Shows:
```
L
 MY SECOND LINE
```

Extend three lines with YYY.

You Type: E3"␣YYY" [ENTER]
Screen Shows:
```
E3" YYY"
 MY SECOND LINE YYY
 INSERT A LINE YYY
 MY THIRD LINE YYY
```

Move back 2 lines.

You Type: -2 [ENTER]
Screen Shows:
```
-2
 INSERT A LINE YYY
```

Move the edit pointer to the end of the line and then move the edit pointer back four characters. Display the current line, starting at the edit pointer.

You Type: +0 <4 L [ENTER]
Screen Shows:
```
+0 <4 L
   YYY
```

Truncate the line at the current position of the edit pointer. This command removes the YYY extension.

You Type: U [ENTER]
Screen Shows:
```
U
 INSERT A LINE
```

Go to the top of the text and display the contents of the buffer.

You Type: [CTRL] [7] L* [ENTER]
Screen Shows:
```
^L*
 **************

 MY FIRST LINE XXX
 MY SECOND LINE YYY
 INSERT A LINE
 MY THIRD LINE YYY
 FIFTH LINE
 LAST LINE
```

Delete the current line and the next line.

You Type: D2 [ENTER]
Screen Shows:
```
D2
**************
```

Move the edit pointer forward two lines.

You Type: +2 [ENTER]
Screen Shows:
```
+2
 INSERT A LINE
```

Delete this line.

You Type: D [ENTER]
Screen Shows:
```
D
 INSERT A LINE
```

Display the current line.

You Type: L [ENTER]
Screen Shows:
```
L
  MY THIRD LINE YYY
```

Move the edit pointer to the right three characters and display the text.

You Type: >3 L [ENTER]
Screen Shows:
```
>3 L
 THIRD LINE YYY
```

Kill (delete) the 11 characters that constitute THIRD LINE.

You Type: K11 [ENTER]
Screen Shows:
```
K11
 THIRD LINE
```

Go to the beginning of the line and display it.

You Type: -Ø [ENTER]
Screen Shows:
```
-Ø
 MY YYY
```

Concatenate (combine) two lines. Move the edit pointer to the end of the line; delete the character at the end of the line; move the edit pointer back to the beginning of the lines. Display the line.

You Type: +Ø K -Ø [ENTER]
Screen Shows:
```
Ø K -Ø
 MY YYYFIFTH LINE
```

Separate the two lines by inserting an end-of-line character.

You Type: >6 I/ / [ENTER]
Screen Shows:
```
>6 I/ /
 MY YYY
```

Note: The end of line character is inserted before the current position of the edit pointer.

You Type: L [ENTER]

Screen Shows:
```
L
 FIFTH LINE
```

Sample Session 2

Clear the buffer by deleting its contents.

You Type: [CTRL] [7] D* [ENTER]

Insert lines.

You Type:
```
□ONE TWO THREE 1.0 [ENTER]
□ONE [ENTER]
□□TWO [ENTER]
□□□THREE [ENTER]
□ONE TWO THREE 2.0 [ENTER]
□ONE [ENTER]
□□TWO [ENTER]
□□□THREE [ENTER]
□ONE TWO THREE 3.0 [ENTER]
```

Screen Shows:
```
ONE TWO THREE 1.0
ONE
   TWO
    THREE
ONE TWO THREE 2.0
 ONE
   TWO
    THREE
ONE TWO THREE 3.0
```

Go to the top of the text, and display all lines in the buffer.

You Type: [CTRL] [7] L* [ENTER]

Screen Shows:
```
^L*
ONE TWO THREE 1.0
ONE
 TWO
   THREE
ONE TWO THREE 2.0
ONE
 TWO
   THREE
ONE TWO THREE 3.0
```

Search for the next occurrence of TWO.

You Type: S "TWO" [ENTER]

Screen Shows:
```
S"TWO"
 ONE TWO THREE 1.0
```

Search for all occurrences of TWO that are between the edit pointer and the end of the buffer.

You Type: S*/TWO/ [ENTER]

Screen Shows:
```
S*/TWO/
 ONE TWO THREE 1.0
  TWO
 ONE TWO THREE 2.0
  TWO
 ONE TWO THREE 3.0
```

Go to the top of the buffer, and change the first occurrence of THREE to ONE.

You Type: [CTRL][7] C/THREE/ONE/ [ENTER]

Screen Shows:
```
^ C/THREE/ONE/
 ONE TWO ONE 1.0
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 2, and then use the search command to find each occurrence of TWO that begins in Column 2. Skip all other occurrences.

You Type: [CTRL][7] A2 S*/TWO/ [ENTER]

Screen Shows:
```
^ A2 S*/TWO/
  TWO
  TWO
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 1, and change each occurrence of ONE that begins in that column to XXX.

Note: ONE in Line 1 is not changed, since it does not begin in Column 1.

You Type: [CTRL][7] AC*/ONE/XXX/ [ENTER]

Screen Shows:
```
^ AC*/ONE/XXX/
 XXX TWO ONE 1.0
 XXX
 XXX TWO THREE 2.0
 XXX
 XXX TWO THREE 3.0
```

Go to the top of the buffer, and display the text.

You Type: CTRL 7 L* ENTER

Screen Shows:

```
L*
 XXX TWO ONE 1.0
 XXX
  TWO
   THREE
 XXX TWO THREE 2.0
 XXX
  TWO
   THREE
 XXX TWO THREE 3.0
```

Change the remaining ONE to XXX.

Note: The anchor is no longer set. It is reset to zero after each command is executed.

You Type: C/ONE/XXX/ ENTER

Screen Shows:

```
C/ONE/XXX/
XXX TWO XXX 1.0
```

Move to the beginning of the current line.

You Type: -0 ENTER

Screen Shows:

```
-0
XXX TWO XXX 1.0
```

Change three occurrences of XXX to ZZZ.

You Type: C3/XXX/ZZZ/ ENTER

Screen Shows:

```
C3/XXX/ZZZ/
ZZZ TWO XXX 1.0
ZZZ TWO ZZZ 1.0
ZZZ
```

Sample Session 3

Clear the buffer by deleting its contents:

You Type: CTRL 7 D* ENTER

Display the directory of buffers and macros. The dollar sign ($) identifies the secondary buffer as Buffer 0. The asterisk (*) identifies the primary buffer as Buffer 1. Edit has no macros defined. This is the initial environment when you start Edit.

You Type: .DIR [ENTER]

Screen Shows:

```
.DIR

 BUFFERS:
 $      0
 *      1

 MACROS:
```

Insert some lines into Buffer 1 so that later you can identify it.

You Type:

```
□BUFFER ONE 1.0 [ENTER]
□BUFFER ONE 2.0 [ENTER]
□BUFFER ONE 3.0 [ENTER]
□BUFFER ONE 4.0 [ENTER]
```

Screen Shows:

```
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Display the text in Buffer 1.

You Type: [CTRL][7] L* [ENTER]

Screen Shows:

```
^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Make Buffer 0 the primary buffer. Buffer 1 becomes the secondary buffer.

You Type: B0 [ENTER]

Screen Shows:

```
B0
```

Display the directory of buffers and macros.

Note: The symbols identifying the buffers are now reversed.

You Type: .DIR [ENTER]

Screen Shows:

```
.DIR

 BUFFERS:
 $      1
 *      0

 MACROS:
```

Insert some lines into Buffer 0.

You Type:
```
□BUFFER ZERO 1.0 [ENTER]
□BUFFER ZERO 2.0 [ENTER]
□BUFFER ZERO 3.0 [ENTER]
□BUFFER ZERO 4.0 [ENTER]
```
Screen Shows:
```
BUFFER ZERO 1.0
BUFFER ZERO 2.0
BUFFER ZERO 3.0
BUFFER ZERO 4.0
```

Display the text in Buffer 0.

You Type: [CTRL] [7] L* [ENTER]

Screen Shows:
```
^L*
 BUFFER ZERO 1.0
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Switch to Buffer 1.

You Type: B [ENTER]

Screen Shows:
```
B
```

Display the text in Buffer 1.

You Type: [CTRL] [7] L* [ENTER]

Screen Shows:
```
^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Move the edit pointer to Line 3 in this buffer.

You Type: +2 [ENTER]

Screen Shows:
```
+2
 BUFFER ONE 3.0
```

Switch to Buffer 0.

You Type: B0 [ENTER]

Screen Shows:
```
B0
```

Display the text in Buffer 0.

You Type: L* [ENTER]

Screen Shows:
```
L*
 BUFFER ZERO 1.0
 BUFFER ZERO 2.0
 BUFFER ZERO 3.0
 BUFFER ZERO 4.0
```

Move the edit pointer to Line 2 in this buffer.

You Type: + [ENTER]
Screen Shows: +
BUFFER ZERO 2.0

Switch to Buffer 1.

You Type: B [ENTER]
Screen Shows: B

Display the text in Buffer 1 from the current position of the edit pointer.

Note: The position of the edit pointer has not changed since you switched to Buffer 0.

You Type: L* [ENTER]
Screen Shows: L*
BUFFER ONE 3.0
BUFFER ONE 4.0

Switch to Buffer 0.

You Type: B0 [ENTER]
Screen Shows: B0

Display the text in Buffer 0 from the current position of the edit pointer.

Note: The position of the edit pointer has not changed since you switched to Buffer 1.

You Type: L* [ENTER]
Screen Shows: L*
BUFFER ZERO 2.0
BUFFER ZERO 3.0
BUFFER ZERO 4.0

Delete the contents of Buffer 0.

You Type: [CTRL][7] D* [ENTER]
Screen Shows: ^D*

BUFFER ZERO 1.0
BUFFER ZERO 2.0
BUFFER ZERO 3.0
BUFFER ZERO 4.0

Make Buffer 1 the primary buffer and Buffer 0 the secondary buffer.

You Type: B [ENTER]
Screen Shows: B

Move two lines from the primary buffer (Buffer 1) into the secondary buffer (Buffer 0).

You Type: [CTRL][7] P2 [ENTER]
Screen Shows:
```
^P2
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Switch to Buffer 0, and show that the lines were moved to it.

You Type: B0[CTRL][7]L* [ENTER]
Screen Shows:
```
B0^L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Switch to Buffer 1. Go to the bottom of the buffer, and get the text out of the secondary buffer.

You Type: B/G* [ENTER]
Screen Shows:
```
B/G*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Show the contents of the buffer.

Note: The order of the lines is changed as a result of moving the text.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 BUFFER ONE 3.0
 BUFFER ONE 4.0
 BUFFER ONE 1.0
 BUFFER ONE 2.0
```

Move two lines into the secondary buffer.

You Type: P2 [ENTER]
Screen Shows:
```
P2
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Move to the bottom of the buffer, and get the lines back out of the secondary buffer.

You Type: /G* [ENTER]
Screen Shows:
```
/G*
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Show that the order of the lines is restored.

You Type: [CTRL][7]L*

Screen Shows:

```
L*
 BUFFER ONE 1.0
 BUFFER ONE 2.0
 BUFFER ONE 3.0
 BUFFER ONE 4.0
```

Sample Session 4

Clear the buffer by deleting its contents:

You Type: [CTRL][7]D* [ENTER]

Enter some lines of text.

You Type:

```
□LINE ONE [ENTER]
□SECOND LINE OF TEXT [ENTER]
□THIRD LINE OF TEXT [ENTER]
□FOURTH LINE [ENTER]
□FIFTH LINE [ENTER]
□LAST LINE [ENTER]
```

Screen Shows:

```
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FOURTH LINE
FIFTH LINE
LAST LINE
```

Open the file Oldfile for writing.

You Type: .WRITE"oldfile" [ENTER]

Screen Shows:

```
.WRITE"oldfile"
```

Write all lines to the file.

You Type: [CTRL][7]W* [ENTER]

Screen Shows:

```
^W*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
 FIFTH LINE
 LAST LINE

*END OF TEXT*
```

Close the file.

You Type: .WRITE// [ENTER]

Screen Shows:

```
.WRITE//
```

Verify that the buffer is empty.

You Type: [CTRL][7]L* [ENTER]
Screen Shows: ^L*

Open the file Oldfile for reading.

You Type: .READ"oldfile" [ENTER]
Screen Shows: .READ"oldfile"

Create a new file called Newfile for writing.

You Type: .WRITE"newfile" [ENTER]
Screen Shows: .WRITE"newfile"

Read four lines from the input file. The screen shows the lines as they are read in.

You Type: R4 [ENTER]
Screen Shows:
```
R4
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
```

Read all the remaining text from the file. The screen shows the lines. When there is no more text, the screen shows the *END OF FILE* message.

You Type: R* [ENTER]
Screen Shows:
```
R*
 FIFTH LINE
 LAST LINE

*END OF FILE*
```

Go to the top of the buffer, and display the text to make sure it is inserted into the buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FOURTH LINE
 FIFTH LINE
 LAST LINE
```

Write three lines to the output file, and display the lines.

You Type: W3 [ENTER]
Screen Shows:
```
W3
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
```

Move to the next line and display it.

You Type: + [ENTER]
Screen Shows:
```
+
 FIFTH LINE
```

Show that when writing lines, the editor starts at the current line and not at the top of the buffer.

You Type: W [ENTER]
Screen Shows:
```
W
 FIFTH LINE
```

Go to the top of the buffer, and display the text to be sure that the lines were written to the output file.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 FOURTH LINE
 LAST LINE
```

Clear the buffer.

You Type: [CTRL][7]D* [ENTER]
Screen Shows:
```
^D*
 FOURTH LINE
 LAST LINE
```

Switch to Buffer 2. Open the input file Oldfile, and read two lines from it.

You Type: B2 .READ"oldfile" R2 [ENTER]
Screen Shows:
```
B2 .READ"oldfile" R2
 LINE ONE
 SECOND LINE OF TEXT
```

Switch to Buffer 1. Open the input file Oldfile and read one line of text.

You Type: B .READ"oldfile" R [ENTER]
Screen Shows:
```
B .READ"oldfile" R
 LINE ONE
```

Switch to Buffer 2, and read one line.

Note: Your place in the file was not lost.

You Type: `B2 R [ENTER]`

Screen Shows:
```
B2 R
 THIRD LINE OF TEXT
```

Switch to Buffer 1, and read one line of text.

Note: Your place in the file was not lost.

You Type: `B R [ENTER]`

Screen Shows:
```
B R
 SECOND LINE OF TEXT
```

Switch to Buffer 2, and delete its contents.

You Type: `B2 [CTRL][7]D* [ENTER]`

Screen Shows:
```
B2 ^D*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
```

Insert some extra lines into the buffer.

You Type:
```
□EXTRA LINE ONE [ENTER]
□EXTRA LINE TWO [ENTER]
```

Screen Shows:
```
EXTRA LINE ONE
EXTRA LINE TWO
```

Try to write B2 buffer to file. It fails because you have not opened a file in this buffer.

You Type: `[CTRL][7]W* [ENTER]`

Screen Shows:
```
^W*
*FILE CLOSED*
```

Close the file for Buffer 1, and return to Buffer 2.

You Type: `B .WRITE// B2 [ENTER]`

Screen Shows:
```
B .WRITE// B2
```

Open the old "write" file for reading, and then read it back in.

You Type: `.READ"newfile" R* [ENTER]`

Screen Shows:
```
.READ"newfile" R*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FIFTH LINE

*END OF FILE*
```

Display the contents of the buffer.

Note: It read the file into the beginning of the buffer, since that was the position of the edit pointer.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
 LINE ONE
 SECOND LINE OF TEXT
 THIRD LINE OF TEXT
 FIFTH LINE
 EXTRA LINE ONE
 EXTRA LINE TWO
```

Sample Session 5

Delete all text from the edit buffer.

You Type: CTRL 7 D * ENTER

Insert three lines.

You Type:

```
□LINE ONE ENTER
□LINE TWO ENTER
□LINE THREE ENTER
```

Screen Shows:

```
LINE ONE
LINE TWO
LINE THREE
```

Create a new macro using an empty string.

You Type: .MAC// ENTER

Screen Shows: M:

Display the contents of the macro mode, which is now open.

Note: The E prompt is now M.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
```

Define the macro.

You Type:

```
□FIND ENTER
□S"TWO" ENTER
```

Screen Shows:

```
FIND
S"TWO"
```

Display the contents of the macro.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
FIND
S"TWO"
```

Close the macro's definition.

You Type: Q [ENTER]
Screen Shows: E:

Display the directory of buffers and macros.

You Type: .DIR [ENTER]
Screen Shows:
```
.DIR
 BUFFERS:
 $          0
 *          1

 MACROS:
   FIND
```

Display the contents of the edit buffer.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^ L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND macro to find the string TWO.

You Type: .FIND [ENTER]
Screen Shows:
```
.FIND
 LINE TWO
```

Reopen the definition of the FIND macro.

You Type: .MAC/FIND/ [ENTER]
Screen Shows:
```
.MAC/FIND/
M:
```

Show that the macro is still intact.

You Type: [CTRL][7]L* [ENTER]
Screen Shows:
```
^L*
 FIND
 S"TWO"
```

Add the numeric parameter and the string parameter to the macro's header.

You Type: C/FIND/FIND #N $STR/ [ENTER]
Screen Shows:
```
C/FIND/FIND #N $STR/
 FIND #N $STR
```

Move to the second line of the macro.

You Type: + [ENTER]
Screen Shows:
```
+
 S"TWO"
```

Give the macro's parameters to the S command. Now the FIND macro will perform the same function as the S command.

You Type: `C/"TWO"/ #N $STR/ ENTER`
Screen Shows:
```
C/"TWO"/ #N $STR
 S #N $STR
```

Close the macro's definition.

You Type: `Q ENTER`
Screen Shows:
```
E:
```

Display the contents of the edit buffer.

You Type: `CTRL 7 L* ENTER`
Screen Shows:
```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND macro to find the next two occurrences of LINE.

You Type: `.FIND 2 /LINE/ ENTER`
Screen Shows:
```
.FIND 2 /LINE/
 LINE ONE
 LINE TWO
```

Create a new macro.

You Type: `.MAC// ENTER`
Screen Shows:
```
.MAC//
M:
```

Define the macro FIND_LINE, which performs the same function as the S command except that it returns the edit pointer to the head of the line after finding the last occurrence of STR.

You Type: `□FIND_LINE #N $STR ENTER`
Screen Shows:
```
FIND_LINE #N $STR
```

You Type: `□S #N $STR ENTER`
Screen Shows:
```
S #N $STR
```

Turn off the verify mode.

You Type: `□V0 ENTER`
Screen Shows:
```
V0
```

Move the edit pointer to the first character of the current line.

You Type: `-0 ENTER`
Screen Shows:
```
-0
```

Close the macro's definition.

You Type: Q [ENTER]
Screen Shows:
```
Q

E:
```

Display the contents of the edit buffer.

You Type: [CTRL] [7] L * [ENTER]
Screen Shows:
```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Use the FIND_LINE macro to search for the string TWO.

You Type: .FIND_LINE/TWO/ [ENTER]
Screen Shows:
```
.FIND_LINE/TWO/
 LINE TWO
```

Show that the FIND_LINE macro left the edit pointer at the head of the line.

You Type: L [ENTER]
Screen Shows:
```
L
 LINE TWO
```

Create a new macro.

You Type: .MAC// [ENTER]
Screen Shows:
```
.MAC//
M:
```

Use the exclamation point (!) command to comment itself. Type the following:

```
CONVERT_TO_LINES #N [ENTER]
! This is a comment [ENTER]
! [ENTER]
! This macro converts the next n [ENTER]
! space characters to new line [ENTER]
! characters. [ENTER]
V0          ! Turn verify mode off [ENTER]
            ! to prevent intermediate results [ENTER]
            ! from being displayed. [ENTER]
! [ENTER]
[           ! Begin loop [ENTER]
.SEARCH/ /  ! Search for the space character. [ENTER]
I//         ! Insert empty line (new line character). [ENTER]
-           ! Back up one line. [ENTER]
C/ //       ! Delete the next space character. [ENTER]
L +         ! Show line, move past it. [ENTER]
] #N        ! End of loop. Repeat #N times. [ENTER]
```

Close the macro's definition.

You Type: `Q [ENTER]`
Screen Shows:

```
Q
E:
```

Display the contents of the edit buffer.

You Type: `[CTRL][7]L* [ENTER]`
Screen Shows:

```
^L*
 LINE ONE
 LINE TWO
 LINE THREE
```

Convert all space characters to new line characters.

Note: The loop stops when the C command in the macro cannot find a space to delete.

You Type: `.CONVERT_TO_LINES * [ENTER]`
Screen Shows:

```
.CONVERT_TO_LINES *
 LINE
 LINE
 LINE
```

Display the contents of the edit buffer.

You Type: CTRL 7 L * ENTER

Screen Shows:

```
^L*
 LINE
 ONE
 LINE
 TWO
 LINE
 THREE
```

Edit Quick Reference Summary

EDIT	OS-9 loads the editor and starts it without creating any read or write files. Perform text-file operations by opening files after the editor is running.
EDIT *newfile*	OS-9 loads the editor and starts it. If *newfile* does not exist, Edit creates it and makes it the initial write file. Although this command does not create an initial read file, you can open read files after starting Edit.
EDIT *oldfile*	OS-9 loads the editor and starts it, making the initial read file *oldfile*. The editor creates a new file called SCRATCH as the initial write file. When the edit session is complete, Edit deletes *oldfile* and renames SCRATCH to *oldfile*.
EDIT *oldfile newfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called *newfile* as the initial write file.

Edit Commands

.MACRO	Executes the macro specified by the name following the period (.).
!	Places comments inside a macro, and ignores the remainder of the command line.
□	Inserts a line before the current position of the edit pointer.
ENTER	Moves the edit pointer to the next line, and displays it.
+*n*	Moves the edit pointer forward *n* lines and displays the line.
-*n*	Moves the edit pointer backward *n* lines and displays the line.
+0	Moves the edit pointer to the last character of the line.

-0	Moves the edit pointer to the first character of the current line and displays it.
>*n*	Moves the edit pointer forward *n* characters.
<*n*	Moves the edit pointer backward *n* characters.
CTRL 7 or □ for external terminals	Moves the edit pointer to the beginning of the text.
/	Moves the edit pointer to the end of the text.
[commands] *n*	Repeats the sequence of commands between the two brackets *n* times.
:	Skips to the end of the innermost loop or macro if the fail flag is not on.
A*n*	Sets the SEARCH/CHANGE anchor to Column *n*, restricting searches and changes to those strings starting in Column *n*. This command remains in effect for the current command line.
A0	Returns the anchor to the normal mode of searching so that strings are found regardless of the column in which they start.
B*n*	Makes buffer *n* the primary buffer.
C*n str1 str2*	Changes the next *n* occurrences of *str1* to *str2*.
D*n*	Deletes *n* lines.
E*n str*	Extends (adds the string to the end of) the next *n* lines.
G*n*	Gets *n* lines from the secondary buffer, starting from the top. Inserts the lines before the current position in the primary buffer.
I*n str*	Inserts a line containing *n* copies of the string before the current position of the edit pointer.
K*n*	Kills *n* characters starting at the current position of the edit pointer.
L*n*	Lists (displays) the next *n* lines, starting at the current position of the edit pointer.

M*n*	Changes workspace (memory) size to *n* bytes.
P*n*	Puts (moves) *n* lines from the position of the edit pointer in the primary buffer to the position of the edit pointer in the secondary buffer.
Q	Quits editing (and terminates editor). If you specified a file(s) when you entered Edit, Buffer 1 is written to the output file. The remainder of the input file is copied to the output file. All files are closed.
R*n*	Reads *n* lines from the buffer's input file.
S*n str*	Searches for the next *n* occurrences of the string.
T*n*	Tabs to Column *n* of the present line. If *n* is greater than the line length, Edit extends the line with space.
U	Unextends (truncates) a line at the current position of the edit pointer.
V*mode*	Turns the verify mode on or off.
W*n*	Writes *n* lines to the buffer's output file.
X*n*	Displays *n* lines that precede the edit position. The current line is counted as the first line.

Pseudo Macros

.CHANGE *n str1 str2*	Changes *n* occurrences of *str1* to *str2*.
.DEL *str*	Deletes the macro specified by *str*.
.DIR	Displays the directory of buffers and macros.
.EOB	Tests for the end of the buffer.
.EOF	Tests for the end of the file.
.EOL	Tests for the end of the line.
.F	Exits the innermost loop or macro and sets the fail flag.
.LOAD *str*	Loads macros from the path specified in the string.

.MAC *str*	Opens the macro specified by the string for definition. If you give an empty string, Edit creates a new macro.
.NEOB	Tests for not end of buffer.
.NEOF	Tests for not end of file.
.NEOL	Tests for not end of line.
.NEW	Writes all lines up to the current line to the initial output file, and then attempts to read an equal amount of text from the initial input file. The text read-in is appended to the end of the edit buffer.
.NSTR *str*	Tests to see if *string* does not match the characters at the current position of the edit pointer.
.READ *str*	Opens an OS-9 text file for reading, using *string* as the pathlist.
.S	Exits the innermost loop or macro and succeeds (clears the fail flag).
.SEARCH *n str*	Searches for *n* occurrences of *str*.
.SAVE *str1 str2*	Saves the macros specified in *str1* on the file specified by the pathlist in *str2*.
.SHELL *command line*	Calls OS-9 shell to execute the command line.
.SIZE	Displays the size of memory used and the amount of memory available in the workspace.
.STAR *n*	Tests to see if *n* equals asterisk (infinity).
.STR *str*	Tests to see if *string* matches the characters at the current position of the edit pointer.
.WRITE *str*	Opens an OS-9 text for writing, using *str* as a pathlist.
.ZERO *n*	Tests *n* to see if it is zero.
[	Starts at a macro loop; press CTRL 8.
]	Ends at a macro loop; press CTRL 9.

[^]	Moves edit pointer to beginning of buffer; press CTRL 7.

Editor Error Messages

BAD MACRO NAME	You did not begin the first line in a macro with a legal name. You can close the definition of a macro after you give it a legal name.
BAD NUMBER	You have entered an illegal numeric parameter, probably a number greater than 65,535.
BAD VAR NAME	You have specified an illegal variable name, omitted the variable name, or included a $ or # character in the commands parameter list.
BRACKET MISMATCH	You have not entered brackets in pairs or the brackets are nested too deeply.
BREAK	You pressed CTRL C or E to interrupt the editor. After printing the error message, the editor returns to command entry mode.
DUPL MACRO	You attempted to close a macro definition with an existing macro name. Rename the macro before trying to close its definition.
END OF FILE	You are at the end of the edit buffer.
FILE CLOSED	You tried to write to a file that is not open. Either specify a write file when starting the editor from OS-9, or open an output file using the .WRITE pseudo macro.
MACRO IS OPEN	You must close the macro definition before using the command.
MISSING DELIM	The editor could not find a matching delimiter to complete the string you specified. You must put the entire string on one line.
NOT FOUND	The editor cannot find the specified string or macro.

UNDEFINED VAR — You used a variable that is not specified in the macro's definition parameter list. A variable parameter can be used only in the macro in which it is declared.

WHAT ?? — The editor does not recognize a command. You typed a command that does not exist or misspelled a name.

WORKSPACE FULL — The buffer did not have room for the text you want to insert. Increase the workspace, or remove some text.

OS-9 Error Codes

The following table shows OS-9 error codes in hexadecimal and decimal. Error codes other than those listed are generated by programming languages or user programs.

OS-9 Error Codes

Code HEX	Code DEC	Code Meaning
$01	001	UNCONDITIONAL ABORT. An error occurred from which OS-9 cannot recover. All processes are terminated.
$02	002	KEYBOARD ABORT. You pressed [BREAK] to terminate the current operation.
$03	003	KEYBOARD INTERRUPT. You pressed [SHIFT][BREAK] either to cause the current operation to function as a background task with no video display or to cause the current task to terminate.
$B7	183	ILLEGAL WINDOW TYPE. You tried to define a text type window for graphics or used illegal parameters.
$B8	184	WINDOW ALREADY DEFINED. You tried to create a window that is already established.
$B9	185	FONT NOT FOUND. You tried to use a window font that does not exist.
$BA	186	STACK OVERFLOW. Your process (or processes) requires more stack space than is available on the system.
$BB	187	ILLEGAL ARGUMENT. You have used an argument with a command that is inappropriate.
$BD	189	ILLEGAL COORDINATES. You have given coordinates to a graphics command which are outside the screen boundaries.

Code HEX	Code DEC	Code Meaning
$BE	190	INTERNAL INTEGRITY CHECK. System modules or data are changed and no longer reliable.
$BF	191	BUFFER SIZE IS TOO SMALL. The data you assigned to a buffer is larger than the buffer.
$C0	192	ILLEGAL COMMAND. You have issued a command in a form unacceptable to OS-9.
$C1	193	SCREEN OR WINDOW TABLE IS FULL. You do not have enough room in the system window table to keep track of any more windows or screens.
$C2	194	BAD/UNDEFINED BUFFER NUMBER. You have specified an illegal or undefined buffer number.
$C3	195	ILLEGAL WINDOW DEFINITION. You have tried to give a window illegal parameters.
$C4	196	WINDOW UNDEFINED. You have tried to access a window that you have not yet defined.
$C8	200	PATH TABLE FULL. OS-9 cannot open the file because the system path table is full.
$C9	201	ILLEGAL PATH NUMBER. The path number is too large, or you specified a non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL. Your system cannot handle an interrupt request, because the polling table does not have room for more entries.
$CB	203	ILLEGAL MODE. The specified device cannot perform the indicated input or output function.
$CC	204	DEVICE TABLE FULL. The device table does not have enough room for another device.

Code HEX	Code DEC	Code Meaning
$CD	205	ILLEGAL MODULE HEADER. OS-9 cannot load the specified module because its sync code, header parity, or cyclic redundancy code is incorrect.
$CE	206	MODULE DIRECTORY FULL. The module directory does not have enough room for another module entry.
$CF	207	MEMORY FULL. Process address space is full or your computer does not have sufficient memory to perform the specified task.
$D0	208	ILLEGAL SERVICE REQUEST. The current program has issued a system call containing an illegal code number.
$D1	209	MODULE BUSY. Another process is already using a non-shareable module.
$D2	210	BOUNDARY ERROR. OS-9 has received a memory allocation or deallocation request that is not on a page boundary.
$D3	211	END OF FILE. A read operation has encountered an end-of-file character and has terminated.
$D4	212	RETURNING NON-ALLOCATED MEMORY. The current operation has attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT. The file structure of the specified device is damaged.
$D6	214	NO PERMISSION. The attributes of the specified file or device do not permit the requested access.
$D7	215	BAD PATH NAME. The specified pathlist contains a syntax error, for instance an illegal character.
$D8	216	PATH NAME NOT FOUND. The system cannot find the specified pathlist.

Code HEX	Code DEC	Code Meaning
$D9	217	SEGMENT LIST FULL. The specified file is too fragmented for further expansion.
$DA	218	FILE ALREADY EXISTS. The specified filename already exists in the specified directory.
$DB	219	ILLEGAL BLOCK ADDRESS. The file structure of the specified device is damaged.
$DC	220	PHONE HANGUP - DATA CARRIER DETECT LOST. The data carrier detect is lost on the RS-232 port.
$DD	221	MODULE NOT FOUND. The system received a request to link a module that is not in the specified directory.
$DF	223	SUICIDE ATTEMPT. The current operation has attempted to return to the memory location of the stack.
$E0	224	ILLEGAL PROCESS NUMBER. The specified process does not exist.
$E2	226	NO CHILDREN. The system has issued a *wait service* request but the current process has no dependent process to execute.
$E3	227	ILLEGAL SWI CODE. The system received a software interrupt code that is less than 1 or greater than 3.
$E4	228	PROCESS ABORTED. The system received a signal Code 2 to terminate the current process.
$E5	229	PROCESS TABLE FULL. A fork request cannot execute because the process table has no room for more entries.
$E6	230	ILLEGAL PARAMETER AREA. A fork call has passed incorrect high and low bounds.
$E7	231	KNOWN MODULE. The specified module is for internal use only.

Code HEX	Code DEC	Code Meaning
$E8	232	INCORRECT MODULE CRC. The cyclic redundancy code for the module being accessed is bad.
$E9	233	SIGNAL ERROR. The receiving process has a previous, unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE. The system cannot locate the specified module.
$EB	235	BAD NAME. The specified device, file, or module name is illegal.
$EC	236	BAD MODULE HEADER. The specified module header parity is incorrect.
$ED	237	RAM FULL. No free system random access memory is available: the system address space is full, or there is no physical memory available when requested by the operating system in the system state.
$EE	238	UNKNOWN PROCESS ID. The specified process ID number is incorrect.
$EF	239	NO TASK NUMBER AVAILABLE. All available task numbers are in use.

Device Driver Errors

I/O device drivers generate the following error codes. In most cases, the codes are hardware-dependent. Consult your device manual for more details.

Code HEX	Code DEC	Code Meaning
$F0	240	UNIT ERROR. The specified device unit doesn't exist.
$F1	241	SECTOR ERROR. The specified sector number is out of range.
$F2	242	WRITE PROTECT. The specified device is write-protected.

Code HEX	Code DEC	Code Meaning
$F3	243	CRC ERROR. A cyclic redundancy code error occurred on a read or write verify.
$F4	244	READ ERROR. A data transfer error occurred during a disk read operation, or there is a SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR. An error occurred during a write operation.
$F6	246	NOT READY. The device specified has a *not ready* status.
$F7	247	SEEK ERROR. The system attempted a seek operation on a non-existent sector.
$F8	248	MEDIA FULL. The specified media has insufficient free space for the operation.
$F9	249	WRONG TYPE. An attempt is made to read incompatible media (for instance an attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY. A non-shareable device is in use.
$FB	251	DISK ID CHANGE. You changed diskettes when one or more files are open.
$FC	252	RECORD IS LOCKED-OUT. Another process is accessing the requested record.
$FD	253	NON-SHARABLE FILE BUSY. Another process is accessing the requested file.

Color Computer 2 Compatibility

Color Computer 3 OS-9 Level Two provides compatibility with the Color Computer 2 and OS-9 Level One by letting you use the video display in the Alphanumeric mode (including *Semigraphic* box graphics) and in the Graphics mode. To control the display, it has many built-in functions that you activate using ASCII control characters. Any program written in a language using standard output statements (such as PUT in BASIC) can use these functions. Color Computer BASIC09 has a Graphics Interface Module that can automatically generate most of these codes using BASIC09 RUN statements.

The Color Computer's display system uses a separate memory area for each Display mode. Therefore, operations on the Alpha display do not affect the Graphics display and vice-versa. You can select either display with software control. (See *Getting Started With Extended Color BASIC* for more detailed information.)

The system interprets 8-bit characters sent to the display according to their numerical values, as shown in this chart:

Character Range (Hex)	**Mode/Function**
00 - 0E	Alpha—Cursor and screen control.
0F - 1D	Graphics—Drawing and screen control.
1B	Alpha, Graphics—Changing Palette colors. Alpha mode: 1B 31 2 h change cursor color 1B 31 c h change foreground color 1B 31 d h change background color where h is a hex number from 0 to 3F (0 to 63 decimal) which determines the color.

Character Range (Hex)	Mode/Function
	Graphics mode: 1B 31 pr h — changes foreground/background color where pr is a palette register # (0 - F, hex) where h is a hex number from 0 to 3F (0 to 63 decimal) which determines the color.
20-5F	Alpha—Uppercase characters.
60 - 7F	Alpha—Lowercase characters.
80 - FF	Alpha—Semigraphic patterns.

The device driver CC3IO calls a subroutine module named VDGInt to handle all text and graphics for the Color Computer 2 compatibility mode.

Alpha Mode Display

The Alpha mode is the standard operational mode. Use it to display alphanumeric characters and semigraphic box graphics. Use it also to simulate the operation of a typical computer terminal with functions for scrolling, cursor positioning, clearing the screen, deleting lines, and so on.

The Alpha mode assumes that each 8-bit code the system sends to the display is an ASCII character. If the high-order bit of the code is clear, the system displays the appropriate alphanumeric character. If the high-order bit is set, OS-9 generates a Semigraphic 6 graphics box. See *Getting Started With Extended Color BASIC* for an explanation of semigraphic functions.

The standard 32-column Alpha mode display is handled by the I/O subroutine module VDGInt. CC3IO calls this module (included in the standard boot file) to process all text and semigraphic output.

The following chart provides codes for screen display and cursor control. You can use the functions from the OS-9 system prompt by typing DISPLAY, followed by the appropriate codes. For instance, to clear the screen, type:

```
display 0c [ENTER]
```

To position the cursor at column 16, Line 5 and display the word HELLO, type:

```
display 02 30 25 48 45 4c 4c 4f [ENTER]
```

You can also use the following codes in a language, such as BASIC09. To do so, use decimal numbers with the CHR$ function, such as:

```
print chr$(02);chr$(48);chr$(37);chr$(72)
;chr$(69);chr$(76);chr$(76);chr$(79)
```

Using Alpha Mode Controls with Windows

The control functions in the following chart also function properly under the high resolution windowing systems. References to "screen" are also references to windows.

Alpha Mode Command Codes

Hex Control Code	Decimal Control Code	Name/Function
$01	01	HOME—Returns the cursor to the upper left corner of the screen.
$02	02	CURSOR XY—Moves the cursor to character X of line Y. To arrive at the values for X and Y, add 20 hexadecimal to the location where you want to place the cursor. For example, to position the cursor at Character 5 of Line 10 (hexadecimal A), do these calculations: 5 + 20 = 25 hexadecimal 0A + 20 = 2A hexadecimal The two coordinates are $25 and $2A.
$03	03	ERASE LINE—Erases all characters on the line occupied by the cursor.
$04	04	CLEAR TO END OF LINE—Erases all characters from the cursor position to the end of the line.

Hex Control Code	Decimal Control Code	Name/Function
$05	05	CURSOR ON-OFF—Allows alteration of the cursor based on the value of the next character. Codes are as follow:

Hex	Dec	Char	Function	Default Color
$20	32	space	Cursor OFF	
$21	33	!	Cursor ON	Blue
$22	34	"	Cursor ON	Black
$23	35	#	Cursor ON	
$24	36	$	Cursor ON	
$25	37	%	Cursor ON	
$26	38	&	Cursor ON	
$27	39	'	Cursor ON	
$28	40	(	Cursor ON	
$29	41	)	Cursor ON	
$2A	42	*	Cursor ON	

$06	06	CURSOR RIGHT—Moves the cursor to the right one character position.
$07	07	BELL—Sounds a bell (beep) through monitor speaker.
$08	08	CURSOR LEFT—Moves the cursor to the left one character position.
$09	09	CURSOR UP—Moves the cursor up one line.
$0A	10	CURSOR DOWN (linefeed)—Moves the cursor down one line.
$0C	12	CLEAR SCREEN—Erases the entire screen, and homes the cursor (positions it at the upper left corner of the screen).
$0D	13	RETURN—Returns the cursor to the leftmost character on the line.
$0E	14	DISPLAY ALPHA—Switches the screen from Graphic mode to Alphanumeric mode.

Graphics Mode Display

Use the Graphics mode to display high-resolution 2- or 4-color VDG graphics. The Graphics mode includes commands to set color, plot and erase individual points, draw and erase lines, position the graphics cursor, and draw circles.

You must execute the *display graphics* command before using any other Graphics mode command. This command displays the graphics screen and sets a display format and color.

The first time you enter the display graphics command, OS-9 allocates a 6144-byte display memory. There must be at least that much contiguous free memory available. (You can use MFREE to check free memory.) The system retains the display memory until you give the *end graphics* command, even if the program that initiated the Graphics mode finishes. Always use the end graphics command to release the display memory when you no longer need the Graphics mode.

Graphics mode supports two basic formats. The 2-color format has 256 horizontal by 192 vertical points (G6R mode). The 4-color format has 128 horizontal by 192 vertical points (G6C mode). Either mode provides both color sets. Regardless of the resolution of the selected format, all Graphics mode commands use a 256 by 192 point coordinate system. The X and Y coordinates are always positive numbers. Point 0,0 is the lower left corner of screen.

Many commands use an invisible *graphics cursor* to reduce the output required to generate graphics. You can explicitly set this cursor to any point by using the *set graphics cursor* command. You can also use any other commands that include x,y coordinates (such as *set point*) to move the graphics cursor to the specified position.

Any graphics function that *draws* on the graphics screen requires that the VDGInt module is loaded into memory during the system boot.

Graphics Mode Selection Codes

Code	Format
00	256 x 192 two-color graphics
01	128 x 192 four-color graphics

Color Set and Foreground Color Selection Codes

	2-Color Format			4-Color Format	
	Char	Back-ground	Fore-ground	Back-ground	Fore-ground
Color Set 0	00	Black	Black	Green	Green
	01	Black	Green	Green	Yellow
	02			Green	Blue
	03			Green	Red
Color Set 1	04	Black	Black	Buff	Buff
	05	Black	Buff	Buff	Cyan
	06			Buff	Magenta
	07			Buff	Orange
Color Set 2	08			Black	Black
	09			Black	Dark Green
	10			Black	Med. Green
	11			Black	Light Green
Color Set 3	12			Black	Black
	13			Black	Green
	14			Black	Red
	15			Black	Buff

Graphics Mode Control Commands

Hex Control Code	Decimal Control Code	Name/Function
$0F	15	DISPLAY GRAPHICS—Switches the screen to the Graphics mode. Use this command before any other graphics commands. The first time you use it, the system assigns a 6-kilobyte display buffer for graphics. If 6K of contiguous memory isn't available, OS-9 displays an error. Follow the *display graphics* command with two characters specifying the Graphics mode and color/color set, respectively.
$10	16	PRESET SCREEN—Presets the entire screen to the color code passed by the next character.

Hex Control Code	Decimal Control Code	Name/Function
$11	17	SET COLOR—Sets the foreground color (and color set) to the color specified by the next character but does not change the Graphics mode.
$12	18	END GRAPHICS—Disables the Graphics mode, returns the 6K byte graphics memory area to OS-9 for other use, and switches to Alpha mode.
$13	19	ERASE GRAPHICS—Erases all points by setting them to the background color, and positions the graphics cursor at the desired position.
$14	20	HOME GRAPHICS CURSOR—Moves the graphics cursor to coordinates 0,0 (the lower left corner).
$15	21	SET GRAPHICS CURSOR—Moves the graphics cursor to the given x,y coordinates. For x and y, the system uses the binary value of the two characters that immediately follow.
$16	22	DRAW LINE—Draws a line in the foreground color from the graphics cursor position to the given x,y coordinates. For x and y, the system uses the binary value of the two characters that immediately follow. The graphics cursor moves to the end of the line.
$17	23	ERASE LINE—Operates the same as the *draw line* function, except that OS-9 draws the line in the background color, thus erasing the line.
$18	24	SET POINT—Sets the pixel at point x,y to the foreground color. For x and y, the system uses the binary values of the two characters that immediately follow. The graphics cursor moves to the point set.

Hex Control Code	Decimal Control Code	Name/Function
$19	25	ERASE POINT—Operates the same as the *set point* function, except that OS-9 draws the point in the background color, thus erasing the point.
$1A	26	DRAW CIRCLE—Draws a circle in the foreground color using the graphics cursor as the center point and using the the binary value of the next character as the radius.
$1C	28	ERASE CIRCLE—Operates the same as the *draw circle* function, except that OS-9 draws the circle in the background color, thus erasing the circle.
$1D	29	FLOOD FILL—*paints* with the foreground color, starting at the graphics cursor position and extending over adjacent pixels having the same color as the pixel under the graphics cursor.

Note: When you call FILL the first time, it requests allocation of a 512-byte stack for the fill routine. The system does not return this memory until you terminate graphics with the *end graphics* command.

Note: The chart uses hexadecimal codes for compatibility with the OS-9 DISPLAY command.

Display Control Codes Summary

1st Byte Dec	1st Byte Hex	2nd Byte	3rd Byte	Function
00	00			Null
01	01			Home alpha cursor
02	02	Column + 32	Row + 32	Position alpha cursor
03	03			Erase line

1st Byte Dec	Hex	2nd Byte	3rd Byte	Function
04	04			Erase to End of line
05	05	Cursor Code		Alter Cursor
06	06			Move cursor right
07	07			Sound terminal bell
08	08			Move cursor left
09	09			Move cursor up
10	0A			Move cursor down
11	0B			Erase to End of Screen
12	0C			Clear screen
13	0D			Carriage return
14	0E			Select Alpha mode
15	0F	Mode	Color Code	Select Graphics mode
16	10	Color Code		Preset screen
17	11	Color Code		Select color
18	12			Quit Graphics mode
19	13			Erase screen
20	14			Home Graphics cursor
21	15	X Coord	Y Coord	Move graphics cursor
22	16	X Coord	Y Coord	Draw line to x/y
23	17	X Coord	Y Coord	Erase line to x/y
24	18	X Coord	Y Coord	Set point at x/y
25	19	X Coord	Y Coord	Clear point at x/y
26	1A	Radius		Draw circle
28	1C	Radius		Erase circle
29	1D			Flood Fill

OS-9 Keyboard Codes

Key Definitions With Hexadecimal Values

NORM	SHFT	CTRL	NORM	SHFT	CTRL	NORM	SHFT	CTRL
0 30	0 30	--	@ 40	60	NUL 00	p 70	P 50	DLE 10
1 31	! 21	| 7C	a 61	A 41	SOH 01	q 71	Q 51	DC1 11
2 32	" 22	00	b 62	B 42	STX 02	r 72	R 52	DC2 12
3 33	# 23	~ 7E	c 63	C 43	ETX 03	s 73	S 53	DC3 13
4 34	$ 24	00	d 64	D 44	EOT 04	t 74	T 54	DC4 14
5 35	% 25	00	e 65	E 45	EMD 05	u 75	U 55	NAK 15
6 36	& 26	00	f 66	F 46	ACK 06	v 76	V 56	SYN 16
7 37	' 27	^ 5E	g 67	G 47	BEL 07	w 77	W 57	ETB 17
8 38	(28	[5B	h 68	H 48	BSP 08	x 78	X 58	CAN 18
9 39	) 29	] 5D	i 69	I 49	HT 09	y 79	Y 59	EM 19
: 3A	* 2A	00	j 6A	J 4A	LF 0A	z 7A	Z 5A	SUM 1A
; 3B	+ 2B	00	k 6B	K 4B	VT 0B			
, 2C	< 3C	{ 7B	l 6C	L 4C	FF 0C			
- 2D	= 3D	_ 5F	m 6D	M 4D	DR 0D			
. 2E	> 3E	} 7D	n 6E	N 4E	CO 0E			
/ 2F	? 3F	\ 5C	o 6F	O 4F	CI 0F			

Function Keys

	NORM	SHFT	CTRL
BREAK	05	03	1B
ENTER	0D	0D	0D
SPACE	20	20	20
←	08	18	10
→	09	19	11
↓	0A	1A	12
↑	0C	1C	13

OS-9 Keyboard Control Functions

Key Definitions for Special Functions and Characters

Key Combination	Control Function or Character
[ALT]	Alternate key—Sets the high order bit on a character. Press [ALT] *char*.
[CTRL]	Use as a control key.
[BREAK] or [CTRL][E]	Stops the program currently executing.
[CTRL][-]	Generates an underscore (_).
[CTRL][,]	Generates a left brace ({).
[CTRL][.]	Generates a right brace (}).
[CTRL][/]	Generates a reverse slash (\).
[CTRL][BREAK]	Generates an end-of-file (EOF). This sequence is the same as pressing [ESC] on a standard terminal.
[←] or [CTRL][H]	Generates a backspace.
[SHIFT][←] or [CTRL][X]	Deletes the entire current line.
[SHIFT][BREAK] or [CTRL][C]	Interrupts the video display of a running program. This sequence reactivates the shell and then runs the program as a background task.
[CTRL][0]	Upper-/lowercase shift lock function.
[CTRL][1]	Generates a vertical bar (|) in reverse video.
[CTRL][3]	Generates a tilde (~) character.
[CTRL][7]	Generates an up arrow or caret (^).
[CTRL][8]	Generates a left bracket ([).
[CTRL][9]	Generates a right bracket (]).

Key Combination	Control Function or Character
CTRL A	Repeats the previous command line.
CTRL D	Redisplays the command line.
CTRL W	Temporarily halts output to the screen. Press any key to resume output.
CTRL CLEAR	Enable/Disable Keyboard mouse.
CLEAR	Change screens.
SHIFT CLEAR	Change screens in reverse order.
ENTER	

Index

ACIAPAK 5-6, 5-7, 6-96
active state 4-2
address 2-4
    memory 4-5
allocate memory for devices 6-55
alpha mode B-3
    select B-10
alphanumeric mode B-1
ampersand separator 3-6
append files 6-68
application program 1-3
arglist 6-2
ASCII 2-5
    control characters B-1
    convert 6-38
ASM 3-2
asterisk, editor 7-3
ATTR 2-10, 6-5
attribute 2-5, 2-8, 2-10, 6-5
auto-answer modem 6-96, 6-108

background
    color B-7
    process 3-7
    task 4-1
    screen 5-2
backspace 6-93
    character 6-94, 6-106, 6-107
    editor 7-2
    over line 6-93, 6-106
BACKUP 5-4, 6-7
backup a directory 6-39
BASIC09 2-5, 2-6, 3-13, B-1
baud rate 5-4, 5-5, 5-6, 6-96, 6-98, 6-109
begin a window 6-103
bell
    character 6-95, 6-108
    sound B-10
bit 2-1
    stop 5-5, 5-6
    user 2-11
bitmap 2-5
block
    number 4-5
    devices 1-2
bootstrap 5-1
    file 5-2
box graphics B-3
brackets 6-3
buffer 3-7, 7-2
    edit 7-1
    secondary 7-1
    text 7-1
BUILD 2-6, 3-10, 6-10, 6-71, 6-75
built-in commands 3-1, 3-11
byte 2-1

carriage return B-10
CC3Disk 5-1
CC3Go 5-2
CC3IO 5-1, B-2
chaining programs 6-44
change
    attributes 2-10, 2-11
    directory 6-12, 6-91, 6-84
    file name 6-84
    priority 3-12, 6-88
    system parameters 6-93
character
    ASCII 2-5
    delete 6-107
    devices 1-2
    backspace 6-94, 6-106, 6-107
    bell 6-95, 6-108
    delete line 6-94, 6-107
    dup 6-95, 6-107
    end-of-file 6-94, 6-107

character (*cont'd*)
end-of-record 6-94, 6-107
lowercase B-1
pause 6-95, 6-108
quit 6-95, 6-108
reprint 6-95, 6-107
terminate 6-95, 6-108
uppercase B-2
CHD 3-11, 6-12
check disk structure 6-25
child process 3-6, 4-2
CHX 3-11, 6-12
circle
draw B-9
erase B-9, B-10
clear
screen B-5
to end-of-line B-4
clock 5-2
cluster 2-4, 2-5
CMDS directory 5-1, 5-3, 5-4
CMP 6-14
COBBLER 6-16, 6-72
code
alpha mode control B-4
cursor B-5
object 2-7
position-independent 4-8
re-entrant 4-6
color
background B-7
foreground B-7, B-8
select B-10
set, graphics B-7
combine files 6-68
command
grouping 3-2, 3-9
help 6-51
interpreter 6-90
line 3-1, 3-2
parameters, editor 7-3
separator 3-1, 3-5
summary, editor 7-55
command codes
alpha mode B-4
graphics B-7
commandname 6-2
commands
ASM 3-2
ATTR 2-10, 2-11, 6-5
BACKUP 6-7
BUILD 2-6, 3-10, 6-10, 6-71, 6-75
built-in 3-11
CHD 3-11, 6-12
CHX 3-11, 6-12
CMP 6-14
COBBLER 6-16, 6-72
CONFIG 5-2, 5-3, 5-4, 6-18
COPY 2-3, 3-6, 4-8, 6-22
DATE 6-24
DCHECK 6-25
DEINIZ 6-30
DEL 6-31
DELDIR 2-3, 6-33
DIR 2-6, 2-9, 6-35
DISPLAY 6-38
DSAVE 6-39
DUMP 2-8, 6-72
ECHO 6-42
edit macro 7-28
editor 7-2
ERROR 5-2, 6-43
EX 3-11, 6-44
FORMAT 6-46
FREE 6-49
GET 2-6
HELP 6-51
i 3-11
IDENT 3-3, 6-52
INIZ 6-55
KILL 3-12, 6-56
LINK 6-58
LIST 2-3, 2-5, 2-8, 3-4, 6-59
LOAD 4-7, 6-61

commands (*cont'd*)
MAKDIR 2-3, 2-11, 6-63
MDIR 6-64
MERGE 6-68
MFREE 6-69
MODPATCH 6-70
MONTYPE 6-74
OS9GEN 5-2, 5-3, 6-76
p 3-12
PROCS 3-7, 4-2, 6-80
PUT 2-6
PWD 6-82
PXD 6-82
RENAME 6-84
RUNB 3-13
SEEK 2-6
SETIME 5-3, 6-86
SETPR 3-12, 6-88
SHELL 3-6, 6-90
t 3-12
TMODE 6-93
TUNEPORT 6-98
UNLINK 4-7, 4-8, 6-100
w 3-12
WCREATE 6-103
x 3-12
XMODE 5-4, 5-5, 5-7, 6-106
comment, in a program 3-12
compare files 6-14
concurrent
execution 3-5, 6-91
mode 3-10
process 3-9
task 4-1
CONFIG 5-2, 5-3, 5-4, 6-18
control
characters, ASCII B-1
keys, editor 7-2
convert to ASCII 6-38
COPY 2-3, 3-6, 4-8, 6-22
copy
a directory 6-39
diskettes 6-7
CPU 4-1
priority 6-88
CRC 2-7, 6-71
create
a directory 6-63
a file 6-10
OS9Boot 6-16, 6-76
process 3-6
system diskette 5-3, 5-4, 6-16, 6-18, 6-76
current
directory 4-4, 6-12
processes 6-80
cursor
on/off B-5
codes B-5
control B-1, B-4
graphics B-6, B-8
home B-4
move B-5, B-10
cyclic redundancy checksum 2-7

data format 2-1
data output, halt 7-3
data
redirect 3-4
input/output 1-2
passing 4-4
process 2-1
sending 2-1
transfer 2-1
DATE 6-24
date 2-5
set 6-86
day 6-2
DCHECK 6-25
deallocate a device 6-30
DEINIZ 6-30
DEL 6-31
delay, not ready 5-6
DELDIR 2-3, 6-33

delete
a character 6-107
a directory 6-33
a line 7-3
a memory module 4-7, 6-100
files 6-31
line character 6-94, 6-107
lines, editor 7-10
descriptor
device 1-2
file 2-3
detach a device 6-30
device
allocate memory 6-55
block-oriented 1-2
character 1-2
deallocate 6-30
descriptor 1-2, 5-1
driver 1-2, 2-1, 5-1
driver initialization 6-55
name 2-12, 2-13
window 2-12 - 2-13
devname 6-2
DIR 2-6, 2-9, 6-35
directory 2-2, 2-3
attribute 2-8
change 6-12, 6-91
change name 6-84
CMDS 5-1, 5-3, 5-4
copy 6-39
create 6-63
current 4-4, 6-12
delete 6-33
list 6-35
module 4-6
ownership 2-8
SYS 5-1, 5-4
view 6-82
working 6-12
dirname 6-2
disable echo 6-94, 6-107
disk
cluster 2-4
file 2-3, 2-4
I/O 3-8
initialization 6-46
names 2-12
ownership 2-8
sector 2-4
structure, check 6-25
raw I/O 3-8
unused sectors 6-49
diskette
copy 6-7
density 2-5
tracks 2-5
system 2-2
DISPLAY 6-38
display
a directory 6-35
current processes 6-80
date and time 6-24
error message 6-43
execution directory 6-82
file contents 6-59
free memory 6-69
graphics B-7
help 6-51
memory module names 6-64
messages 6-42
on next line 7-2
text, editor 7-6
unused disk sectors 6-49
working directory 6-82
double density 2-5
draw
a circle B-9
a line B-8, B-10
drivers, device 1-2
DSAVE 6-39
DUMP 2-8, 6-72
dup character 6-95, 6-107
duplicate
last line 6-95
line 6-107

ECHO 6-42
echo 6-92
    enable/disable 6-94, 6-107
edit
    buffer 7-1
    commands, EDIT 7-5
    pointer 7-1, 7-2, 7-7
EDIT, editor 7-5
editor 7-1
    backspace 7-2
    command summary 7-55
    command syntax 7-4
    commands 7-2
    control keys 7-2
    delete lines 7-10
    error messages 7-59
    getting started 7-4
    insert lines 7-10
    interrupt 7-3
    numeric parameters 7-3
    quick reference 7-55
    searching 7-13
    substituting 7-13
    terminate 7-2
    text file operations 7-18
    using the asterisk 7-3
ellipsis 6-3
enable echo 6-94, 6-101
end graphics B-8
end-of-file
    terminate 7-2
    character 6-94, 6-101
end-of-line
    clearing B-4
    erase B-10
end-of-record character 6-94, 6-107
erase
    a circle B-9, B-10
    a line B-10
    graphics B-8
    line B-4, B-8, B-9
    point B-9
erase (*cont'd*)
    to end-of-line B-10
Errmsg 5-2
ERROR 5-2, 6-43
error 3-12, 6-92
    message file 5-2
    messages, editor 7-59
    output 6-91
    path 3-4
establish a directory 6-63
EX 3-11, 6-44
exclamation mark separator 3-8
execute
    a program 6-90
    permission 2-9, 2-10
execution
    concurrent 3-5, 6-91
    modifier 3-1, 3-3
    sequential 3-5, 3-6, 6-91

fields 2-6
file 2-2 - 2-4
    attribute 2-8
    change name 6-84
    compare 6-14
    copy 6-22
    create 6-10
    delete 6-31
    descriptor 2-3
    descriptor sector 2-5
    display contents 6-59
    load in memory 6-61
    merge 6-68
    manager 5-1
    OS9Boot 5-4
    ownership 2-8
    pointer 2-4
    procedure 2-6, 3-10, 3-11
    random access 2-6
    security 2-8
    single-user 2-8
    size 2-5

file (*cont'd*)
Startup 2-6, 5-1, 5-3, 5-4
text 2-5
filename 2-3, 6-2
fill portion of screen B-9
flood fill B-9, B-10
floppy disk names 2-12
fonts 5-2
foreground color B-7, B-8
fork 3-7, 4-6
request 4-3
FORMAT 6-46
FREE 6-49

generate messages 6-42
GET 2-6
getting started, editor 7-4
graphic window fonts 5-2
graphics B-1
color set B-7
command codes B-7
cursor B-6, B-8
mode, select B-10
end B-8
erase B-8
medium resolution B-6
VDG B-6
group 2-7
grouping, commands 3-9

halt data output 7-3
hardware 1-2
header
information 6-52
module 2-7, 3-3, 4-7
HELP 6-51
hex 6-2
hexadecimal code display 6-38
home
alpha cursor B-9
cursor B-4
hours 6-2

I-code 3-13
I/O
paths 3-4
transfers 3-8
raw 3-8
ID, process 4-4
IDENT 3-3, 6-52
images, pointer 5-2
immortal
process 6-91
shell 3-11
INIT 5-1
initialize
a disk 6-46
a window 6-103
INIZ 6-55
input 2-1
lines 3-12
path 3-4
redirect 6-91
standard 4-4
insert lines, editor 7-10
interpreter, commands 6-90
interprocess communication 3-7
interrupt editor 7-3
IOMAN 1-2, 1-3, 5-1

kernel 1-1, 1-2
keyboard 1-1
keyword 3-1 - 3-3
KILL 6-56
kill 3-12
a directory 6-33, 6-33
files 6-31

length
of video page 6-94
word 5-5, 5-6, 6-96, 6-109
line
backspace 6-93, 6-106
delete, editor 7-3
draw B-8, B-10
duplicate 6-95

line (*cont'd*)
duplication 6-107
erase B-4, B-8, B-9, B-10
syntax 6-1
linefeed 6-94, 6-107
lines, command 3-1
LINK 6-58
LIST 2-3, 2-5, 2-8, 3-4, 6-59
list
a directory 6-35
current processes 6-80
memory module names 6-64
segment 2-5
with program files 2-8
LOAD 4-7, 4-7, 6-61
lock a module 6-58
lockout 2-11
logical sector 2-3, 2-4
lowercase 6-93, 6-106
characters B-2

machine language 3-12
macro text editor 7-1
macros, edit 7-25
MAKDIR 2-11, 2-3, 6-63
management, memory 4-5
manager
pipe 1-2
random block 1-2
mark space 6-95, 6-108
MDIR 6-64
MDM kill 5-6, 5-7
medium resolution graphics B-6
memory
address 4-5
allocation 3-1
display free 6-69
load a file into 6-61
management 1-1, 4-5
size modifier 3-3
memory modules
lock 6-58
memory modules (*cont'd*)
unlink 6-100
deleting 4-7
display names 6-64
MERGE 6-68
messages with ECHO 6-42
messages, error 6-43
MFREE 6-69
minutes 6-2
MMU 4-5
mode, alpha B-3
mode
alphanumeric B-1
concurrent 3-10
semigraphic B-1
modem 1-1, 5-6, 5-7
auto-answer 6-108
name 2-12
modifier 3-1 - 3-3
execution 3-1, 3-3
memory size 3-3
redirection 3-5
modname 6-2
MODPAK 5-7
MODPATCH 6-70, 6-71, 6-72, 6-73
module 1-3
deleting memory 4-7
directory 4-6
header 2-7, 3-3, 4-7
header information 6-52
loading 4-7
lock in memory 6-58
primary 4-3
program 2-7
unlink 4-8
month 6-2
MONTYPE 6-74, 6-75
move cursor B-4, B-5, B-10
multiprogramming 4-1
multitasking 1-1

name
device 2-12, 2-13
modem 2-12

name (*cont'd*)
    printer 2-12
    program 3-3
    terminal 2-12
next line, display 7-2
not ready delay 5-6
notations, syntax 6-1
null count 6-94, 6-107
number
    priority 3-12
    user 2-8, 4-4
numeric parameters, editor 7-3

object code 2-7
operating system 1-3
opts 6-2
OS9Boot 5-1, 5-4
    create 6-16, 6-76
OS9Gen 5-2, 5-3, 6-72, 6-76
OS9p2 5-1
output 2-1
    error 6-91
    path 3-4, 4-4
    redirect 3-11, 6-91
owner 2-5, 2-8

page length, video 6-107
pages 3-3
paint B-9
parameter 3-1, 4-4
    change system 6-93
    command editor 7-3
paramlist 6-2
parent process 4-2
parity 5-6, 5-7, 6-95, 6-108
passing data 4-4
pathlist 6-2
paths 2-1
    I/O 3-4
    output 4-4
    standard 3-4, 6-93
patterns, semigraphic B-2
pause 6-94
    character 6-95, 6-108
    screen 6-107
permission 6-2
    execute 2-9, 2-10
    read 2-9, 2-10
    write 2-9, 2-10
physical sector 2-4
PIC 4-8
pipe 1-2, 3-7, 5-1
pipelines 3-7, 3-8
PIPEMAN 5-1
Piper 5-1
point
    erase B-9
    set B-8, B-10
pointer
    edit 7-1, 7-2
    editor 7-7
    file 2-4
    images 5-2
port 1-2
port, RS-232 5-4, 6-109
position alpha cursor B-9
position-independent 2-7
    code 4-8
prepare a disk 6-46
preset screen B-7
previous line repeat 7-2
primary module 4-3
PRINTER 5-1
printer 1-1, 1-2
    name 2-12
    test 6-98
priority
    number 3-12
    change 6-88
    process 4-2, 4-4
procedure file 2-6, 3-10, 3-11
process
    background 3-7
    chaining 6-44
    child 3-6
    create 3-6
    current 6-80

process (*cont'd*)
data 2-1
fork 3-7
ID 4-4
memory size 6-91
priority 4-2, 4-4
properties 4-4
sibling 4-3
state 4-2
terminate 6-56
time sharing 4-1
processor time 4-1
procID 6-2
PROCS 3-7, 4-2, 6-80
program
application 1-3
chaining 6-44
comments in 3-12
execution 6-90
modules 2-7
name 3-3
size 3-3
prompt 3-12
prompting 6-92
properties, process 4-4
public 2-9, 2-10
PUT 2-6
PWD 6-82
PXD 6-82

quick reference, editor 7-55
quit character 6-95, 6-108

RAM 4-5
random access 1-2
files 2-6
random block file manager 1-2
rate, baud 5-4, 5-5, 5-6, 6-96, 6-98, 6-109
raw I/O 3-8
RBF 5-1
read 2-1, 2-11, 2-4
permission 2-10, 2-9
readers 3-8
record 2-2, 2-6
lockout 2-11
redirect
data 3-4
input 6-91
output 6-91
redirection 3-1
modifiers 3-4, 3-5
output 3-11
symbols 3-5
re-entrant code 4-6
remove
directory 6-33
files 6-31
memory module 6-100
RENAME 6-84
repeat previous line 7-2
reprint character 6-95, 6-107
reserved characters 3-3
ROOT 2-2, 2-3
route data 3-4
RS-232 5-1, 5-4, 6-96, 6-109
run-time module 3-12
RUNB 3-13

SAVE 6-72
SCF 5-1
screen
alpha B-5
background 5-2
clear B-5
control B-1
pause 6-94, 6-107
preset B-7
scroll pause 6-94, 6-107
searching, editor 7-13
secondary buffer 7-1
seconds 6-2
sector 2-4
copy 6-7
displayed unused 6-49
file descriptor 2-5
logical 2-3

security
    file 2-8
    permission 6-5
SEEK 2-6
segment list 2-5
select
    alpha mode B-10
    color B-10
    graphics mode B-10
semicolon, sequential
  execution 3-6
semigraphic
    mode B-1
    patterns B-2
sending data 2-1
separator 3-1
    ampersand 3-6
    command 3-5
    exclamation mark 3-8
sequential execution 3-5, 3-6,
  6-91
set a window 6-103
set a point B-8, B-10
set priority 3-12
SETIME 5-3, 6-86
SETPR 6-88
share time 4-1
shell 1-3, 3-1–3-3, 3-8, 6-3
SHELL 3-6, 6-90
show
    a directory 6-35
    error message 6-43
    execution directory 6-82
    file contents 6-59
    free memory 6-69
    header information 6-52
    memory module
      names 6-64
    working directory 6-82
sibling processes 4-3
sign bit 2-2
simultaneous execution 3-5
single-user file 2-8
SIO 5-1
size
    file 2-5
    process memory 6-91
    program 3-3
slash in device names 2-13
sleeping 4-3
software fonts 5-2
sound bell B-10
standard input 4-4, 6-93
standard paths 3-4, 6-93
start a window 6-103
Startup 2-2, 2-6, 5-1, 5-3,
  5-4, 6-75
state 4-2, 4-2
Stdfonts 5-2
Stdpats 5-2
Stdptrs 5-2
stop bit 5-5, 5-6
string parameters, editor 7-4
subdirectory 2-3
    delete 6-33
submanager 1-2
subshell 3-10
substituting, editor 7-13
summary, commands 6-3, 6-4
super user 6-56
switch screen B-5
symbols, redirection 3-5
syntax 6-1
SYS directory 5-1, 5-4
system
    administrator 1-1
    date 6-86
    disk create 5-3, 5-4
    parameters 6-93
    priority 6-88
    time 6-86
system diskette 2-2
    create 6-16, 6-18, 6-76

task, background 4-1
term 1-1
TERM 5-1
TERM-VDG 6-96
TERM-WIN 6-109

terminal name 2-12
terminals 1-2
terminate
    a character 6-95, 6-108
    a process 6-56
    the editor 7-2
    on error 6-92
test delay loop 6-98
text 6-2, B-2
    buffers 7-1
    display, editor 7-6
    editing 7-1
    file operations, editor 7-18
    files 2-5
tick 4-1, 4-2
tickcount 6-2
time 2-5, 6-24
    sharing, process 4-1
    CPU 4-1
    processor 4-1
    set 6-86
timeslice 4-1, 4-2
TMODE 6-93
tracks 2-5
transfer, I/O 3-8
transferring data 3-7
TUNEPORT 6-98
turn on
    cursor B-5
    echo 6-92
    prompting 6-92
type 5-7
    ACIA 6-96, 6-108
    of window 6-103
    value 5-7

UNLINK 4-7, 4-8, 6-100
unused disk sectors 6-49
update mode 2-11
uppercase 6-93, 6-106
    characters B-2

user
    bit 2-11
    number 2-8, 4-4

value 6-2
    type 5-7
variable 6-1
VDG graphics B-6
video 1-1
    page length 6-94, 6-107
view
    current processes 6-80
    error messages 6-43
    working directory 6-82

waiting state 4-3
WCREATE 6-103
window 5-2
    alpha mode controls B-3
    descriptor 2-12
    initialization 6-103
    type 6-103
word length 5-5, 5-6, 6-96, 6-109
working directory 6-12
write 2-1, 2-4, 2-11
    permission 2-9, 2-10

XMODE 5-4, 5-5, 6-106

year 6-2

BASIC09
Reference

Contents

Chapter 1 Looking At The Basics
Using BASIC09 1-2
Requesting More Memory 1-3
Writing Procedures 1-5
Modules of Other Languages 1-5
Executing Procedures 1-5
Leaving BASIC09 1-5
The Keyboard and BASIC09 1-5

Chapter 2 Sample Session
Creating a Procedure 2-1
Commands and Program Lines 2-2
Executing a Procedure 2-3

Chapter 3 The System Mode
Renaming Procedures 3-2
Listing Procedure Names 3-2
Listing Procedures 3-2
Listing Procedure Names to a File 3-4
Listing Procedures to a Printer 3-4
Using a Wildcard 3-5
Saving Procedures 3-5
Loading Procedures 3-6
Deleting Procedures from the Workspace 3-6
Changing Directories 3-7
Executing OS-9 Commands 3-8

Chapter 4 The Edit Mode
Edit Commands 4-1
Using the Editor 4-2
Searching Through a Procedure 4-4
Using [ENTER] 4-4
Using the Plus Sign to Move Forward 4-4
Accessing a Line Using the Line Number 4-5
Using the Minus Sign to Move Backward 4-5
The Global Symbol 4-5
Using LIST 4-6
Deleting Lines 4-6
Changing Text 4-7
Searching for Text 4-9

Renumbering Lines 4-10
Adding Lines 4-10
The Next Step 4-12

Chapter 5 The Debug Mode
Entering the Debug Mode 5-1
When Things Go Wrong 5-4
Using the Trace Function 5-5
What About Loops? 5-5
In Multiple Procedures 5-6

Chapter 6 Data and Variables
Data Types 6-1
The Byte Data Type 6-2
The Integer Data Type 6-3
The Real Data Type 6-3
String Variables 6-4
The Boolean Type 6-5
Automatic Type Conversion 6-6
Constants 6-6
String Constants 6-7
Variables 6-7
Passing Variables 6-8
Arrays 6-9
Complex Data Types 6-13

Chapter 7 Expressions, Operators, and Functions
Manipulating Data 7-1
Expressions 7-1
Type Conversion 7-2
Operators 7-2
BASIC09 Expression Operators 7-3
Arithmetic Operators 7-3
Hierarchy of Operators 7-4
Relational Operators 7-5
String Operators 7-6
Logical Operators 7-7
Functions 7-7

Chapter 8 Disk Files
Types of Access for Files 8-1
Sequential Files 8-2
Sequential File Creation, Storage, and Retrieval 8-2
Changing Data in a Sequential File 8-4
INPUT and Sequential Files 8-5

Random Access Files 8-5
Creating Random Access Files 8-6
Using Arrays With Random Access Files 8-9
Using Complex Data Structures 8-11

Chapter 9 Displaying Text and Graphics
ASCII Codes 9-1
Low Resolution Graphics Characters 9-4
Special Characters in High-Resolution 9-8
Medium-Resolution Graphics 9-8
Formats and Colors 9-10
The Draw Pointer 9-12
High-Resolution Graphics 9-26
Establishing a Hardware Window 9-32
Defining Windows 9-33
The Palette 9-34
Establishing a Graphics Window 9-35
Starting a Shell in a Window 9-36
Using High-Level Graphics with 128K 9-37
Creating Windows From BASIC09 9-39
Creating Overlay Windows 9-41
The Graphics Cursor and the Draw Pointer 9-42
High Resolution Text 9-42
Using Fonts 9-43
High Resolution Quick Reference 9-44

Chapter 10 BASIC09 Quick Reference
Statements and Functions 10-1
Commands By Type 10-7
Statements 10-7
Transcendental Functions 10-7
Numeric Functions 10-7
String Functions 10-7
Miscellaneous Functions 10-7
Data Types 10-8
Types of Access for Files 10-8
Command Mode 10-9
Edit Commands 10-10
Debug Commands 10-11

Chapter 11 BASIC09 Command Reference
Keyword Format 11-11
The Syntax Line 11-1
Sample Programs 11-3

Chapter 12 Program Optimization
Optimum Use of Numeric Data Types 12-1
Arithmetic Functions Ranked by Speed 12-3
Quicker Loops 12-3
Arrays and Data Structures 12-4
The PACK Command 12-4
Minimizing Constant Expressions and Subexpressions 12-4
Input and Output 12-4

Appendix A Error Codes
Signal Errors A-1
BASIC09 Error Codes A-1
Windowing and System Errors A-3

Appendix B The Inkey Program
Assembly Language Listing of Inkey B-1

Index

Chapter 1

Looking at the Basics

BASIC09 is a computer language created for use with the OS-9 operating system. Along with standard BASIC language statements and functions, it includes the most useful elements of the PASCAL computer language.

In brief, BASIC09's advantages are:

Fast execution speed — BASIC09 compiles procedure lines as you enter them. When you finish a procedure, you can compile it further. The result? Procedures that execute nearly as fast as machine language.

Full feature editing — The text editor features automatic line formatting, search, search and change, global search, global search and change, line renumbering, and much more. You can move in and out of the editor quickly and easily.

Modular programming functions — You can write small, easy-to-understand procedures, then chain them to create sophisticated programs. You can call one procedure from another, regardless of whether the called procedure is in memory or on disk.

Interfacing to OS-9 — Both you and your procedures can take advantage of almost any OS-9 function from within BASIC, including the execution of disk management commands and application programs.

Structured programming — You can structure procedures more efficiently and clearly by taking advantage of a variety of loop commands, optional line numbering, and BASIC09's ability to call modules written in other computer languages.

Memory saving features

Strings can be any length. For each operation, you can select the most efficient of five available data types. Compiled procedures use less space. You can save several procedures into one file.

Complex data structures

Combine any type of data into a single dimensioned data structure that you can move, store, and assign easily and quickly.

Sophisticated graphics

BASIC09 has three levels of graphics. The high resolution graphics and text capabilities feature more than 50 functions.

High speed, precision math

BASIC09 has a full range of fast and accurate math and transcendental capabilities including powers, roots, trigonometry, logic, and Boolean functions.

Simple and fast debugging

BASIC09 provides superior debugging functions. It checks syntax as you enter lines. It points to the location of your errors and tells you what they are. You can stop programs, enter the debugger, then continue execution. Execution errors automatically put you in a debugging mode where you can examine values, and step and trace your way through faulty procedures.

Using BASIC09

Before anything else, make a backup copy of your BASIC09/CONFIG diskette. You can do this using the BACKUP command. If you are not familiar with BACKUP, see Chapter 3 of *Getting Started With OS-9*.

To use BASIC09, boot your computer as described in *Getting Started With OS-9*. Replace the system diskette in Drive /D0 with the BASIC09/CONFIG backup diskette and type:

```
basic09 [ENTER]
```

After a short pause, during which OS-9 loads BASIC09 from the diskette, the screen displays the copyright and a new *prompt*, like this:

```
                    BASIC09
                RS VERSION 01.00.01
          COPYRIGHT 1980 BY MOTOROLA INC.
            AND MICROWARE SYSTEMS CORP.
              REPRODUCED UNDER LICENSE
                   TO TANDY CORP.
              ALL RIGHTS RESERVED

Basic09
Ready
B:
```

The B: indicates that your computer is in the BASIC09 *command mode*. From the command mode, you can issue instructions to the system executive to manipulate procedures (programs).

Requesting More Memory

Unless you specify otherwise, BASIC09 automatically sets aside 8192 bytes of memory as a workspace into which you can type or load procedures. BASIC09 reserves approximately 1200 bytes of the workspace for internal use, leaving you with 6992 bytes for workspace.

There are two ways to set aside more memory for BASIC09 operations:

- You can reserve extra memory when you first enter BASIC09 by using the OS-9 *memory size* option. For instance, to reserve 18,176 bytes, enter this command to initialize BASIC09:

  ```
  basic09 #18k ENTER
  ```

- You can also request additional memory after loading BASIC09. At the command prompt, `B:`, type:

  ```
  mem 18000 ENTER
  ```

  This tells BASIC09 to set aside a total of 18,000 bytes of memory, if they are available.

In both cases, because BASIC09 rounds the amount you request to the next multiple of 256, the actual reserved memory is 18176 bytes.

> **Note:** If your system does not have enough free memory to reserve the amount you specify, the workspace size does not change.

You can also use the MEM command to reduce memory. However, BASIC09 does not reduce the size of the workspace if doing so destroys resident procedures.

Writing Procedures

BASIC09 is a *modular* programming language. Several procedures can occupy memory at the same time. Each procedure performs a particular function but can also interact with others to form a sophisticated program.

To create or change procedures, enter the *edit mode* by typing either `edit` [ENTER] or [E] [ENTER] at the `B:` prompt. From now on, when directing you to enter the edit mode, this manual uses the easier to type [E] command.

Each time you type a procedure line and press [ENTER], the editor checks for common errors. This automatic checking lets you catch mistakes before you run the program, saving you testing and rewriting time. You can even let the automatic checking help you learn the rules of BASIC09. If you are not sure about a syntax, go ahead and type it the way you think is correct. If you guess wrong, BASIC09 shows where the error is and displays a message to tell what is wrong.

BASIC09's use of modules lets you divide large and complex projects into smaller, easily manageable sections. Not only are the smaller procedures easier to write and understand, they are also easier to test. As well, because BASIC09 lets you *call* procedures that are outside the *workspace* (the computer's memory where you write and edit procedures), you can accumulate libraries of procedures to incorporate into future programs.

You can work on a program's procedures either individually or as a group. For example, to work on the procedures as a group, save your workspace procedures into a single disk file. When you subsequently load the file, BASIC09 automatically loads all of the procedures.

Modules of Other Languages

BASIC09 can incorporate procedures from other languages, such as Pascal, C, or assembly language. Several users can then share the procedures.

Executing Procedures

You execute or *run* programs from the command mode. When you enter a procedure, BASIC09 compiles it. This means that the procedure is ready for execution as soon as you exit the edit mode. For instance, if you create a program named Greeting, you can execute it by typing from the command mode:

```
run greeting [ENTER]
```

Leaving BASIC09

There are two ways you can exit from BASIC09:

- At the `B:` prompt, type:

  ```
  bye [ENTER]
  ```

- Or, at the `B:` prompt, press [CTRL][BREAK].

When you use either method, the OS-9 prompt appears immediately indicating that the operating system is waiting for a command.

> **Note:** When you exit BASIC09, you lose all procedures residing in the workspace. Be sure to save them on disk before leaving BASIC09.

The Keyboard and BASIC09

You can use some keys and *key sequences* to produce special characters and to accomplish special BASIC09 functions. You initiate a key sequence by pressing one key and holding it down while pressing a second key. The following list summarizes your keyboard's special functions:

[ALT]	Produces graphic characters. Press [ALT] *char* where *char* is a keyboard character).
[CTRL]	A control key that you use with other keys. (See below.)
[BREAK] or [CTRL][E]	Stops the current program execution and returns to the `B:` prompt in BASIC09's command mode.
[←] or [CTRL][H]	Moves the cursor back one space.
[CTRL][_]	Produces an underscore character.
[CTRL][,]	Produces a left brace ({).
[CTRL][.]	Produces a right brace (}).
[CTRL][#]	Produces a tilde (~).
[CTRL][/]	Produces a backslash (\).
[CTRL][BREAK]	Performs an ESCAPE function and sends an end-of-file message to a program receiving keyboard input. To be recognized, [CTRL][BREAK] must be the first thing typed on a line.
[SHIFT][BREAK]	Stops execution of a program and causes BASIC09 to enter the Debug mode.
[CLEAR]	Displays the next window.
[SHIFT][CLEAR]	Displays the previous window.
[SHIFT][←] or [CTRL][X]	Deletes the current line.
[CTRL][0]	Activates or deactivates the shift lock function.
[CTRL][1]	Produces a vertical bar (|).
[CTRL][7]	Produces an up arrow (↑).
[CTRL][8]	Produces a left bracket ([).
[CTRL][9]	Produces a right bracket (]).

CTRL A	Redisplays the last line typed, and positions the cursor at the end of the line, but does not process the line. Press ENTER to process the line, or edit the line by backspacing. If you edit, press CTRL A again to display the edited line.
CTRL D	Redisplays the current command line.
CTRL W	Temporarily halts video output. Press the space bar to resume output.
ENTER	Performs a carriage return or executes the current command line.

Sample Session

Although BASIC09 has several functions or modes, they all work together to make programming as simple as possible. The easiest way to learn how BASIC09 and its functions operate is to write and run a program. This chapter provides sample statements and instructions to help you learn how to use BASIC09.

To create and execute a program:

1. Load BASIC09 and enter the edit mode.
2. Type the BASIC program.
3. Enter the system mode and test the program's execution.
4. Debug the program. (Correct any programming errors.)
5. Save the completed program on disk.
6. Load the program into memory and use it.

To begin the program, execute BASIC09. To be sure you have enough room in which to work, reserve a workspace of 10,000 bytes by typing:

```
basic09 #10K [ENTER]
```

The BASIC09 system mode prompt, `B:`, appears after the copyright message. In the system mode, you can do such things as save and load procedures, change workspace size, and rename and delete procedures.

Creating a Procedure

To write procedures, you must be in the edit mode. You get there by typing:

```
e [ENTER]
```

This causes the screen prompt to change to `E:`, and the screen displays:

```
PROCEDURE Program
```

Because you didn't give a program name when you entered `e`, BASIC09 selects the name Program for you. Now, you must write the code to make Program do something.

Commands and Program Lines

There are two responses you can give at the edit mode prompt. You can type an edit command, or you can type a program line. If you type a program line, **you must type a space as the first character in the line.** If you type an edit command, do **not** precede it with a space. To make listings easier to read, this manual uses the symbol ␣ to indicate spaces before every line. It also uses the ␣ symbol in some procedure lines to indicate the correct number of spaces needed. Whenever you see either a space or a ␣ symbol in a procedure you are typing, press the space bar.

To type the procedure in this chapter, begin each line at the `E:` prompt. After typing a line, check it for mistakes. If you make a mistake, use [←] to move the cursor back. Correct the mistake. Then, type the remaining portion of the line. If there are no mistakes, press [ENTER].

BASIC09 checks each line when you press [ENTER]. If you make a mistake in syntax or form, BASIC09 displays an error message. An arrow points to the place in the program line where the error occurred, and a message number indicates the type of error. Refer to Appendix A for an explanation of the error codes.

If, after you enter a line, you find that you made a mistake, type `d` [ENTER] to delete the line. Then, retype the line. Later, the manual tells you how to change text in existing lines.

The following program helps you do a bit of arithmetic. To get a feel for BASIC09, type and execute the program as directed. Remember, when you see either a space or ␣, press the space bar.

```
␣DIM NUMBER1,NUMBER2:INTEGER
␣INPUT "Type Number...";NUMBER1
␣INPUT "Type another....";NUMBER2
␣PRINT "The sum of the numbers is... ";
␣PRINT NUMBER1 + NUMBER2
␣END
```

Executing a Procedure

To execute the procedure, quit the edit mode by typing `q` [ENTER]. The compiler further processes your procedure, and the `B:` prompt reappears. To execute the program, type:

```
run [ENTER]
```

Type in numbers when asked, and the procedure produces their sum. If you want to save the program on disk, the next chapter tells you how. Chapter 4 introduces several other edit mode commands to search, display, insert, and change programs lines and text.

Chapter 3

The System Mode

The BASIC09 *command interpreter* processes system commands. At the `B:` prompt, you can enter system commands in either upper- or lowercase letters. Some commands operate on the procedures in the workspace. Others provide functions independent of any procedures. Following is a list of all system commands and their purposes.

Command	Function
$	Calls the shell command interpreter to execute an OS-9 command.
BYE or [CTRL] [BREAK]	Returns you to the OS-9 system or to the program that called BASIC09.
CHD	Changes the current OS-9 data directory.
CHX	Changes the current OS-9 execution directory.
[ENTER] **or DIR**	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or E	Enters the procedure editor-compiler mode.
KILL	Erases one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a disk file into the workspace.
MEM	Displays in bytes the current workspace size, or reserves a specified amount of memory for the workspace.
PACK	Condenses (compiles) one or more procedures.
RENAME	Changes a procedure's name.
RUN	Causes a procedure in the workspace to execute.
SAVE	Writes one or more procedures to disk.

Renaming Procedures

BASIC09's RENAME function is important for two reasons: First, it lets you load into the workspace procedures that have the same name. After you rename the workspace procedure you can load the second file. Second, if you let BASIC09 use the default procedure name, "Program," you can rename the procedure before saving it to disk. By doing this, you avoid writing over—and destroying—an existing procedure file.

To change the name of the procedure you created in the previous chapter from Program to Add, type:

```
rename program add [ENTER]
```

Listing Procedure Names

You can use the DIR command to see if RENAME worked properly. DIR displays the names and sizes of all procedures in memory. Because programmers use this command frequently, the system recognizes a shorthand call. Instead of typing `dir` [ENTER], you only need to press [ENTER]. This displays a table of the procedures in the following format:

```
 Name              Proc-Size      Data-Size
 *add                   182             32
  add1                  217             42
  add2                  218             42
2198 free
```

Proc-Size refers to the number of memory bytes required for the procedure. *Data-Size* refers to the number of memory bytes required for the procedure's variables and data structures. The asterisk indicates the current procedure. System commands act on the current procedure unless you indicate otherwise.

The last line of the DIR display tells you how many free bytes of memory remain in the BASIC09 workspace.

Listing Procedures

You can use the LIST command to view procedure lines. To display the current procedure, type:

```
list [ENTER]
```

For example, this is the listing of a procedure named Alpha.bak:

```
PROCEDURE Alpha_bak
 0000     DIM A:STRING
 0007     DIM T:INTEGER
 000E
 000F     PRINT "Here is the alphabet
          backwards:"
 0032     PRINT
 0034     FOR T=90 TO 65 STEP -1
 004A       PRINT CHR$(T);
 0051       PRINT " ";
 0057     NEXT T
 0062     PRINT
 0064     PRINT
 0066     END
```

When you list a BASIC09 procedure, the system precedes each line with a *relative* storage address. The relative address of the first procedure line is always 0. In the previous example, the beginning address of the second procedure line in the workspace is 07 units from the beginning. The beginning address of the third line is 0E hexadecimal (14 decimal) storage units from the procedure beginning.

These I-Code addresses provide a way for the compiler to let you know where it finds an error when one occurs.

Because BASIC09 compiles programs into I-Code, it must *disassemble* them before it can display them on the screen. This means that the lines might not look exactly as typed. For instance, BASIC09 converts lowercase *keywords* (command names) to uppercase. BASIC09 also eliminates some spaces. If your program uses control statements such as IF/THEN, FOR/NEXT, and LOOP/ENDLOOP, the lines in these decision making or looping structures are indented as shown in the Alpha.bak example. Regardless of the appearance of your listed procedures, they execute correctly if you type their commands correctly.

Listing Procedures to a File

There might be times when you want to send a formatted procedure listing, including I-Code addresses, directly to a file. You can do this using OS-9's redirection symbol, >. To save the Alpha.bak procedure on a file named Alpha.list in the current data directory, type:

```
list alpha.bak >alpha.list [ENTER]
```

If you have several procedures in the workspace and want to list more than one to a disk file, separate the procedure names with commas, like this:

```
list alpha.one,alpha.two,alpha.three >alpha.all
[ENTER]
```

In both of the preceding cases, the system creates the Alpha.list file and stores the specified listings in it. If you use a file name that already exists, BASIC09 displays the prompt:

```
Rewrite?:
```

If you press [Y], the system destroys the original file and overwrites it with the new listing. If you press [N], the LIST process terminates.

If you wish to list a procedure, or group of procedures, to a file that is not in the current data directory, be sure to specify the complete pathlist, such as:

```
list alpha.bak > /d1/programs/alpha.rev [ENTER]
```

Listing Procedures to a Printer

In the same manner as you list procedures to a disk file, you can list one or more procedures to your printer. Make certain your printer is connected and turned on, then again use the redirection symbol, but this time specify the printer device, like this:

```
list alpha.bak >/p [ENTER]
```

Or:

```
list alpha.one,alpha.two,alpha.three >/p [ENTER]
```

Using a Wildcard

Using the OS-9 wildcard, *, you can list all procedures in the workspace. For instance, if the procedures Alpha.one, Alpha.two, and Alpha.three exist, list them to the screen by typing:

```
list* ENTER
```

Send the list to a file by typing:

```
list* alpha.all ENTER
```

Or send the list to your printer by typing:

```
list* /p
```

Note: When you use the wildcard, the name of the file or device to receive the listing immediately follows the LIST* command. Do not use the redirection symbol.

Saving Procedures

You can save one or more procedures to disk using the SAVE command. Unlike LIST, SAVE does not include relative addresses. However, the syntaxes for the SAVE and LIST commands are identical. To save the procedure Alpha.bak to the current data directory, type:

```
save alpha.bak alpha.bak ENTER
```

If Alpha.bak is the current procedure, you can save it in a file named Alpha.bak by typing `save` ENTER.

To save all of the procedures in the workspace to a file named All.programs in the current data directory, type:

```
save* All.programs ENTER
```

As with LIST, to save one or more procedures in a file that is not in the current data directory, make sure you specify a complete pathlist.

To save all the files in the workspace to a disk file with the same name as the current procedure, type `save*` ENTER.

If the disk file you specify does not exist, BASIC09 creates it. If it does exist, the system displays the prompt:

```
Rewrite?:
```

Press [Y] to write over the old file with the specified file. The old file is destroyed.

Press [N] to terminate the SAVE operation.

Loading Procedures

To load a saved procedure back into BASIC09's workspace, use the LOAD command and specify the appropriate pathlist. For instance, if your current directory is still the directory containing Alpha.bak, load the procedure by typing:

```
load alpha.bak [ENTER]
```

To load Alpha.bak from the PROGRAMS directory on Drive /D1, type:

```
load /d1/programs/alpha.rev [ENTER]
```

You can run and edit a loaded procedure in exactly the same manner as you would a procedure you created.

You can load any number of procedures into the workspace as long as your computer has sufficient memory. However, be careful that you do not load a procedure with the same name as a procedure already existing in the workspace. If you do, the new procedure overwrites (destroys) the original procedure. You can rename workspace procedures to avoid this problem.

Deleting Procedures from the Workspace

You can clear the workspace of one or more procedures using the KILL command. For instance, to remove Alpha.bak from the workspace, type:

```
kill alpha.bak [ENTER]
```

To remove more than one procedure from the workspace, separate the procedure names with commas. To delete Alpha.one and Alpha.two, type:

```
kill alpha.one,alpha.two [ENTER]
```

To clear the entire workspace, regardless of the number of procedures it contains, use the BASIC09 wildcard, *. Type:

```
kill* [ENTER]
```

Changing Directories

You change working directories in BASIC09 and OS-9 in the same manner, by using the CHD and CHX commands. CHD changes the data directory, and CHX changes the execution directory.

BASIC09 saves files in, or loads files from, the data directory, unless you specify differently in the command pathlist. It stores packed procedures in, or loads PACKed procedures from, the execution directory, unless you specify differently in the command's pathlist.

Also, if you want to access OS-9 commands from BASIC09, the system first looks for the commands in memory. If they are not there, it looks for them in the execution directory, unless you specify differently.

If your data directory is the ROOT directory, and you wish to change to a directory named PROGRAMS that is a subdirectory of the ROOT directory, type the following command from the command mode `B:` prompt

```
chd programs [ENTER]
```

If your current execution directory is the system's CMDS directory, and you want to change to a CMDS directory in the subdirectory BASIC, type:

```
chx basic/cmds [ENTER]
```

Whenever you change to a directory other than an immediate subdirectory, specify a complete pathlist.

Executing OS-9 Commands

BASIC09 lets you use OS-9 commands at any time from the system mode. To do so, precede the command with a dollar sign ($). For instance, to look at the current data directory, type:

```
$dir [ENTER]
```

To view the current execution directory, type:

```
$dir x [ENTER]
```

All OS-9 commands are available, and you can copy files, format diskettes, list files, or use any other functions from the system mode. The only restriction is that your computer must have enough free memory to handle the command you call. If you find that there is not enough memory, try using the MEM command to reduce reserved memory. Then, try the command again.

Auto-Execute Procedures

The BASIC09 compiler makes two passes through the procedures you write. When you enter the command, the compiler performs an initial compilation, checking for any syntax errors. When you leave the edit mode, the system compiles the procedure a second time and checks for any programming errors. With the PACK command, you can further compile your procedures so that they are smaller and execute even faster.

PACK causes an extra compiler pass that removes names, line numbers, and non-executable statements. **Before packing a procedure, be sure you save it. Unless you do so, you cannot make further changes to the procedure.**

Once you pack a file, you cannot list or edit the packed version. However, if you save the procedure to disk before packing, you can still list and edit the original file, then pack it again.

When you save a packed procedure on disk, BASIC09 does not normally store it in the data directory. Because the procedure is now *executable*, the system stores it in the current execution directory.

For instance, to convert Alpha.bak to a packed procedure in the execution directory, type:

```
pack alpha.bak [ENTER]
```

If you want to save a packed procedure under a different filename, use the OS-9 redirection symbol:

```
pack alpha.bak > backwards [ENTER]
```

After packing a procedure, you can delete it from the workspace. If you then run it, BASIC09 automatically loads the file from disk and executes it.

The following is a sequence of commands that demonstrate packing and executing a procedure named Alpha.bak:

`pack alpha.bak` ENTER	packs the procedure and stores it in the execution directory.
`kill alpha.bak` ENTER	deletes the procedure from the workspace.
`run alpha.bak` ENTER	loads the file into memory *outside* the workspace and executes it.
`kill alpha.bak` ENTER	deletes the module from memory

You do not need to kill the file immediately after execution, but until you do, the file reduces available memory.

The Edit Mode

You briefly used the BASIC09 built-in editor to create the Add procedure in Chapter 2. In addition to the features you learned there, the editor has other important functions.

Although you can use any text editor or word processor to write BASIC09 procedures, the BASIC09 editor offers two handy features:

- It is both *string* and *line number* oriented. You can search for strings of characters, and replace them, and you can reference text with optional line numbers.
- It interfaces with the compiler and *decompiler*. This feature lets BASIC09 check continuously for syntax errors and enables you to use procedures that conserve memory.

Edit Commands

The following is a summary of the edit commands:

Command	Function
ENTER	Moves the edit pointer to the next line. Causes a command to execute.
+number	Moves the edit pointer ahead *number* lines.
+*	Moves the edit pointer to the last line.
-number	Moves the edit pointer back *number* lines.
-*	Moves the edit pointer to the first line.
text	Inserts an unnumbered text line before the current line.
ntext	Inserts the line numbered *n* in its correct numeric position.
n	Moves the edit pointer to the line numbered *n*.
c/*str1*/*str2*/	Changes the next occurrence of *str1* to *str2*.

Command	Function
c*/*str1*/*str2*/	Changes all occurrences of *str1* to *str2*.
d	Deletes the current line.
d*	Deletes all the lines in the procedure.
l	Lists the current procedure line.
l*	Lists all the procedure lines.
q	Terminates the edit session.
r	Renumbers lines beginning at the current line in increments of 10.
r*	Renumbers all lines in increments of 10.
r *n*	Renumbers lines beginning at Line *n* in increments of 10.
r *n1 n2*	Renumbers lines beginning at Line *n1* in increments of *n2*.
s /*string*/	Searches for the first occurrence of *string*.
s* /*string*/	Searches for all occurrences of *string*.

Using the Editor

The easiest way to understand the edit commands is to use them. The following sections show you the functions of BASIC09 edit mode.

The manual uses line numbers in the following procedure to acquaint you with all the functions of the editor. Remember, however, line numbers are not required with BASIC09. Procedures and programs without line numbers are shorter, faster, and easier to read.

First, you need a procedure with which to work. Position yourself in the system mode. Then, type this line:

```
e prose ENTER
```

Now, type the following. (Remember, the small rectangle represents a space.)

```
□100 DIM PHRASES(30):STRING
□120 FOR T=1 TO 30
□130 READ PHRASES(t)
□140 NEXT T
□160 PRINT
□170 FIRST=RND(10)
□180 SECOND=RND(9)+11
□190 THIRD=RND(9)+21
□200 PRINT PHRASES(FIRST);
□210 PRINT PHRASES(SECOND);
□220 PRINT PHRASES(THIRD);
□240 PRINT
□300 DATA "Love□","An orange□",
"Humanity□","A kiss□"
□310 DATA "A dark cloud□","A goose feather□",
"A Popsicle□"
□320 DATA "Home cooking□","Cold pizza□",
"Rock n' Roll□"
□330 DATA "is charming like□","makes me dream of□"
□340 DATA "is as sticky as□","can ooze
like□","smells like□"
□350 DATA "can be as tough to forget as□","can
hurt like□"
□360 DATA "can be as cynical as□","makes a mockery
of□"
□370 DATA "drives me as crazy as□"
□380 DATA "a sticky lollipop.","a web of
intrigue."
□390 DATA "castor oil.","a chocolate bath.","a
broken toe."
□400 DATA "honey and things.","personal
defeat.","a wet diaper."
□410 DATA "strange happenings.","a pennyless
purse."
```

When you finish typing the procedure, type `q` [ENTER] to return to the system mode. Now you can test the program by typing either:

```
run [ENTER]
```

or

```
run prose [ENTER]
```

After trying the procedure, return to the edit mode by typing `e` [ENTER].

After displaying the procedure's name, the editor displays Line 100 preceded by an asterisk. The asterisk lets you know which line is the *current line* (or the line at which the edit pointer is located).

Searching Through a Procedure

You can examine a procedure in three ways:

- Press [ENTER] to display the procedure one line at a time.
- Skip through the procedure to a particular line.
- List part or all of the procedure to the screen.

When you use either of the first two methods, the line you select to display becomes your current line. When you use the third method, the current line does not change.

Using [ENTER]

If you are still positioned at Line 100, but want to examine the first line of data, Line 300, press [ENTER] 12 times to move down.

Using the Plus Sign to Move Forward

Another method of moving to a specific line is to type a plus sign followed by the number of lines you need to advance to get there. Positioned at Line 100, you can type:

```
+12 [ENTER]
```

Whether you press [ENTER] or use the plus sign, the last line displayed is now your current line.

Accessing a Line Using the Line Number

The third way to move to a particular line is to type the line number, followed by ENTER. For instance, to jump back to Line 100, type:

```
100 ENTER
```

The editor displays Line 100 and makes it your current line.

Using the Minus Sign to Move Backward

In the same manner that you move forward in the procedure using the plus sign, you can move backward using the minus sign, or hyphen.

Type `300` ENTER to return to Line 300. To display Line 240 and make it your current line, type:

```
- ENTER
```

To display Line 190 and make it your current line, type:

```
-4 ENTER
```

The Global Symbol

The BASIC09 editor also makes use of the asterisk as a global symbol. For instance, following a command with an asterisk causes that command to affect the entire procedure.

This feature lets you move quickly to the beginning and end of the procedure. To return to Line 100, the first line, type:

```
-* ENTER
```

To move to the end of the procedure, past all the numbered lines, type:

```
+* ENTER
```

Using LIST

The LIST command lets you select one or more lines for display on your screen. To see this, make the first line your current line, then type:

```
1 [ENTER]
```

To list one or more lines, type the LIST command followed by the number of lines you want displayed. For instance, typing `15 [ENTER]` causes the current line and four others to appear on the screen, as shown in the following sequence of commands and the resulting display:

```
-* [ENTER]
15 [ENTER]
PROCEDURE Prose
100 DIM PHRASES(30): STRING
120 FOR T=1 TO 30
130READ PHRASES(T)
140 NEXT T
160 PRINT
```

You can also use LIST with the BASIC09 global symbol, *. Typing an asterisk after the LIST command produces a listing of the entire procedure.

Deleting Lines

Earlier, the manual showed that you can delete the current line by typing `d [ENTER]`. Because this is such a simple process, be sure you don't do it by accident. Removing the wrong line, or too many lines, is very frustrating in a complex procedure.

You can also remove a group of lines from a procedure by typing `d`, followed by the number of lines you want to delete. This command deletes the current line and specified following lines. Again, be careful.

You can remove all of the lines in a procedure by using the global symbol, *. Typing `d* [ENTER]` erases all procedure text. However, the procedure name still resides in the workspace. To delete an entire procedure, including the name, use the KILL command from the system mode.

If you decide you don't like the nouns used in the DATA lines of the Prose procedure, erase all of the DATA lines containing nouns (Lines 300-320) and replace them. To do so, make Line 300 your current line by typing:

```
300 [ENTER]
```

Then type:

```
d [ENTER]
```

Line 300 disappears and Line 310 takes its place as the current line.

An alternate method of deleting the DATA lines uses only one command. To delete Lines 300 through 410, follow the DELETE command with the number of lines you want to remove—in this case, three:

```
d3 [ENTER]
```

Lines 300, 310, and 320 disappear. Line 330 becomes the current line. Move back a line to check that the deletions worked. The line numbers now skip from 240 to 330.

Now, you need new nouns for the procedure. Type them in the same style as the old lines, such as:

```
□300 DATA "A Telephone□","A tickle□",
"A girl□","A boy□"
□315 DATA "Bad luck□","Money□","A bad bet□",
"A lumpy bed□"
□320 DATA "A deep thought□","Sunlight□"
```

Save a copy of your procedure to disk by exiting the editor and using the SAVE command. Then return to the edit mode and try the global delete by typing:

```
d* [ENTER]
```

Changing Text

Using CHANGE tells the editor to search for existing text and replace it with new text. CHANGE, like DELETE, can easily cause unwanted results if you are not careful.

The CHANGE command requires that you use *delimiters* to separate the command from the search text, and to separate the search text from the new text. You can select any of the following characters for a delimiter, as long as it does not appear in either the search text or the new text:

! # % ^ & ( ) - + = { } [ ] " " < > , . ? / \ |

Do not use the global symbol (*) for search and replace operations. This manual uses a slash (/) as the CHANGE delimiter.

The following steps outline the correct use of CHANGE:

1. Position the editor either before or on the line in which you want to make a change.
2. Type `c` (for CHANGE). Do not use a preceding space.
3. Type a delimiter character, such as `/`.
4. Type the characters to be changed, following them with the delimiter.
5. Type the new text, followed by the delimiter.
6. Press [ENTER].

Note: It is a good idea to turn on OS-9's upper- and lowercase function before attempting change or search operations. If you do not, you cannot tell whether the text you want to find is upper- or lowercase, or some combination of the two. If you type the wrong case, the change or search fails.

In case you didn't notice when typing the procedure, Line 410 contains an incorrectly spelled word, pennyless. To correct this error, type the following:

```
c/pennyless/penniless/ [ENTER]
```

Immediately, the editor displays Line 410, with pennyless changed to penniless.

Suppose you decide to change the number of sentence combinations available in Prose. The procedure now has 30 data entries. If you add five subjects, five verb phrases, and five objects, the procedure also needs other changes (for instance, the DIM statement in Line 100, the loop size in Line 120, and the RND statements in Lines 170 through 190).

A quick way to change the number 30 in Lines 100 and 120 is to use CHANGE's global function. To change all occurrences of 30 to 45, position the editor at Line 100, and type:

```
c*/30/45/ [ENTER]
```

Use the CHANGE and global CHANGE functions to adjust the RND statement values in Lines 170, 180, and 190.

As well as making changes, you can use the CHANGE command to quickly delete portions of text within a line. To do this, type delimiters without new text, in this fashion:

```
c/□feather// [ENTER]
```

This command changes the text `A goose feather` in Line 210 to `A goose`.

Searching for Text

The editor's SEARCH command, S, works in the same manner as the CHANGE command. However, SEARCH only requires you to specify a block of text to find.

With SEARCH, you use delimiters to enclose the text to find. To test the function, position the editor at the beginning of text by typing:

```
-* [ENTER]
```

Now, search for the word phrases, by typing:

```
s/phrases/ [ENTER]
```

The screen displays:

```
*0000 100 DIM phrases(30):STRING
```

To find all occurrences of phrases throughout the procedure, use the global symbol. Type:

```
s*/phrases/ [ENTER]
```

Renumbering Lines

The RENUMBER command, `R`, reorders all numbered lines and all references to numbered lines. You can give RENUMBER either one or two parameters. The first is the beginning line number. The second is the increment you want. The default increment is 10.

For instance, the Prose procedure line numbers skip from Line 100 to Line 120. You can renumber the entire procedure by moving the editor to Line 100, and then typing:

```
r 10 [ENTER]
```

To change the numbering to increments of 5, beginning at Line 100, type:

```
r 100,5 [ENTER]
```

You can also change line numbering in portions of the procedure. To do this move the editor to the line where you want the new numbering to begin. Then, type in the new parameters. To renumber Line 100 as Line 200 and continue with increments of 10, position the editor at Line 100. Then, type:

```
r 200,10 [ENTER]
```

If you are not positioned at the first line of a procedure, but you wish to renumber all lines, you can use the global symbol to do the job. From anywhere in the procedure, type:

```
r* 100,10 [ENTER]
```

This renumbers the entire procedure in increments of 10.

Adding Lines

There are two ways to add new lines to a procedure. You can:

- Position the editor one line below the position for the new line. Then, type the new line and press [ENTER]. When inserting lines without numbers, be sure to type a space as the first character of the line to tell the editor that the following text is a new procedure line.
- Type a new line, giving it a line number that falls between two existing line numbers.

The following procedure adds more choices to the Prose program. It also adds a feature that lets you press [ENTER] for additional output, rather than having to rerun the procedure. Use the information presented in this section to help you insert the new lines into your program. Because you must change some lines, as well as add lines, the following listing includes the entire procedure.

Referring to the original Prose listing, the lines to change are: 100, 120, 170, 180, and 190.

The lines to add are: 110, 150, 230, 250, 260, 270, 305, 325, 372, 374, 376, 420, 430.

```
PROCEDURE prose2
100 DIM PHRASES(45):STRING
110 DIM RESPONSE:STRING
120 FOR T=1 TO 45
130 READ PHRASES(t)
140 NEXT T
150 REPEAT
160 PRINT
170 FIRST=RND(15)
180 SECOND=RND(14)+16
190 THIRD=RND(14)+31
200 PRINT PHRASES(FIRST);
210 PRINT PHRASES(SECOND);
220 PRINT PHRASES(THIRD);
230 PRINT
240 PRINT
250 PRINT "□□□□□□□□□□Press ENTER for another
witticism..."
260 INPUT "□□□□□□□□□□Or press the SPACEBAR and press
ENTER to end...",RESPONSE
270 UNTIL RESPONSE>""
300 DATA "Love□","An orange□","Humanity□",
"A kiss□"
305 DATA "A computer□","A book□","Misery□"
310 DATA "A dark cloud□","A goose feather□",
"A Popsicle□"
320 DATA "Home cooking□","Cold pizza□",
"Rock n' Roll□"
325 DATA "Snow in June□","A glass house□"
330 DATA "is charming like□","makes me dream of□"
```

```
340 DATA "is as sticky as□","can ooze like□",
"smells like□"
350 DATA "can be as tough to forget as□",
"can hurt like□"
360 DATA "can be as cynical as□",
"makes a mockery of□"
370 DATA "drives me as crazy as□"
372 DATA "can bother me like□","blackens my hopes
like□"
374 DATA "can tickle me like□","can be as funny
as□"
376 DATA "has the effect of□"
380 DATA "a sticky lollypop.","a web of
intrigue."
390 DATA "castor oil.","a chocolate bath.","a
broken toe."
400 DATA "honey and things.","personal
defeat.","a wet diaper."
410 DATA "strange happenings.","a penniless
purse."
420 DATA "a slimy snake.","a bad habit."
430 DATA "a bad memory chip.","a good fight.","a
silly friend."
```

The Next Step

Even the best programmers make mistakes—a lot of them. BASIC09 provides a way to catch programming mistakes quickly and correct them. The next chapter tells you about BASIC09's powerful debugging functions.

Chapter 5

The Debug Mode

The term *debug* refers to the process of finding programming errors and correcting them. BASIC09's debugging features include *symbolic* debugging capabilities that let you examine variable values and test and manipulate procedures.

With Debug, you can:

- Examine and change variables.
- Trace procedure execution. Debug lets you execute procedures and watch them run in slow motion.
- Pause procedure execution.
- Resume procedure execution.
- Set procedure *breakpoints* that automatically switch to the debug mode.
- Select the use of degrees or radians for trigonometric functions.
- Perform calculations.
- Call OS-9 system commands.

Entering the Debug Mode

You enter Debug:

- Automatically, whenever an error occurs during the execution of a procedure (unless you have included an ON ERROR GOTO statement to handle the error).
- Automatically, when a procedure executes a PAUSE statement.
- When you press CTRL C during the execution of a procedure.

You can tell when BASIC09 enters the Debug mode by the appearance of the `D:` prompt. When you see `D:`, followed by the cursor, Debug is waiting for your command.

The following is a reference of all the Debug commands and what they accomplish:

Command	Function
$	Calls OS-9's command shell interpreter to run a program or an OS-9 command. From the Debug prompt, type `$`, followed by the name of the program or command you want to execute. Example: `$list procedure_one` ENTER
BREAK	Sets a breakpoint immediately before the specified procedure. Use this command to re-enter Debug when one procedure calls another. If you have three procedures that call each other—Proc1, Proc2, and Proc3—and Proc3 does not seem to pass the correct values to Proc2 when it returns, set a breakpoint at Proc2. This causes BASIC09 to enter Debug before re-entering Proc2. You can then check your variable values. You can use one breakpoint for each active procedure. Debug removes breakpoints immediately after encountering them. A procedure must run before you can set a breakpoint in it. Use BREAK to stop execution when a called procedure returns to a procedure previously executed. Example: `BREAK proc2` ENTER
CONT	Causes procedure execution to continue. Example: `cont` ENTER
DEG/RAD	Selects either degrees or radians as the unit of measurement for trigonometric functions. DEG and RAD affect only the current procedure. Examples: `deg` ENTER `rad` ENTER

DIR

Displays the name, size, and variable storage requirements of each procedure in the workspace. The current working procedure has an asterisk before its name. All packed procedures have a dash before their names. DIR also shows the available memory in the workspace.

If you provide a pathlist, DIR sends its data to the specified file.

Example: `dir` [ENTER]
`dir procedures` [ENTER]

Q

Terminates execution of the procedure, closes any open paths, and exits to the System mode.

Example: `q` [ENTER]

LET

Assigns a new value to a variable. You must specify variable names that are already initialized by your program. In the Debug mode, you cannot use LET to copy one array to another array as you can in BASIC procedures.

Example: `let a := 0` [ENTER]
`let fruit := "oranges"` [ENTER]

LIST

Displays a source listing of the suspended procedure. The display is formatted and includes I-code addresses. An asterisk appears to the left of the last executed statement.

Example: `list` [ENTER]

PRINT

Displays the values of variables used in the suspended procedure. You cannot introduce new variable names in the Debug mode, and you cannot display array structures.

Example: `print fruit` [ENTER]

STATE

Lists the *nesting* order of active procedures. STATE displays the highest-level procedure at the bottom of the calling list. The lowest-level procedure is the suspended procedure.

Example: `state` [ENTER]

STEP

Causes execution of the suspended procedure in specified increments. For example, typing `STEP 5` [ENTER] executes the equivalent of the next five statements. If you enter STEP without an increment value, the step rate is 1.

Using STEP with the trace function lets you observe the source lines as they execute.

Because compiled I-code contains actual statement memory addresses, the *top* or *bottom* statements of loop structures execute only once. For example, in FOR/NEXT loops, FOR executes once, and the statement following FOR appears to be the top of the loop.

TRON/TROFF

Turns on or turns off the trace function. Trace on (TRON) causes the system to reconstruct the compiled code of each statement line into source code. Debug displays the source code before the statement is executed. If the statement causes the evaluation of one or more expressions, Debug displays each result following the statement. The result is preceded by an equal sign.

The trace function is local to the current procedure. If the suspended procedure calls another procedure, Debug suspends the trace function until control returns to the original procedure. However, once you turn on trace for a procedure, it continues in effect until you turn it off. This means that if you turn trace on in a called procedure, and another procedure subsequently calls it, trace continues to display the called procedure's operations.

Example: `tron` [ENTER]
`troff` [ENTER]

When Things Go Wrong

Programming errors show up in two ways. Either your procedure produces incorrect results, or it terminates prematurely.

In the first instance, you can stop your procedure and enter Debug by pressing [CTRL][C].

However, sometimes your program executes too quickly to allow you to stop it at the appropriate place. In this case, you can use the Edit mode to insert a PAUSE command where you wish the procedure to stop. PAUSE causes the procedure to halt execution and enter the Debug mode.

Once in Debug, you can use the PRINT command to examine the procedure variables. You can use LET to manipulate the variable values to determine where the error or errors occur. Perhaps you forgot to initialize a variable or forgot to increase a loop counter.

Using the Trace Function

Sometimes, errors are more difficult to discover. If so, the next step is to use the trace function. To do this, type:

```
tron [ENTER]
```

Now press [ENTER]. Each time you press [ENTER], Debug executes one line of the procedure. You can see the original source statement, and if an expression is evaluated, Debug prints the result of the expression, preceded by an equal sign.

In this manner, you can step through the entire procedure, or any part of it, examining variable values as you go.

What About Loops?

The STEP command is helpful if you find yourself tracing the operation of a loop. Once you determine that the loop works correctly, you can avoid tedious, step-by-step repetitions by turning trace off and using STEP to quickly run through the loop. Then, turn trace back on and resume single-step debugging. For instance, type:

```
troff [ENTER]
step 200 [ENTER]
tron [ENTER]
```

In Multiple Procedures

Although the trace function is local to a procedure, you can pause and turn on the trace function in as many procedures as you wish. Trace continues to operate in each procedure until you turn it off using TROFF.

To cause a procedure to halt execution when it is called by another procedure, use the BREAK command.

Data and Variables

Data Types

Data is information on which a computer performs its operations. Data is always numeric but, depending on your computer application, it can represent values, symbols, or alphabetic characters. This means that the same items of *physical* data can have very different *logical* meanings, depending on how a program interprets it.

For instance, 65 can represent:

- A numeric value to be used in a calculation.
- The location of a memory address.
- The *offset* of a memory location.
- The two character symbols 6 and 5.
- The character A in the ASCII table. ASCII is the abbreviation for the American Standard Code for Information Interchange.

Because of the differences in how BASIC09 uses data, the system lets you define five types of data. For instance, there are three ways to represent numbers. Each has its own advantages and disadvantages. The decision to use one way or another depends on the specific program you are developing. The five BASIC09 data types are byte, integer, real, string, and Boolean.

In addition to the preceding data types, there are *complex data types* you can define. The manual discusses complex data structures at the end of this chapter.

The *byte*, *integer*, and *real* data types represent numbers.

The *string* data type represents character data (alphabet, punctuation, numeric characters, and other symbols). The default length of strings is 32 characters. Using the DIM statement, you can specify strings of both longer and shorter lengths.

The *Boolean* data type represents the logical value, TRUE or FALSE.

You can create arrays (lists) of any of these data types with one, two, or three dimensions. The following table shows the data types and their characteristics:

Type	Allowable Values	Memory Requirements
BYTE	Whole numbers (0 to 255)	One byte
INTEGER	Whole numbers (-32768 to 32767)	Two bytes
REAL	Floating point ($\pm 1*10^{\pm 38}$)	Five bytes
STRING	Letters, digits, punctuation	One byte per character
BOOLEAN	True or false	One byte

Real numbers appear to be the most versatile. They have the greatest range and are floating point. However, arithmetic operations involving real numbers execute much more slowly than those involving integer or byte values. Real numbers also take up considerably more memory storage space than the other two numeric data types.

Arithmetic involving byte values is not appreciably faster than arithmetic involving integers, but byte data conserves memory.

If you do not specify the type of variable (a symbolic name for a value) in a DIM statement, BASIC09 assumes the variable is real.

The Byte Data Type

Byte variables hold unsigned eight-bit data (integers in the range 0 through 255). Using byte values in computations, BASIC09 converts the byte values to 16-bit integer values. If you store an integer value that is too large for the byte range, BASIC09 stores only the least-significant eight bits (a value of 255 or less), and does not return an error.

The Integer Data Type

Integer variables require two bytes (16 bits) of storage. They can fall in the range -32768 to 32767. If a calculation involves both integer values and real values, BASIC09 presents the result of the calculation as a real number.

You can also use hexadecimal values in integer data. To do so, precede the value with the dollar sign ($). For instance, to represent the decimal value 199 as hexadecimal, type `$C7`. The hexadecimal value range is $0000 through $FFFF.

If you give an integer variable a value that is outside the integer range (greater than 32767 or less than -32768), BASIC09 does not produce an error. Instead it *wraps around* the value range. For instance, the calculation 32767 + 1 produces a result of -32768.

This means that numeric comparisons made on values in the range 32768 through 65535 deal with negative numbers. You should limit such comparisons to tests for equality or non-equality. Functions such as LAND, LNOT, LOR, and LXOR use integer values but produce results on a non-numeric, bit-by-bit, basis.

Division of an integer by another integer yields an integer. BASIC09 discards any remainder.

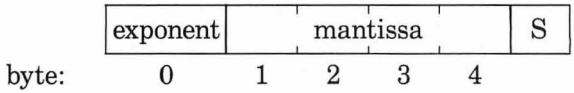
The Real Data Type

If you do not assign a data type to a variable, BASIC09 assumes the variable is real. However, programs are easier to understand if you define all variable types.

BASIC09 stores as real values any constants that have decimal points. If a constant does not have a decimal point, BASIC09 stores it as an integer.

BASIC09 requires five consecutive memory bytes to store real numbers. The first byte is the exponent, in binary two's complement. The next four bytes are the binary sign and magnitude of the mantissa. The mantissa is in the first 31 bits; the sign of the mantissa is in the last (least-significant) bit of the last byte. The following illustration shows the memory storage of a real number:

Internal Representation of Real Numbers



The exponent covers the range $2.938735877 \times 10^{-39}$ (2^{-128}) through $1.701411835 \times 10^{38}$ (2^{127}) as powers of 2. Operations that result in values out of the representation range cause an overflow or underflow error. You can handle such errors using the ON ERROR command.

The mantissa covers the range 0.5 through .9999999995 in steps of 2^{-31}. This means that real numbers can represent values .0000000005 apart. BASIC09 rounds operation values that fall between these points to the nearest point.

Because floating point arithmetic is inherently inexact, a sequence of operations can produce a cumulative error. Proper rounding, as implemented in BASIC09, reduces the effect of this problem, but cannot eliminate it. When using real quantities in comparisons, be sure your computations can produce the exact value you desire.

String Variables

A string is a variable-length sequence of ASCII values. The length can vary from 0, a *null* string, to the capacity of the memory available to BASIC09.

You can define a string variable either explicitly, using the DIM statement, or implicitly by appending the dollar sign ($) to the variable identifier (variable name). For example, title$ implicitly identifies a string variable.

Unless you specify otherwise, BASIC09 assigns a maximum string length of 32 characters. Using the DIM statement, you can specify a maximum length either less than or greater than 32. To conserve memory, use DIM to assign only the maximum length you need for any string variable.

The beginning of a string is always Character 1. The BASE statement, which sets numeric variable base numbers as either 0 or 1, does not affect string variables.

If an operation results in a string too long to fit in the assigned maximum storage space, the system truncates the string on the right. It does **not** produce an error.

String storage is fixed at the dimensioned length. The sequence of actual string byte values is terminated by the value of zero, or by the maximum length allotted to the string. Any unused storage after the zero byte allows the stored string to expand and contract within its assigned length.

The following example shows the internal storage of a variable dimensioned as `string [6]` and assigned the value "SAM". Note that Byte 4 contains the string terminator 00. The string does not use bytes following 00.

	S	A	M	00		
byte:	1	2	3	4	5	6

If you assign the value "ROBERT" to the variable, BASIC09 does not need to terminate the string with 00 because the string is full:

	R	O	B	E	R	T
byte:	1	2	3	4	5	6

The way BASIC09 handles string storage is important when you write programs. If you do not specify a length for strings you define, the system uses the default length 32. As you can see, this wastes computer memory if you store strings of only four or five characters.

The Boolean Type

A Boolean operation always returns either the character string "TRUE" or "FALSE". You cannot use the Boolean data type for numeric computation—only for comparison logic.

Do not confuse the Boolean operations AND, OR, XOR, and NOT (which operate on the Boolean values TRUE and FALSE) with the logical functions LAND, LOR, LXOR, and LNOT (which use integer values to produce numeric results on a bit-by-bit basis). An attempt to store a non-Boolean value in a Boolean variable, causes an error.

Automatic Type Conversion

When an operation mixes numeric data types (byte, integer, or real values), BASIC09 automatically and temporarily converts the values to the type necessary to retain accuracy. This conversion lets you use numeric quantities of mixed types in most calculations.

The system returns a type-mismatch error when an expression includes types that cannot legally mix. These errors are reported by the second compiler pass, which occurs automatically when you exit the edit mode.

Because type conversion takes additional execution time, you can speed calculations by using values of a single type.

Constants

Constants are values in a program that do not change. They can use any of the five data types. The following are examples of constants in a procedure:

```
HOME$="Fort Worth"
VALUE$="$25,000
VALUE=25
PAYMENT=99.99
ANSWER="TRUE"
MEMORY=$0CFF
PRINT "The End"
```

Numeric constants are either integers or real numbers. If a numeric constant includes a decimal point or uses the "E format" exponential form, it causes BASIC09 to store the number in the real format, even if it could store the number in integer or byte format.

You can use this feature to *force* a real format. For instance, to make the number 12 a real number, type it as 12.0. You might want to force real values in this way when all other values in an expression are real so that BASIC09 does not have to do a time-consuming type conversion at run time.

BASIC09 also stores as real numbers any numbers that do not have decimal points but that are too large to store as integers. Here are some examples of legal real constants:

1.0	9.8433218	-.01
-999.000099	100000000	5644.34532
1.95E+12	-99999.9E-33	

BASIC09 treats numbers that do not have a decimal point and are in the range -32768 through +32767 as integers. You must always precede hexadecimal numbers with a dollar sign.

Following are examples of legal integer constants:

12	-3000	55
$20	$FF	$09
0	-12	-32768

String Constants

A string constant consists of a sequence of characters enclosed in double quotation marks, such as:

```
"The End"
```

To place a string constant in a string type variable, use the equal symbol in this manner:

```
TITLE$ = "Masters Of Magic"
```

To include double quotation marks within a string, use two sets of double quotation marks, like this:

```
"An ""older man"" is wiser."
```

A string can contain characters that have ASCII values in the range 0 through 255.

Variables

In BASIC09, a variable is *local* to the procedure in which it is defined. A variable defined in one procedure has no meaning in another procedure unless you use the RUN and PARAM statements to pass the variable when you call the other procedure.

The local nature of variables lets you use the same variable name in more than one procedure and, unless you specify otherwise, have the variables operate independently of each other.

You can assign variables using either the LET statement with the assign symbol (=), or by using the assign symbol alone. For instance, both the following command lines are legal:

```
LET PAYMENT=44.50
PAYMENT=44.50
```

When you call a procedure, BASIC09 allocates storage for the procedure's variables. It is not possible to force a variable to occupy an absolute address in memory. When you exit a procedure, the system returns the storage allotted for variables, and you lose the stored values.

If you write a procedure to call itself (a *recursive* procedure), the call creates separate storage space for variables.

> **Note:** Unlike other BASICS, BASIC09 does not automatically initialize variables by setting them to zero. When you execute a procedure, all variables, arrays, and structures have random values. Your procedure must initialize the variables you specify to the values you require.

Passing Variables

When one procedure passes variable values to another procedure, BASIC09 refers to the passed variables as *parameters*. You can pass variables either by *reference* or by *value*.

BASIC09 does not protect variables passed by reference. Therefore, the called procedure can change the values and return the new values. BASIC09 **does** protect variables passed by value, so, the called program cannot change them.

To pass a parameter by reference, enclose the name of the variable in parentheses as part of the RUN statement in this manner:

```
RUN RANDOM(10)
```
passes the value 10 to a procedure called Random

The system evaluates the storage address of each passed variable, and sends the variable to the called procedure. The called procedure associates the storage addresses with the names in its local PARAM statement. It then uses the storage area as though it had created it locally. This means it can change the value of the parameter before returning it to the calling procedure.

To pass parameters by value, write the value to be passed as an expression. BASIC09 evaluates the expression at the time of the call. To use a variable in an expression without changing its value, use null constants, such as 0 for a number or "" for a string, in this manner:

```
RUN ADDCOLUMN(x+0)          passes the value of x by
                            value
RUN TRANSLATE(w$+"")        passes the contents of w$ by
                            value
```

To pass parameters by value, BASIC09 creates a temporary variable. It places the result of the expression in the temporary variable and sends the address to the called procedure. This means that the value given to the called procedure is a *copy* of the result of the expression, and the called procedure cannot change the original value.

The results of expressions containing numeric constants are either integer or real values; there are no byte constants. To send byte-type variables to a procedure, pass the values by reference. Therefore, if a RUN statement evaluates an integer as a parameter and sends it to a byte-type variable, the byte variable uses only the high-order byte of the two-byte integer.

Arrays

An *array* is a group of related data values stored consecutively in memory. The system knows the entire group by a variable name. Each data value is an *element*. You use a *subscript* to refer to any element of the array. For example, an array named Graf might contain five elements referred to as:

```
GRAF(1)   GRAF(2)   GRAF(3)   GRAF(4)   GRAF(5)
```

You can use each of these elements to store a different value, such as:

```
GRAF(1) = 25
GRAF(2) = 47
GRAF(3) = 39
GRAF(4) = 18
GRAF(5) = 50
```

Note: Normally, array elements start with 1 in BASIC09. However, you can use the BASE command to cause array elements to begin at 0.

The previous example illustrates a single-dimensioned array. The elements are arranged in one row and only one subscript is used for each element.

The following procedure lets you type values for a GRAF array, and displays the results in a simple graph.

```
PROCEDURE GRAF
□DIM GRAF(5):REAL
□SHELL "DISPLAY ØC"
□FOR T=1 TO 5
□PRINT "Value for Item #"; T; "□";
□INPUT GRAF(t)
□NEXT T
□PRINT
□PRINT
□PRINT "This is how your graph stacks up..."
□PRINT
□FOR T=1 TO 5
□PRINT "Item #"; T; "□";
□FOR U=1 TO GRAF(T)
□PRINT CHR$(79);
□NEXT U
□PRINT
□NEXT T
□PRINT
□END
```

This program uses a single dimension array—in effect, a list.

You can also create arrays with more than one dimension — more than one element for each row. You might use a two-dimensioned array in a program to store names and addresses. Instead of creating separate arrays for the name, address, and zip code, you could set up one array with two dimensions.

The following program, used to enter the names of a company's employees, shows how this might be done. See the second line for the DIM syntax. When you run the procedure, it asks you for a name, address, and zip code for each of 10 employees. After you type the information for all the entries, the procedure displays the information on the screen.

```
PROCEDURE Names
□DIM NAME(10,3):STRING
□SHELL "DISPLAY 0C"
□BASE 0
□FOR T=0 TO 9
□PRINT "Type Employee Name No."; T; ": ";
□INPUT NAME(T,0)
□PRINT "Type Employee Address No."; T; ": ";
□INPUT NAME(T,1)
□PRINT "Type Employee Zip Code No."; T; ": ";
□INPUT NAME(T,2)
□NEXT T
□SHELL "DISPLAY 0C"
□PRINT "And the names are..."
□PRINT
□FOR T=0 TO 9
□PRINT NAME(T,0); "□"; NAME(T,1); "□"; NAME(T,2)
□NEXT T
□END
```

The DIM statement reserves space in memory for a string array named Name, with two dimensions. As you enter data, the Name field is stored in `Name(0,0)`, `Name(1,0)`, `Name(2,0)`, and so on. The Address field is stored in `Name(0,1)`, `Name(1,1)`, `Name(2,1)`, and so on. The Zip field is stored in `Name(0,2)`, `Name(1,2)`, `Name(2,2)`, and so on. This continues until you fill the *grid*, 10 entries with three items each.

You can also create arrays with three dimensions. The following program adds one more dimension that keeps track of each employee's company. It dimensions `Name$` as `Name$(2,10,3)`. The first dimension contains either 0 or 1 to indicate to which company the employee belongs.

```
PROCEDURE names2
□DIM NAME$(2,10,3):STRING
□SHELL "DISPLAY 0C"
□BASE 0
□FOR X=0 TO 1
□PRINT
□PRINT
□FOR T=0 TO 9
□PRINT
□IF X=0 THEN
□PRINT "Type a Wiggleworth Company employee
name..."
□ELSE
□PRINT "Type a Putforth Company employee name..."
□ENDIF
□PRINT "Type Name No."; T; ": ";
□INPUT NAME$(X,T,0)
□PRINT "Type Address No."; T; ": ";
□INPUT NAME$(X,T,1)
□PRINT "Type Zip Code No."; T; ": ";
□INPUT NAME$(X,T,2)
□NEXT T
□NEXT X
□SHELL "DISPLAY 0C"
□PRINT "The Wiggleworth employees are..."
□PRINT
□X=0
□FOR T=0 TO 9
□PRINT NAME$(X,T,0); "□"; NAME$(X,T,1); "□";
NAME$(X,T,2)
□NEXT T
□PRINT
□PRINT "The Putforth Company employees are..."
□PRINT
□X=1
□FOR T=0 TO 9
□PRINT NAME$(X,T,0); "□"; NAME$(X,T,1); "□";
NAME$(X,T,2)
□NEXT T
□END
```

The easiest way to understand three dimensional arrays is to consider the first dimension as a *page*. In other words, if the first dimension in the string is 0, the employee is on the Wiggleworth Company's page. If the first dimension in the string is 1, the employee is on the Putforth Company's page.

Complex Data Types

In addition to the five standard data types, you can create your own data types. Using the TYPE command, you can define a new data type as a *vector* (a single-dimensioned array) of any previously defined type.

For example, in the previous program, the Name variable can contain only one type of data, the string type. However, using the TYPE command you can create a variable that accepts several data types.

Suppose you create an employee list procedure that uses the following variables, of the following size and types:

Name	Length	Contents	Type
Name	25	employee name	string
Street	20	street address	string
City	10	city of address	string
Zip	—	address zip code	integer
Sex	—	false = male, true = female	Boolean
Age	—	employee age	byte

You can combine all these variables into one complex data type. To do so, dimension the variables within a TYPE command line, like this:

```
TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
```

This creates a new BASIC09 type, called Employee. Employee requires its variables to have six fields of the name, size, and type shown in the previous chart.

Once you create the new data type, you can define variables to use it. For instance, the following program line defines Worker as type employee, with 10 elements in the array:

```
□DIM WORKER(10):EMPLOYEE
```

To put the employee data type to work, collect your data with INPUT commands. Then, store the data into the new Worker array. The following program demonstrates how you might do this:

```
PROCEDURE worker
□REM          Dimension variables for input
□DIM NM:STRING[25]
□DIM ST:STRING[20]
□DIM CTY:STRING[10]
□DIM ZP:REAL
□DIM SX:BOOLEAN
□DIM AG:BYTE
□REM          Create new type and array using new
type
□TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10 ]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
□DIM WORKER(10):EMPLOYEE
□REM
□FOR T=1 TO 10
□INPUT "Name:□",NM
□INPUT "Street:□",ST
□INPUT "City:□",CTY
□INPUT "Zip:□",ZP
□INPUT "Sex:□",SX
□INPUT "Age:□",AG
□REM        Store input in the Worker array using
field names
□WORKER(T).NAME=NM
□WORKER(T).STREET=ST
□WORKER(T).CITY=CTY
□WORKER(T).ZIP=ZP
□WORKER(T).SEX=SX
□WORKER(T).AGE=AG
□PRINT
□PRINT "* * * * * * * * * * * * * * * * * * * *"
□PRINT
□NEXT T
□SHELL "DISPLAY (0C)"
□PRINT "The names in your files now are..."
□PRINT
□FOR T=1 TO 10
□PRINT WORKER(T).NAME
□PRINT WORKER(T).STREET
□PRINT WORKER(T).CITY
```

```
□PRINT WORKER(T).ZIP
□IF WORKER(T).SEX=TRUE
□THEN PRINT "Female"
□ELSE
□PRINT "Male"
□ENDIF
□PRINT WORKER(T).AGE
□PRINT
□PRINT "* * * * * * * * * * * * * * * * * * * * *"
□PRINT
□NEXT T
```

Note that the Sex field is defined as Boolean. This means that you can respond only in two ways, TRUE or FALSE. The method of input requires only one byte of storage. To use this data you need to handle it so TRUE and FALSE indicate female and male.

Complex data types can contain more than one field. Each field can be of any data type. You reference the fields of a complex data type by typing the variable name, its array index, a period (.), and the field name. The following lines, from the Worker procedure, show how BASIC09 stores the data from the input lines into the Worker variable:

```
WORKER(T).NAME=NM
WORKER(T).STREET=ST
WORKER(T).CITY=CTY
WORKER(T).ZIP=ZP
WORKER(T).SEX=SX
WORKER(T).AGE=AG
```

These lines store the values in the variables NM, ST, CTY, ZP, SX, and AG into the NAME, STREET, CITY, ZIP, SEX, and AGE fields of the Worker variable. This operation is within a FOR/NEXT loop that uses T as a counter. In the procedure, T can refer to a value in the range 1 to 10.

The procedure uses the same type of operation to extract the data from the complex data type variable:

```
PRINT WORKER(T).NAME
PRINT WORKER(T).STREET
PRINT WORKER(T).CITY
PRINT WORKER(T).ZIP
IF WORKER(T).SEX=TRUE THEN PRINT "Female"
ELSE PRINT "Male"
ENDIF
PRINT WORKER(T).AGE
```

Using the same methods, you can create complex data types that combine other complex data types and standard data types.

The elements of a complex structure can be copied to another similar structure. Using a single assignment operator, you can write an entire structure to, or read an entire structure from, mass storage as a single entity. For example:

```
PUT #2, WORKER(T)
```

Because the system defines the elements of complex-type storage during compilation, it need not do so during *runtime*. This means that BASIC09 can reference complex structure faster than it can reference arrays.

Chapter 7

Expressions, Operators, and Functions

Manipulating Data

BASIC09 uses *expressions* to manipulate data. (Expressions are pieces of data connected by operators.)

An *operator* is a symbol or a word that signifies some action to be performed on the specified data. Each data item is a *value.*

Expressions

When an expression is evaluated, the result is a value of some data type (real, integer, string, byte, or Boolean).

An expression might look like this:

First Value	First Operator	Second Value	Second Operator	Result
6	+	5	=	11

or like this:

First Value	First Operator	Second Value	Second Operator	Result
"Seaside"	+	"Villa"	=	Seaside Villa

When BASIC09 evaluates an expression, it copies each value onto an *expression stack.* Functions and operators take their input values from this stack and return their results to it. Many expressions result in assignments, as do the examples shown. The BASIC09 makes the resulting assignment only after it computes the entire expression. This lets you use the variable that is being modified as one of the values in the expression, such as in this example:

```
X = X + 1
```

The result of an expression is always one of the five BASIC09 data types. However, you can often mix data types within an expression and, in some cases, the result of an expression is of a different data type than any of the values in the expression. Such is the case if you use the *less-than* symbol (<), in this manner:

```
24 < 100
```

The less-than operator compares two integer values. The result of the comparison is Boolean; in this case, the value is TRUE.

Type Conversion

Because BASIC09 performs automatic type conversion of values, you can mix any of the three numeric data types in an expression. When you mix numeric data types, the result is always of the same type as the value having the largest representation, in this order: real < integer < byte.

You can use any numeric type in an expression that produces a real number. If you want an expression to produce a byte or integer type value, the result must be small enough to fit the desired type.

Operators

BASIC09 has operators to deal with all types of data. Each operator, except NOT and negation (unary -), takes two values or *operands*, and performs an operation to produce a result. NOT can accept only one value. The following table lists the operators available and the types of data they accept and produce.

Because the same operators function on the three types of numeric data (byte, integer, and real), these types are referred to by the operand type "numeric."

BASIC09 Expression Operators

Operator	Function	Operand Type	Result Type
–	Negation	numeric	numeric
^ or **	Exponentiation	numeric	numeric
*	Multiplication	numeric	numeric
/	Division	numeric	numeric
+	Addition	numeric	numeric
–	Subtraction	numeric	numeric
NOT	Logical Negation	Boolean	Boolean
AND	Logical AND	Boolean	Boolean
OR	Logical OR	Boolean	Boolean
XOR	Logical Exclusive OR	Boolean	Boolean
+	Concatenation	string	string
=	Equal to	all types	Boolean
<> or ><	Not equal to	all types	Boolean
<	Less than	numeric, string†	Boolean
<= or =<	Less than or equal	numeric, string†	Boolean
>	Greater than	numeric, string†	Boolean
>= or =>	Greater than or equal	numeric, string†	Boolean

† When comparing strings, BASIC09 uses the ASCII values of characters as the basis for comparison. Therefore, 0 < 1, 9 < A, A < B, A < b, b < z, and so on.

Arithmetic Operators

Arithmetic operators perform operations on numeric data. Therefore, both operands in the expression must be numeric. The following table lists the arithmetic operators.

Negation The single dash negates a number's sign: -10 is *negative* 10.

Exponentiation Use a caret (^) or two asterisks (**) to raise a number to a power: 2^3 is 8 (2 x 2 x 2). Similarly, 2**3 is 8.

Multiplication A single asterisk causes multiplication: 2 * 3 is 6.

Division A slash causes division: 6 / 2 is 3.

Addition The plus sign causes addition: 3 + 3 is 6.

Subtraction A dash causes subtraction: 6 - 3 is 2.

Hierarchy of Operators

BASIC09 uses the standard hierarchy of operations when calculating expressions with multiple operators. This means that BASIC09 has an order in which it performs calculations involving more than one operator.

The following BASIC09 operators are listed in order of precedence:

```
NOT     - (negate)
^       **
*       /
+       -
>       <   <>   =   >=   <=
AND
OR      XOR
```

Also, BASIC09:

- Performs operations enclosed in parentheses **before** operations not in parentheses.
- Performs the leftmost operations first when two or more operations are of equal precedence.

You can use parentheses to override this standard precedence. For example:

```
2 + 1 * 3 = 5
```

but

```
( 2 + 1 ) * 3 = 9
```

The following examples show BASIC09 expressions on the left, and the way BASIC09 evaluates them on the right. You can enter the expressions in either form, but the decompiler generates the simpler form, shown on the left.

BASIC09 Representation	Equivalent Form
a=b+c**2/d	a=b+((c**2)/d)
a=b>c AND d>e OR c=e	a=((b>c) AND (d>e)) OR (c=e)
a=(b+c+d)/e	a=((b+c)+d)/e
a=b**c**d/e	a=(b**(c**d))/e
a= -(b)**2	a=(-b)**2
a=b=c	a=(b=c)

Relational Operators

Relational operators make logical comparisons of any type of data and return a result of either TRUE or FALSE. An explanation of the relational operators follows. All relational operators have equal precedence.

= Equal. Returns TRUE if both operands are equal, or FALSE if they are not equal.

< Less than: Returns TRUE if the first operand is less than the second, or FALSE if is not.

> Greater than: Returns TRUE if the first operand is greater than the second, or FALSE if it is not.

<> or >< Unequal: Returns TRUE if the operands are not equal or FALSE if they are.

<= or =< Less than or equal to: Returns TRUE if the first operand is less than or equal to the second operand. Otherwise, the operation returns FALSE.

>= or => Greater than or equal to: Returns TRUE if the first operand is greater than or equal to the second. Otherwise, the operation returns FALSE.

You normally use relational operators in IF/THEN statements. For example, if your procedure has two numeric variables, Payments and Income, you might include command lines like this:

```
IF PAYMENTS > INCOME THEN
  PRINT "You're Broke!"
ENDIF
```

When you combine arithmetic and relational operators in the same expression, BASIC09 evaluates the arithmetic operations first. For example:

```
IF X*Y/2 <= 14 THEN
  PRINT "Average Score is "; X*Y/2
ENDIF
```

BASIC09 performs the arithmetic operation x*y/2, then compares the result with the value 14.

When you use relational operators with strings, BASIC09 compares the strings character by character. When it finds two characters that do not match, it checks to see which character has the lower ASCII code value. The string containing the character with the lower value comes first.

Consider this example:

```
PRINT "hunt" > "hung"
```

BASIC09 compares each character in each string. Because the first three characters are the same, the result of the operation is based on the comparison of t and g. Because t (ASCII value = 116) is "greater than" g (ASCII value = 103), the command prints TRUE.

String Operators

The *string operator* is the plus sign (+). This symbol appends one string to another. All operands must be strings, and the resulting value is one string. Examine, for example, the following line, which appends three strings:

```
PRINT "My friends are " + "Jack and " + "Jill."
```

It prints: `My friends are Jack and Jill.`

Logical Operators

The logical, or Boolean, operators make logical comparisons of Boolean values. The following table describes the results yielded by each logical operator given the specified TRUE/FALSE values:

Operator	Meaning of Operation	First Operand	Second Operand	Result
NOT	The result is the opposite of the operand.	TRUE FALSE		FALSE TRUE
AND	When both values are TRUE, the result is TRUE. Otherwise, the result is FALSE.	TRUE TRUE FALSE FALSE	TRUE FALSE TRUE FALSE	TRUE FALSE FALSE FALSE
OR	When both values are FALSE, the result is FALSE. Otherwise, the result is TRUE.	TRUE TRUE FALSE FALSE	TRUE FALSE TRUE FALSE	TRUE TRUE TRUE FALSE
XOR	When only one of the values is TRUE, the result is TRUE. Otherwise the result is FALSE.	TRUE TRUE FALSE FALSE	TRUE FALSE TRUE FALSE	FALSE TRUE TRUE FALSE

Use logical operators in IF/THEN statements such as:

```
IF PAYMENTS < INCOME AND INCOME+SAVINGS >
PAYMENTS THEN
  PRINT "You'll have to use your savings to get
out of this mess."
ENDIF
```

Functions

Functions are operation sequences the system performs on data. In a statement, BASIC09 performs functions first. Chapter 11, "Command Reference," describes the following functions.

Functions returning results of type real

SIN	Calculates the trigonometric sine of a number.
COS	Calculates the trigonometric cosine of a number.
TAN	Calculates the trigonometric tangent of a number.
ASN	Calculates the trigonometric arcsine of a number.
ACS	Calculates the trigonometric arccosine of a number.
ATN	Calculates the trigonometric arctangent of a number.
LOG	Calculates the natural logarithm (base e) of a number.
LOG10	Calculates the logarithm (base 10) of a number.
EXP	Calculates e (2.71828183) raised to the specified positive power.
FLOAT	Converts byte or integer type numbers to real numbers.
INT	Calculates the largest whole number less than or equal to the specified number.
PI	Represents the constant 3.14159265.
SQR	Calculates the square root of a positive number.
SQRT	Calculates the square root of a positive number. Its function is identical to SQR.
RND	Returns a random number.

Functions returning results of any numeric type

The resulting type depends on the input type.

ABS	Calculates the absolute value of a number.
SGN	Returns a value to indicate the sign of the specified number (-1 if the number is less than 0, 0 if the number is 0, or 1 if the number is greater than 0).
SQ	Calculates the square of a number.
VAL	Converts a string to a numeric value.

Functions returning results of type integer or type byte

FIX	Rounds a real number and converts it to an integer.
MOD	Calculates the modulus (remainder) of two numbers.
ADDR	Returns the absolute memory address of a variable, an array, or a structure.
SIZE	Returns (in bytes) the storage size of a variable, an array, or a structure.
ERR	Returns the error code of the most recent error.
PEEK	Returns the byte value at a specified memory address.
POS	Returns the current character position of the print buffer.
ASC	Returns the numeric value (ASCII code) of a string character.
LEN	Returns the length of a string.
SUBSTR	Returns the starting position of the specified substring within a string, or returns 0 if it cannot find the substring.

Functions performing bit-by-bit logical operations on integer or byte data and returning integer results. Do not confuse these functions with Boolean type operators.

LAND	Calculates the logical AND of two values.
LOR	Calculates the logical OR of two values.
LXOR	Calculates the logical EXCLUSIVE OR of two values.
LNOT	Calculates the logical NOT of a value.

Functions returning a result of type string

CHR$	Returns the character having a specified ASCII value.
DATE$	Returns the system's current date and time.
LEFT$	Returns the specified number of characters beginning at the leftmost character of the specified string.
RIGHT$	Returns the specified number of characters beginning at the rightmost character of the specified string and counting backward.
MID$	Returns the specified number of characters starting at the specified position in a string.
STR$	Converts numeric type data to string type.
TRIM$	Removes trailing spaces from the specified string.

Functions returning Boolean values

TRUE	Always returns TRUE.
FALSE	Always returns FALSE.
EOF	Tests for the end of a disk file. Returns TRUE when the end of the file occurs.

Chapter 8

Disk Files

When you tell OS-9 or BASIC09 to store (save) data on a disk, it stores the data in a *logical* block called a *file*. The term logical means that, although the system might store portions of a file's data in several different disk locations, it keeps track of every location and treats the scattered data as though it occupied a single block. It does this automatically and you never need to worry about how the data is stored. File data can be binary data, textual data (ASCII characters), or any other useful information.

Because OS-9 handles all hardware input/output devices (disk drives, printers, terminals, and so on) in the same manner, you can send data to any of these devices in the same way. This means you can send the same information to several devices by changing the path the data follows. For example, you can test a procedure that communicates with a terminal by transferring data to and from a disk drive.

BASIC09 normally works with two types of files—sequential files and random access files. The following chart shows file-access options, their purposes, and the keywords with which to use them:

Types of Access for Files

Access Type	Function	Use with
DIR	Opens a directory file for reading. Use only with READ.	OPEN
EXEC	Specifies that the file to open or create is in the execution directory, rather than the data directory.	OPEN CREATE
READ	Lets you read data from the specified file or device.	OPEN CREATE
WRITE	Lets you write data to the specified file or device.	OPEN CREATE
UPDATE	Lets you read data from and write data to the specified file or device.	OPEN CREATE

Sequential Files

Sequential files send or receive (WRITE or READ) textual data in order, the second item following the first, and so on. You can access sequential data only in the same order as you originally stored it. To read from or write to a particular section of a file, you must first read through all the preceding data in the file, starting from the beginning.

BASIC09 stores sequential file data as ASCII characters. Each block of data is separated by a *delimiter* consisting of a carriage-return character (ASCII Character 13). Because BASIC09 uses this delimiter to determine the end of a *record*, sequential files can contain records of varying length.

Use the WRITE and READ commands to store and retrieve data in sequential files. A WRITE command causes BASIC09 to transfer specified data to a specified file, ending the data with a carriage return. A READ command causes BASIC09 to load from the specified file the next block of data, stopping when it reaches a carriage return.

Sequential File Creation, Storage, and Retrieval

BASIC09 uses the CREATE command to establish both sequential and random access files. A CREATE statement contains:

- The keyword CREATE.
- A path number variable in which BASIC09 stores the number of the path it opens to the new file.
- A comma, followed by the name of the file to create.
- An optional colon, followed by the access mode. If you do not specify an access mode, BASIC09 automatically opens the created file in the UPDATE mode.

The following procedure shows how to create a file and write data into it:

```
PROCEDURE makefile
□DIM PATH:BYTE                    (* establishes a variable
□REM                                 for the path number to the file
□CREATE #PATH,"test":WRITE        (* creates the file TEST
□WRITE #PATH,"This is a test"     (* writes data to the file
□WRITE #PATH,"of sequential files."(* writes another line of data
□CLOSE #PATH                      (* closes the path to the file
□SHELL "LIST TEST"                (* displays the file contents
□END
```

The first line of the procedure dimensions a variable (Path) to hold the number of the path that CREATE opens. This variable should be of byte or integer type.

When you establish a new file with CREATE, you automatically open a path to the file. You do not need to use the OPEN command.

The preceding procedure writes two lines into a file named Test. It then closes the path and uses the OS-9 LIST command to display the contents of the newly created file. You see that the data is successfully stored on disk.

The next procedure shows how to reopen an existing file for sequential access, read the contents of the file, and append data to the end of the file.

The only way to move the *file pointer* to the end of a sequential file is to read all the data already in the file. Once the pointer is at the end of the file, you can add data.

```
PROCEDURE append
□DIM PATH:BYTE                (* dimension variable to hold the number of the
□REM                             path to the opened file.
□OPEN #PATH,"test":UPDATE     (* open file for reading and writing.
□READ #PATH,line$             (* read the first element of the file.
□READ #PATH,line$             (* read the next (the last) element.
□WRITE #PATH,"This is a test" (* write one new line to the file.
□WRITE #PATH,"of appending to a sequential file." (* write another.
□CLOSE #PATH                  (* close the path.
□SHELL "LIST TEST"            (* display the file with the new lines.
□END
```

Because the Test file already exists, this procedure uses OPEN to establish a path to the file. It uses the UPDATE mode of file access because it needs to both read from and write to the file.

The two READ statements read the file's contents and, as a result, move the file pointer to the end of the file. The WRITE statements then append two new lines. After closing the path, the procedure calls on the OS-9 LIST command to display the contents of the file, with its appended lines.

Changing Data in a Sequential File

You can also change data anywhere in a sequential file. However, if your changes are longer than the original data, the operation destroys part of the file. To change data in a sequential file, read the data preceding what you want to change, and write the new data to the file in this manner:

```
PROCEDURE replace
□DIM PATH:BYTE
□OPEN #PATH,"test":UPDATE
□READ #PATH,line$
□READ #PATH,line$
□WRITE #PATH,"Let's put new"   (* write over existing 3rd and
□WRITE #PATH,"words into the old sequential file.'' (* 4th lines.
□CLOSE #PATH
□SHELL "LIST TEST"
□END
```

Notice that the total amount of data in the two new lines is exactly the same as in the two old lines. You can replace an existing line with fewer characters by *padding* the new data with spaces. However, if you try to replace existing lines with longer lines, the new lines write over and destroy other data in the file.

INPUT and Sequential Files

Although you can also use the INPUT command with sequential files, doing so might put unwanted data into them. When a procedure encounters INPUT, it suspends execution and sends a question mark (?) to the screen. This feature makes INPUT both an input and output statement. Therefore, if you open a file using the UPDATE mode, INPUT writes its prompts to the file, destroying data. If you specify text to be displayed with the INPUT command, INPUT writes this text to the file also.

Random Access Files

Random access files store data in fixed- or equal-length blocks. Because each record in a specific file is the same size, you can easily calculate the position of a record.

For instance, suppose you have a file with a record length of 50-bytes (or characters). To access Record 10, multiply the record number (10) by the record length (50) and move the file pointer to the calculated position (500).

A random access file sends and receives data (using PUT and GET) in a binary form, exactly as BASIC09 stores it internally. This feature minimizes the time involved in converting the data to and from ASCII representation, as well as reducing the file space required to store numeric data. You position the random access file pointer using SEEK. Compared to sequential file access, random file access using GET and PUT is very fast.

Using random access commands, you can store and retrieve individual bytes, strings of bytes, individual elements of arrays or total arrays with one PUT or GET command. When you GET a structure, you recover the number of bytes associated with that type of structure.

This means when you GET one element of byte type data, you read one byte. When you GET one element of real type data, you read five bytes. If you GET an array, you read all the elements of the array. This potential for reading entire arrays at once can greatly speed disk access.

As well as moving the file pointer to the beginning of individual records, you can also move it to any position within a record and begin reading or writing one or more bytes from that point.

Creating Random Access Files

You create and open random access files in the same way you create and open sequential files. The only differences are in the commands you use to store and retrieve the data and in the manner you keep track of where elements, or records, of a file begin and end.

Before you can write data to a random access file, you must either CREATE it or open it in the WRITE or UPDATE mode. Once you have a path open to an existing file, use PUT to write data into the file. If you open the file in the READ or UPDATE mode, you can then use the GET command to retrieve data from the file.

The PUT command can use only one parameter, the name of the data element to store. The parameter can be a string, a variable, an array, or a complex data structure.

Before storing data, you must devise a method to store it in blocks of equal size. Knowing the unit size lets you later retrieve the data in its original form. The following procedure shows one way to do this:

```
PROCEDURE putget
□REM This procedure creates a file named Test1, reads 10 data lines,
□REM PUTs them into the file, then closes the file. Next it
□REM opens the file in the READ mode, GETS stored lines and lists
□REM them on the display screen.

□DIM LENGTH:BYTE
□DIM NULL:STRING[25]
□DIM LINE:STRING[25]
□DIM PATH:BYTE
□LENGTH=25
□NULL=""
□BASE 0
□ON ERROR GOTO 10
□DELETE "test1"          (* if the file exists, delete it.
10□ON ERROR

□CREATE #PATH,"test1":WRITE (* create a file named test1.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T       (* find beginning of each file.
□READ LINE$                (* read a line of data.
□PUT #PATH,LINE$           (* store the line in the file.
□NEXT T
```

```
□CLOSE #PATH                (* close the file.

□OPEN #PATH,"test1":READ    (* open the file for reading.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T        (* find the beginning of each file.
□GET #PATH,LINE             (* get a line from the file.
□PRINT LINE                 (* display the line.
□NEXT T

□CLOSE #PATH                (* close the file.
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #10"
```

This procedure creates a file named Test1. The variable named Length stores the length of each line in the file (25 characters). The string variable Null, is a string of 25 space characters. The variable Line contains the data to store in each element (record) in the file. The variable Path stores the path number of the file.

Next, the procedure contains an ON ERROR routine that deletes the file Test1, if it already exists. Without this routine, the procedure produces an error if you execute it more than once.

Next, the routine uses CREATE to open the file Test1. The line `SEEK #PATH, LENGTH*T` sets the file pointer to the proper location to store the next line. Because Length is established as 25, the file lines are stored at 0, 25, 50, 75, and so on.

After the routine initializes storage space, it begins to store data by reading the procedure data lines one at a time, seeking the proper file location, and putting the data into the file. After storing all 10 lines, it closes the file.

The last part of the routine opens the new file, uses the same SEEK routine to position the file pointer, and reads the lines back, one at a time, to confirm that the store routine is successful.

The next short routine shows how you can use a procedure to read any line you select in the file, without reading any preceding lines:

```
PROCEDURE randomread
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM SEEKLINE:BYTE
□DIM PATH:BYTE
□LENGTH=25

□OPEN #PATH,"test1":READ            (* open the file for reading.
□LOOP
□INPUT "Line number to display...",SEEKLINE (* type a line to get.
□EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN (* test if record is valid.
□ENDEXIT                            (* exit loop if not.
□SEEK #PATH,(SEEKLINE-1)*LENGTH     (* find the requested record.
□GET #PATH,LINE                     (* read the record.
□PRINT LINE                         (* display the record.
□PRINT
□ENDLOOP
□PRINT "That's all ..... "          (* end session.
□CLOSE #PATH                        (* close path.
□END
```

The procedure asks for the record number of the line to display. When you type the number (1-10) and press [ENTER], SEEK moves the file pointer to the beginning of the record you want, GET reads it into the variable Line, and PRINT displays it. The calculation `(SEEKLINE-1)*LENGTH` determines the beginning of the line you want. If you type a number outside the range of lines contained in the file (1-10), the procedure drops down to Line 100 and ends.

By changing this procedure slightly, you can replace any line in the procedure with another line. The altered procedure below demonstrates this:

```
PROCEDURE random_replace
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM SEEKLINE:BYTE
□DIM PATH:BYTE
□LENGTH=25

□OPEN #PATH,"test1":UPDATE(* open the file.
□LOOP
□INPUT "Line number to display...",SEEKLINE (* type record to find.
□EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN (* test if valid number.
□ENDEXIT                          (* exit loop if not
□SEEK #PATH,(SEEKLINE-1)*LENGTH (* find the requested record.
□GET #PATH,LINE                   (* get the data.
□PRINT LINE                       (* print the record.
□PRINT
□INPUT "Type new line... ",LINE (* type a new line.
□SEEK #PATH,(SEEKLINE-1)*LENGTH (* find beginning of the record.
□PUT #PATH,LINE                   (* store the new line.

□ENDLOOP                          (* do it all again.

□PRINT "That's all ..... "     (* terminate procedure.
□CLOSE #PATH                      (* close path.
□END
```

This time, the file is opened in the UPDATE mode to allow both reading and writing. You type the line you want to display. A prompt then asks you to type a new line. The procedure exchanges the new line for the original line, and stores it back in the file.

Using Arrays With Random Access Files

BASIC09's random access filing system is even more impressive when used with data structures, such as arrays. Instead of using a loop to store the 10 lines of the Random_replace procedure, you could store them all at once, into one record, using an array. The following procedure illustrates this:

```
PROCEDURE arraywrite
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM RECORD(10):STRING[25]
□DIM PATH:BYTE
□LENGTH=25

□ON ERROR GOTO 10
□DELETE "test1"                (* delete Test1 if it exists.
10□ON ERROR

□CREATE #PATH,"test1":WRITE        (* create Test1.
□BASE 0

□FOR T=0 TO 9
□READ RECORD(T)                    (* Read data lines into RECORD array.
□NEXT T

□SEEK #PATH,0                      (* set pointer to beginning of file.
□PUT #PATH,RECORD                  (* store the entire array into file.
□CLOSE #PATH                       (* close path to file.

□OPEN #PATH,"test1":READ           (* open the file to read.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T               (* find each element.
□GET #PATH,LINE                    (* read an element.
□PRINT LINE                        (* print the element.
□NEXT T
□CLOSE #PATH
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #10"
```

This procedure reads the 10 lines into an array named Records. Then it places the entire array in the Test1 file, using one PUT statement. To show that the structure of the file is still the same, the original FOR/NEXT loop reads the lines, one at a time, and displays them.

Notice that, because you need to write only one element, you can set the file pointer to 0 (`SEEK #PATH,0`). You can *rewind* a file pointer (set it to 0) at any time in this manner.

You could save additional programming space by also reading the 10 lines back into memory as an array. The following procedure uses a new array, Readlines, to call the file back into memory, and displays the lines.

```
PROCEDURE arrayread
□BASE 0
□DIM READLINES(10):STRING[25]
□DIM PATH:BYTE

□OPEN #PATH,"test1":READ        (* open file.
□GET #PATH,READLINES            (* read file into array.
□CLOSE #PATH

□FOR T=0 TO 9
□PRINT READLINES(T)             (* print each element of the array.
□NEXT T
□END
```

Using Complex Data Structures

In the previous section, you stored and retrieved elements of an array that were all the same size, 25 characters. Often you need to store elements of varying sizes, such as when you create a data base program with several fields in one record.

The following examples create a simple inventory system that requires a random access file having 100 records. Each record includes the name of the item (a 25-byte string), the item's list price and cost (both real numbers), and the quantity on hand (an integer).

First, you use the TYPE command to define a new data type that describes such a record. For example:

```
TYPE INV_ITEM=NAME:STRING[25];LIST,COST:REAL;
QTY:INTEGER
```

Although this statement describes a new record type called Inv_item, it does not assign variable storage for the record. The next step is to create two data structures: an array of 100 records of type Inv_item named Inv_array and a working record named
Work_rec. The following lines do this:

```
DIM INV_ARRAY(100):INV_ITEM
DIM WORK_REC:INV_ITEM
```

To determine the number of bytes assigned for each type, you can use BASIC09's SIZE command. SIZE returns the number of bytes assigned to any variable, array, or complex data structure. For example, the command line `SIZE(WORKI_REC)` returns the number 37. The command `SIZE(INV_ARRAY)` returns the number `3700`.

You can use SIZE with SEEK to position a file pointer to a specific record's address.

The following procedure creates a file called Inventory and immediately initializes it with zeroes and null strings. Five INPUT lines then ask you for a record number and the data to store in each field of the record. You can fill any record you choose, from 1 through 100.

When one record is complete, the procedure uses PUT to store the record. Then, it asks you for a new record number. If you wish to quit, enter a number either larger than 100 or smaller than 1.

```
PROCEDURE inventory
□REM Create a data type consisting of a 25-character name field,
□REM a real list price field, a real cost field, and an integer
□REM quantity field.
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER

□DIM INV_ARRAY(100):INV_ITEM (* dimension an array using new type.
```

```
□DIM WORK__REC:INV__ITEM
□REM          (* dimension a working variable of the new type.
□DIM PATH:BYTE

□ON ERROR GOTO 10
□DELETE "inventory"
10□ON ERROR

□CREATE #PATH,"inventory"    (* create a file named Inventory.
□WORK__REC.NAME=" "          (* set all data elements to null or 0.
□WORK__REC.LIST=0
□WORK__REC.COST=0
□WORK__REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK__REC
□NEXT N

□LOOP
□INPUT "Record number? ",RNUM (* enter number of record to write.
□IF RNUM<1 OR RNUM>100 THEN   (* check if number is valid.
□PRINT
□PRINT "End of Session"       (* if not, end session.
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK__REC.NAME  (* type data for record.
□INPUT "List price? ",WORK__REC.LIST
□INPUT "Cost price? ",WORK__REC.COST
□INPUT "Quantity? ",WORK__REC.QTY
□SEEK #PATH,(RNUM-1)*SIZE(WORK__REC) (* find record.
□PUT #PATH,WORK__REC                 (* write record to file.
□ENDLOOP
```

Notice that the INPUT statements reference each field separately, but the PUT statement references the record as a whole.

The next procedure lets you read any record in your Inventory file, and displays that record. If you ask for a record you have not yet filled with meaningful data, the display consists of a null string and zeroes.

```
PROCEDURE readinv
□TYPE INV__ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
□DIM WORK__REC:INV__ITEM
```

```
□DIM PATH:BYTE
□OPEN #PATH,"INVENTORY":READ
□LOOP
□INPUT "Record number to display? ",RNUM
□IF RNUM<1 OR RNUM>100 THEN
□PRINT "End of Session"
□PRINT
□CLOSE #PATH
□END
□ENDIF
□SEEK #PATH,(RNUM-1)*SIZE(WORK_REC)
□GET #PATH,WORK_REC
□PRINT "#","Item","List Price","Cost Price","Quantity"
□PRINT "-------------------------------------------------------------------
---"
□PRINT RNUM,WORK_REC.NAME,WORK_REC.LIST,WORK_REC.COST,WORK_REC.QTY
□PRINT
□ENDLOOP
□END
```

This procedure accesses the file one record at a time. It is not necessary to do so. You can read the entire file into memory at once by dimensioning an inventory array and getting the whole file into it:

```
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
□DIM INV_ARRAY(100):INV_ITEM
□SEEK #PATH,0    (*rewind the file*)
□GET #PATH,INV_ARRAY
```

The examples in this section are simple, yet they illustrate the combined power of BASIC09 complex data structures and the random access file statements. They show that a single GET or PUT statement can move any amount of data, organized in any way you want. Other advantages are of using complex data structures are:

- The procedures are self-documenting. You can see easily what a procedure does because its structures can have descriptive names.
- Execution is extremely fast.
- Procedures are simple and usually require fewer statements to perform I/O functions than other BASICs.

- The procedures are versatile. By creating appropriate data structures, you can read or write almost any kind of data from any file, including files created by other programs or languages.

Displaying Text and Graphics

BASIC09 has three levels of graphics capabilities. The first and third levels can include both graphics designs and text. The second level can display only graphics designs.

ASCII Codes

For low-resolution text screens and high-resolution text and graphic screens, BASIC09 uses ASCII (American Standard Code for Information Interchange) codes to represent the common alphanumeric characters. ASCII is the same code that most small computers use.

A table of the standard codes follows:

Table 9.1
BASIC09 ASCII Codes 0-127
Low- and High-Resolution Screens

Character	Decimal Code	Hexadecimal Code
[BREAK]	03	03
[←]	8	O8
[→]	9	O9
[↓]	10	OA
[CLEAR]	12	0C
[ENTER]	13	0D
Space	32	20
!	33	21
"	34	22
#	35	23
$	36	24
%	37	25
&	38	26
'	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30

Character	Decimal Code	Hexadecimal Code
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[(SHIFT ↓)	91	5B

Character	Decimal Code	Hexadecimal Code
\ (SHIFT CLEAR)	92	5C
] (SHIFT →)	93	5D
↑	94	5E
← (SHIFT ↑)	95	5F
^	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
|	124	7C
}	125	7D
~	126	7E
—	127	7F

You can generate the characters in this chart by pressing the appropriate key, or you can generate them from BASIC09 using the CHR$ function.

Low-Resolution Graphic Characters

In addition to alphanumeric characters, low-resolution graphics also offers graphic characters. Generate these characters by pressing [ALT] at the same time you press a keyboard character. The graphics character codes are in the range 128-255.

Pressing [ALT] while pressing another key, causes OS-9 to add 128 to the ASCII value of the second key. (For the technically minded, OS-9 sets the high bit of the character code.) Therefore, if you press [ALT] [A], you produce graphics character 193. You can also generate graphics characters from BASIC09 using the CHR$ function, and you can PRINT them in the same manner as other characters.

Low-level graphics characters follow a pattern that repeats every 16 characters. Table 9.2 shows the first set of graphic characters, 128-143. Subsequent characters produce the same series of configurations but display in different colors, as shown in Table 9.3.

Table 9.2
Low-Resolution Graphic Character Set

Character	Code	Character	Code	Character	Code	Character	Code
	128		132		136		140
	129		133		137		141
	130		134		138		142
	131		135		139		143

Table 9.3
Low-Resolution Graphics Color Set

ASCII Code	Graphics Block Color
128 - 143	Black and Green
144 - 159	Black and Yellow
160 - 175	Black and Blue
176 - 191	Black and Red
192 - 207	Black and Buff
208 - 223	Black and Light Blue
224 - 239	Black and Cyan
240 - 254	Black and Orange
255	Green

Within each color set, you can easily calculate the number for a particular character. For instance, suppose you want to print a character that has orange upper left and lower right corners. Picture the character divided into four sections, numbered as follows:

8	4
2	1

To calculate a character that has orange at Sections 8 and 1, add the section values to the first value in the orange group, 240, like this:

240 + 8 + 1 = 249

Character 249 is what you want.

The following diagram shows how you might block out a large letter O on the screen. The shaded portions of the characters are colored. The unshaded portions are black. In this case we want the colored portions to be green (the same color as the screen). You can do this using the color set 128 - 143.

8	4	8	4	8	4
2	1	2	1	2	1
8	4	8	4	8	4
2	1	2	1	2	1
8	4	8	4	8	4
2	1	2	1	2	1
8	4	8	4	8	4
2	1	2	1	2	1
8	4	8	4	8	4
2	1	2	1	2	1

Because Section 1 in the upper left character is to be colored, add 1 to the initial character value of 128. The first character value is 129. Moving right, Sections 2 and 1 are colored in the second character. Add 3 to 128 to get a second character value of 131. Calculate all 15 characters in this manner.

You could create a letter O in a BASIC09 procedure by *printing* each of the five rows of three characters. You could use DATA lines to store the ASCII codes for each character, then use loops to read and display the characters they represent.

Although low-level graphics is very rough, it can be useful, and it lets you mix graphics with text.

The following procedure not only creates the letter O, it adds the letter S and the number 9 to display the name of your operating system.

```
PROCEDURE os9prog
□DIM DAT:INTEGER
□PRINT CHR$(12)
□PRINT
□PRINT
□PRINT
□FOR Z=1 TO 5
□PRINT TAB(10);
□FOR T=1 TO 12
□READ DAT
□PRINT CHR$(DAT);
□NEXT T
□PRINT
□NEXT Z
□END
□DATA 129,131,130,143,129,131,131,143,129,131,130,
143
□DATA 133,143,138,143,133,143,143,143,132,140,136,
143
□DATA 133,143,138,143,132,140,140,143,131,131,130,
143
□DATA 133,143,138,143,131,131,130,143,143,143,138,
143
□DATA 132,140,136,143,140,140,136,143,143,143,138,
143
```

Special Characters in High-Resolution

High-resolution graphics does not have graphic characters but it does have other international and special characters. These characters are represented by ASCII codes 128 through 159 as shown in the following table:

Table 9.4
High-Resolution Special Characters

Character	Hex Code	Decimal Code	Character	Hex Code	Decimal Code
Ç	80	128	ó	90	144
ü	81	129	æ	91	145
é	82	130	Æ	92	146
å	83	131	ô	93	147
ä	84	132	ö	94	148
à	85	133	ø	95	149
ā	86	134	û	96	150
ç	87	135	ù	97	151
ê	88	136	Ø	98	152
ë	89	137	Ö	99	153
è	8A	138	O	9A	154
ï	8B	139	§	9B	155
î	8C	140	£	9C	156
ß	8D	141	±	9D	157
Ä	8E	142	°	9E	158
Å	8F	143	*f*	9F	159

Medium-Resolution Graphics

For more sophisticated graphics operations, OS-9 has built-in graphics interface modules that provide a convenient way to access the graphics and joystick functions of the Color Computer 3. The required module for medium-resolution graphics is named GFX. It must be in your execution directory or resident in memory when called by BASIC09.

You can either install GFX in memory using the LOAD command, or wait until BASIC09 calls it for a graphics function. Once loaded, GFX resides in memory until you remove it using the OS-9 UNLINK command or the BASIC09 KILL command.

GFX has a number of functions that you pass to it as parameters with the RUN statement. For instance, the following statement clears the current graphics screen:

```
RUN GFX("CLEAR")
```

Other tasks need such parameters as position, color, and size. The following is a quick reference to all of the GFX functions. Each is explained in detail later:

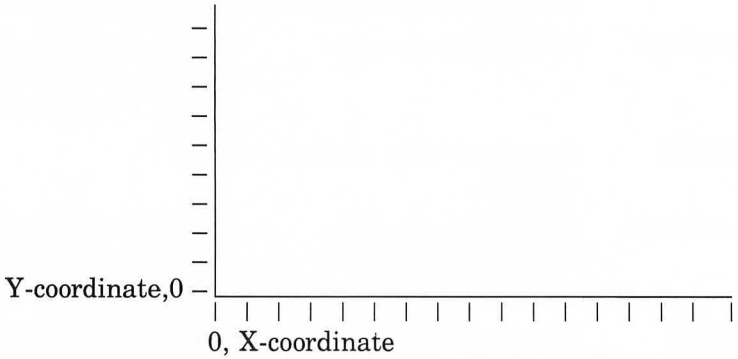
Function	Purpose	Parameters
ALPHA	Sets the screen to the alphanumeric mode.	None.
CIRCLE	Draws a circle.	Radius, optional X- and Y-coordinates, and color.
CLEAR	Clears the screen to a color.	Optional color for screen.
COLOR	Changes the foreground and background colors.	Foreground and background colors.
GCOLR	Reads a pixel's color.	Names of variables in which to store optional X- and Y-coordinates.
GLOC	Returns a video display address.	None.
JOYSTK	Returns the joystick button and X- and Y-coordinate status.	Names of variables in which to return the values.
LINE	Draws a line.	Ending X- and Y-coordinates, optional beginning coordinates, optional color.
MODE	Switches the screen between alphanumeric and graphics, sets the graphics screen color.	Format, Color.
MOVE	Positions the invisible graphics cursor.	X- and Y-coordinates.

Function	Purpose	Parameters
POINT	Moves graphics cursor and sets a point.	X- and Y-coordinates and optional pixel color.
QUIT	Returns screen to alphanumeric mode. Deallocates graphics memory.	None.

Formats and Colors

In medium-resolution graphics, you have a choice of two *formats*. Format 0 provides 256 horizontal points by 192 vertical points. In this format, you can have only two colors on the screen at a time.

Format 1 provides a 128 by 192 point screen and a maximum of four colors on the screen at a time. OS-9 medium-resolution graphics treats the screen as if it were a grid, with coordinate 0,0 at the lower left corner as shown in the following illustration. All points on the grid are positive.



BASIC09 defines colors with numbers or *color codes*. Many GFX functions allow or require color codes as parameters. BASIC09 also divides the color codes into *color sets*. Specifying a color code outside the current color set automatically initializes the new set.

Color Set	Color Code	Format 0		Color Code	Format 1	
		Back-ground	Fore-ground		Back-ground	Fore-ground
1	00	Black	Black	00	Green	Green
	01	Black	Green	01	Green	Yellow
	02			02	Green	Blue
	03			03	Green	Red
2	04	Black	Black	04	Buff	Buff
	05	Black	Buff	05	Buff	Cyan
	06			06	Buff	Magenta
	07			07	Buff	Orange
3				08	Black	Black
				09	Black	Dk Green
				10	Black	Md Green
				11	Black	Lt Green
4				12	Black	Black
				13	Black	Green
				14	Black	Red
				15	Black	Buff

Table 9.5

Use the preceding charts to chose colors for those functions that let you specify foreground or background colors. For instance, to initialize a Format 1 graphics screen with a green background and a red foreground, you type:

```
run gfx("mode",1,3)
```

The following reference section describes all the medium-resolution graphics functions, and provides examples and sample programs. To understand the organization of the commands reference, see "The Syntax Line" in Chapter 11.

The Draw Pointer

Medium-resolution graphics uses a *draw pointer*, or invisible graphics cursor, to determine what area of the screen is affected by graphics operations. When you establish a graphics screen, the draw pointer is located at coordinates 0,0. Some graphic functions automatically change the pointer location on the screen. For instance, the LINE function moves the draw pointer from the beginning coordinates to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the MOVE function to set the draw pointer to new locations.

ALPHA Select alphanumeric screen

Syntax: **RUN GFX("ALPHA")**

Function: Switches from the graphics screen to the alphanumeric (text) screen. The current graphics screen remains intact.

Parameters: None

Examples:

```
RUN GFX("ALPHA")
```

Sample Program:

This procedure lets you choose to draw a circle or rectangle of the size you select. Once you choose the shape and size, it uses the MODE function to select a graphics screen. When the shape is complete, you press [ENTER] to return to a text screen. The procedure uses the ALPHA function to return to the original menu.

```
PROCEDURE alpha
□DIM XCOR,YCOR,SIDE1,SIDE2,RADIUS,T,X,Y,Z:INTEGER
□DIM RESPONSE:STRING[1]
10 REPEAT
□SHELL "DISPLAY 0C"
□PRINT "Do you want to draw"
□PRINT "1) A rectangle"
□PRINT "2) A circle"
□PRINT "   -Press 1 or 2...";
□GET #0,RESPONSE
□PRINT
□IF RESPONSE="1" THEN
□INPUT "Length of Side 1...",SIDE1
□INPUT "Length of Side 2...",SIDE2
□RUN GFX("MODE",0,0)
□RUN GFX("CLEAR")
□XCOR=10
□YCOR=10
□RUN GFX("LINE",XCOR,YCOR,XCOR+SIDE1,YCOR,1)
```

```
RUN GFX("LINE",XCOR+SIDE1,YCOR,XCOR+SIDE1,YCOR+
SIDE2,1)
RUN GFX("LINE",XCOR+SIDE1,YCOR+SIDE2,XCOR,YCOR+
SIDE2,1)
RUN GFX("LINE",XCOR,YCOR+SIDE2,XCOR,YCOR,1)
INPUT RESPONSE
ELSE
IF RESPONSE="2" THEN
INPUT "What radius?...",RADIUS
RUN GFX("MODE",0,1)
RUN GFX("CLEAR")
RUN GFX("CIRCLE",128,90,RADIUS)
INPUT RESPONSE
ENDIF
ENDIF
UNTIL RESPONSE<>"1" AND RESPONSE<>"2"
RUN GFX("ALPHA")
GOTO 10
END
```

CIRCLE Draw a circle

Syntax: **RUN GFX("CIRCLE"[,*xcor,ycor*],*radius* [,*color*])**

Function: Draws a circle of a given radius. If you do not specify a color, BASIC09 uses the current foreground color. If you do not specify X- and Y-coordinates, CIRCLE uses the current graphics cursor position as the circle's center.

Parameters:

radius	The radius of the circle you want to draw.
color	The code of the color you want the circle to be. See the chart earlier in this section for color information.
xcor,ycor	The X- and Y-coordinates for the center of the circle. Specifying coordinates outside the X-coordinate range of 0-255 or outside the Y-coordinate range of 0-191 causes an error.

Examples:

```
RUN GFX("CIRCLE",100)

RUN GFX("CIRCLE",100,3)

RUN GFX("CIRCLE",125,100,100)

RUN GFX("CIRCLE",125,100,100,2)
```

Sample Program:

This procedure uses CIRCLE to draw and erase a circle. The location of the circle changes before each draw/erase operation, causing the circle to move. When it hits the edge of the screen, it reverses its direction at a random angle and *bounces*.

```
PROCEDURE circles
□DIM RADIUS,XCOR,YCOR:INTEGER
□DIM XTEMP,YTEMP:INTEGER
□DIM PATH1,PATH2:INTEGER
□DIM FLAG:INTEGER
```

```
□FLAG=1
□XCOR=5
□YCOR=5
□PATH1=RND(15)+2
□PATH2=RND(10)+2
□XTEMP=249
□YTEMP=185
□RUN GFX("MODE",0,1)
□RUN GFX("CLEAR")
□FOR T=1 TO 200
□WHILE XCOR<250 AND XCOR>4 AND YCOR<186 AND YCOR>4
DO
□RUN GFX("CIRCLE",XTEMP,YTEMP,3,0)
□RUN GFX("CIRCLE",XCOR,YCOR,3,1)
□XTEMP=XCOR
□YTEMP=YCOR
□XCOR=XCOR+PATH1
□YCOR=YCOR+PATH2
□ENDWHILE
□PATH1=RND(15)+2
□PATH2=RND(10)+2
□IF XCOR>249 THEN
□XCOR=249
□ENDIF
□IF XCOR<5 THEN
□XCOR=5
□ENDIF
□IF YCOR>185 THEN
□YCOR=185
□ENDIF
□IF YCOR<5 THEN
□YCOR=5
□ENDIF
□FLAG=FLAG*-1
□IF FLAG<0 THEN
□PATH1=PATH1*-1
□PATH2=PATH2*-1
□ENDIF
□NEXT T
□END
```

CLEAR Clear the screen

Syntax: **RUN GFX("CLEAR"[,*color*])**

Function: Clears the current graphics screen. If you do not specify a color, CLEAR sets the entire screen to the current background color. CLEAR also sets the graphics cursor at coordinates 0,0, the lower left corner of the screen.

Parameters:

color	A code indicating the color to set the screen.

Examples:

```
RUN GFX("CLEAR")

RUN GFX("CLEAR",14)
```

COLOR Change the foreground color

Syntax: **RUN GFX("COLOR",*color*)**

Function: Changes the foreground color (and possibly the color set). COLOR does not change the graphics format or the cursor position.

Parameters:

color	A code indicating the color you want for the foreground. See the chart earlier in this chapter for color information.

Examples:

```
RUN GFX("COLOR",10)
```

Sample Program:

This procedure connects a series of differently colored circles to produce a necklace effect.

```
PROCEDURE necklace
□DIM COLOR,T,U,J,R,FLAG,XCOR,YCOR:INTEGER
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□COLOR=1
□XCOR=1
□YCOR=1
□R=2
□FOR T=1 TO 6
□FOR J=1 TO 40
□XCOR=XCOR+1
□YCOR=YCOR+.8
□IF FLAG<0 THEN
□R=R-1
□ELSE
□R=R+1
□ENDIF
□COLOR=COLOR+1
□IF COLOR>3 THEN COLOR=1
```

```
□ENDIF
□RUN GFX("CIRCLE",XCOR,YCOR,R,COLOR)
□NEXT J
□FLAG=FLAG*-1
□NEXT T
□FOR U=1 TO 10000
□NEXT U
□END
```

GLOC Find the graphics screen location

Syntax: **RUN GFX("GLOC",*storage*)**

Function: Determines the location of the graphics screen in memory and returns the address in the specified variable. When you know the graphic screen address, you can use PEEK and POKE to perform special functions not available in the GFX module, such as filling a portion of the screen with a color or saving a graphics screen to disk.

OS-9 Level Two maps display screens into a program's address space before PEEK and POKE can operate on a display screen. This means that you must have at least eight kilobytes of free memory in the user's address space. Program and data memory requirements must not exceed 56 kilobytes.

Parameters:

storage	An integer or byte type variable in which GLOC stores the memory address of the graphics screen.

Examples:

```
RUN GFX("GLOC",location)
```

Sample Program:

This procedure uses the GLOC function to locate the current graphics screen, then uses POKE to *paint* a series of boxes on the screen.

```
PROCEDURE boxin
□DIM LOCATION,PLACE,COLOR,BEGIN,QUIT,X,TERMINATE,
LINE,T,J:INTEGER
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□RUN GFX("GLOC",LOCATION)
□LOCATION=LOCATION+100 \ PLACE=LOCATION
□BEGIN=1
□QUIT=80
```

```
□COLOR=255
□TERMINATE=10
□LINE=32
□FOR X=1 TO 4
□FOR T=1 TO QUIT
□FOR J=BEGIN TO TERMINATE
□POKE PLACE+J,COLOR
□NEXT J
□PLACE=PLACE+LINE
□NEXT T
□LOCATION=LOCATION+160
□BEGIN=BEGIN+1
□PLACE=LOCATION
□QUIT=QUIT-10
□TERMINATE=TERMINATE-1
□COLOR=COLOR-85
□NEXT X
□INPUT Z$
□END
```

JOYSTK Get joystick status

Syntax: **RUN GFX("JOYSTK",*stick,fire,xcor,ycor*)**

Function: Determines the status of the specified joystick fire button and the X,Y position of the specified joystick handle. Use this function only with a standard joystick or mouse, not with the high-resolution mouse adapter.

Parameters:

stick	The joystick (0 or 1) for which you want to determine the status. 0 indicates the right joystick, 1 indicates the left joystick.
fire	A variable in which JOYSTK returns the status of the specified fire button. *Fire* can be byte, integer, or Boolean type. A value other than 0 or TRUE indicates the button is pressed.
xcor,ycor	Byte or integer type variables in which JOYSTK stores the X- and Y-coordinates of the joystick handle position. The coordinate range is 0-63.

Examples:

```
RUN GFX("JOYSTK",0,shoot,x,y)
```

Sample Program:

This procedure uses the JOYSTK function to draw on the screen with the right joystick.

```
PROCEDURE joydraw
□DIM STICK,FIRE,XCOR,YCOR,XTEMP,YTEMP:INTEGER
□RUN GFX("MODE",0,1)
□RUN GFX("CLEAR")
□JOY=0 \XCOR=0 \YCOR=0
□REPEAT
□XTEMP=XCOR
□YTEMP=YCOR
□RUN GFX("JOYSTK",0,FIRE,XCOR,YCOR)
□XCOR=XCOR*4
□YCOR=YCOR*4
□RUN GFX("LINE",XTEMP,YTEMP,XCOR,YCOR)
□UNTIL FIRE<>0
□END
```

LINE Draw a line

Syntax: **RUN GFX("LINE"[,*xcor1,ycor1*],*xcor2,ycor2* [,*color*])**

Function: Draws a line in the current or specified foreground color in one of the following ways:

- From the current draw position to the specified X,Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X,Y-coordinates.

Parameters:

xcor1,ycor1	Are LINE's beginning X- and Y-coordinates.
xcor2,ycor2	Are LINE's ending X- and Y-coordinates.
color	A code indicating the color you want the line to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("LINE",192,128)

RUN GFX("LINE",0,0,192,128)

RUN GFX("LINE",0,0,192,128,2)
```

Sample Program:

This procedure draws a sine wave of vertical lines across the screen.

```
PROCEDURE waves
□DIM A,B,C,D,X,Y,Z:INTEGER
□CALC=0 \ A=100
□RUN GFX("mode",0,1)
□RUN GFX("CLEAR")
□RUN GFX("COLOR",2)
□FOR X=0 TO 255 STEP 1
□CALC=CALC+.05
□Y=A-SIN(CALC)*15
□Z=Y+25
□RUN GFX("LINE",X,Y,X,Z)
□NEXT X
□END
```

MODE Switch to graphics screen

Syntax: **RUN GFX("MODE",*format,color*)**

Function: Switches the screen from alphanumeric (text) to graphics, selecting the screen format and color code. You must run MODE before you can use any other graphics function. When you do, BASIC09 allocates a six-kilobyte block of memory for graphics. If your system does not have this amount of memory available, OS-9 returns an error message.

Parameters:

format	Either 0 (a two-color 256 by 192 pixel screen) or 1 (a four-color, 128 by 192 pixel screen).
color	A code indicating the color to set the screen. See the chart earlier in this chapter for information on color sets.

Examples:

```
RUN GFX("MODE",1,2)
```

MOVE Move graphics cursor

Syntax: **RUN GFX("MOVE",*xcor,ycor*)**

Function: Moves the invisible graphics cursor to the specified location on the screen. MOVE does not change the display in any way.

Parameters:

xcor,ycor	The coordinates for the cursor.

Examples:

```
RUN GFX("MOVE",192,128)
```

Sample Program:

This procedure draws and *pops* bubbles on the screen using the CIRCLE function. It uses MOVE to select the position for the circles.

```
PROCEDURE bubbles
□DIM XCOR,YCOR,T,R,ARRAY(3,100):INTEGER
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□FOR T=1 TO 20
□ARRAY(1,T)=RND(255)
□ARRAY(2,T)=RND(192)
□ARRAY(3,T)=RND(50)
□RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
□RUN GFX("CIRCLE",ARRAY(3,T),3)
□NEXT T
□FOR T=1 TO 20
□RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
□RUN GFX("CIRCLE",ARRAY(3,T),0)
□SHELL "DISPLAY 07"
□NEXT T
□END
```

POINT Set point to specified color

Syntax: **RUN GFX("POINT",*xcor,ycor*[,*color*])**

Function: Displays a dot at the specified coordinates. If you specify a color, POINT sets the pixel at the new coordinates to that color. Otherwise, POINT sets the pixel at the new coordinates to the foreground color.

Parameters:

xcor,ycor	The X- and Y-coordinates for a pixel.
color	The code of the color you want the pixel to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("POINT",192,128)

RUN GFX("POINT",192,128,2)
```

Sample Program:

This procedure uses POINT to draw filled boxes on the screen.

```
PROCEDURE boxup
□DIM XCOR,YCOR,BEGIN,COLOR,QUIT,TERMINATE,LINE:
INTEGER
□DIM T,X,Y:INTEGER
□XCOR=50 \YCOR=30 \COLOR=1
□BEGIN=1 \START=1 \QUIT=20 \TERMINATE=50
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□FOR T=1 TO 4
□FOR X=BEGIN TO QUIT
□FOR Y=START TO TERMINATE
□RUN GFX("POINT",XCOR+Y,YCOR,COLOR)
□NEXT Y
□YCOR=YCOR+1
□NEXT X
□START=START+10
```

```
□TERMINATE=TERMINATE-10
□COLOR=COLOR+1
□NEXT T
□INPUT Z$
□END
```

QUIT Deallocate graphics screen

Syntax: **RUN GFX("QUIT")**

Function: Switches the screen to the alphanumeric (text) mode and deallocates graphics memory.

Parameters: None

Examples:

```
RUN GFX("QUIT")
```

High-Resolution Graphics

BASIC09's high-resolution graphics greatly expand the capabilities of the Color Computer 3. You can have greater screen resolution (up to 640 by 192 pixels), as many as 64 colors, and the ability to mix graphics and text on one screen. In addition, you can use different text *fonts*, or styles.

The high-resolution module, GFX2, has many more functions than its medium resolution counterpart. GFX2 gives you the ability to:

- Select from 64 colors. OS-9 provides a palette with 16 default colors. You can change any of these default colors to any of the 64 colors available on the Color Computer 3.
- Set border colors.
- Set color patterns.
- Create different types of graphics screen cursors.
- Use logic functions.
- Turn an automatic scaling function off or on.
- Draw outline or filled boxes.
- Draw ellipses and arcs.
- Fill specified areas with specified colors.
- GET and PUT sections of the graphics screen.
- Select character fonts, which include boldfaced, transparent, and proportionally spaced characters.
- Move the cursor. Erase portions of a line or of the screen.
- Select reverse or normal video.
- Underline text.

Also, high-resolution graphics operate through the OS-9 Windowing System. This means that you can run several procedures in different *windows*. You can establish windows to display text, or to display graphics, or both. You can easily display any window.

Establishing a Hardware Window

For your convenience, OS-9 has a number of predefined or *hardware* window formats. Hardware windows are text windows, and you cannot use them for graphic applications. Because hardware windows are predefined, you can easily establish them with the INIZ command. For instance, to establish Window 7, type:

```
iniz w7 [ENTER]
```

However, you cannot see the window until you send a message to it. Type:

```
echo Hello Window 7 > /w7 [ENTER]
```

Now, to see the window and your message press [CLEAR]. To return to the original screen, press [CLEAR] again.

To OS-9, a window is a device and you can send data to it. To view the Errmsg file in the SYS directory of your system diskette, list it to Window 7 by typing:

```
list sys/errmsg > /w7 [ENTER]
```

Press [CLEAR] to move to Window 7 and see the listing. Press [SHIFT][CLEAR] to return to the previous screen.

You can also fork a shell (an execution environment) to a window. To cause a shell to operate in Window 7, type:

```
shell i=/w7& [ENTER]
```

The `i=` function of SHELL tells OS-9 that the window is *immortal.* It does not die after completing a task. To operate OS-9 from the window, press [CLEAR].

Besides Window 7, you have six other predefined windows. The following chart shows all the hardware windows and their parameters:

Window Number	Screen Size Chars/line	Starting Coordinates X-Coord, Y-Coord	Window Size Cols	Rows
1	40	0,0	27	11
2	40	28,0	12	11
3	40	0,12	40	12
4	80	0,0	60	11
5	80	60,0	19	11
6	80	0,13	80	12
7	80	0,0	80	24

Defining Windows

As well as hardware windows, OS-9 also lets you establish windows to your own specifications. You can set definable windows for either text or graphics, or both. You can locate them anywhere on a screen, and you can make them any size.

You initialize definable windows in the same manner you initialize hardware windows, using INIZ. If you want to have text on the window, you must merge SYS/Stdfonts (found on your system diskette) with the window. You can also establish a shell in a definable window, from which you can use OS-9 or BASIC09.

To establish definable windows you must supply OS-9 with information about the type of window you want (its graphic format), its size, and its location on the screen. The easiest way to do this is with the OS-9 WCREATE command.

WCREATE requires a window format code in the form `- s=format code` to tell OS-9 what type of a window you want. The following chart shows the possible window formats you can choose:

Table 9.6

Format Code	Screen Size Cols x Rows	Resolution Width/Height	No. of Colors	Memory Required	Screen Type
01	40 x 24	——	16†	1600	Text
02	80 x 24	——	16†	4000	Text
05	80 x 24	640 x 192	2	16000	Graphics
06	40 x 24	320 x 192	4	16000	Graphics
07	80 x 24	640 x 192	4	32000	Graphics
08	40 x 24	320 x 192	16	32000	Graphics
00*	Specifies the current screen.				
FF	Current display screen. Use when putting several windows on the same physical screen.				

† You have to reconfigure the palette to get 16 colors rather than the default of eight colors. The following section provides information on the palette.

Format Codes 01 and 02 select text screens, and Format Codes 5-8 select graphics screens. The Screen Size column shows the maximum number of text columns and rows available for each screen. The Resolution column shows the maximum *pixels* (graphic units) available for each of the graphic screens. The Memory column shows how much memory OS-9 must set aside for each screen format. Memory requirements depend on the resolution and number of colors selected for a window.

The Palette

BASIC09 has 64 colors you can select for screen displays. The colors are available through a *palette*. The Color Computer's palette can hold 16 colors at once.

The following chart shows the default colors for the palette in Screen Format 7:

Table 9.7

Register	Color	Register	Color
00	Black	**08**	Black
01	Red	**09**	Green
02	Green	**10**	Black
03	Yellow	**11**	Buff
04	Blue	**12**	Black
05	Magenta	**13**	Green
06	Cyan	**14**	Black
07	White	**15**	Orange

Instead of the default colors, you can select any of the 64 colors (0-63) for any of the palette registers. You do this using the PALETTE command described later in this chapter. The BORDER and COLOR commands also affect the colors available in the palette by changing the color in the background and foreground registers, Registers 02 and 03, respectively.

Note: The information in the next section assumes you have a Color Computer 3 with 512 kilobytes of memory. If your computer has 128 kilobytes of memory, skip to the section "High-Level Graphics With 128K."

Establishing a Graphics Window

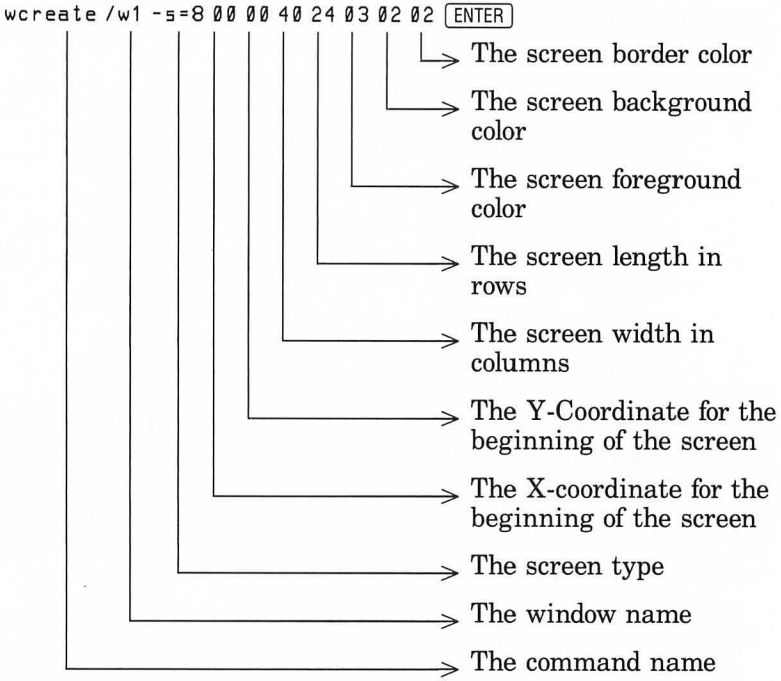
To create any window, you should first initialize it with the INIZ command. Type:

```
iniz w1 [ENTER]
```

So that you can later type in the new window, merge the Stdfonts file with it. Type:

```
merge sys/stdfonts>/w1 [ENTER]
```

Using the information in the preceding tables, use WCREATE to establish a graphics window. The following command line creates a graphics window in Window 1 that has 320 x 192 resolution and that fills the entire screen. The new window has 16 colors available and provides 40 column by 24 line text:



Starting a Shell in a Window

At this point, the new window exists, and you can send data to it. However, if you want to operate from the window, you must install a shell in it. Type:

```
shell i=/w1& ENTER
```

Press CLEAR to move to the new window. To load BASIC09, type:

```
basic09 #10K ENTER
```

Select either more or less memory, according to your needs. Using BASIC09 in a graphics window, you can write procedures to create high-resolution graphics, and you can display the graphics on the same screen.

Using High-Level Graphics With 128K

If your computer is equipped with only 128 kilobytes of memory, you cannot use more than one window with BASIC09. Also, to use even one window, you must follow certain steps to provide enough memory for BASIC09 operations.

Refer to Table 9.6. You must select a window mode that does not use more than 16000 byte of memory—either window Format 5 or Format 6.

To provide enough memory to use BASIC09, you must fork a shell to the window you create, then kill the shell in TERM. Doing this means that you can no longer operate from your TERM screen. However, you can run OS-9 and BASIC09 from the window.

The following steps show you how to create a Format 6 graphics screen in Window 1, write a BASIC09 high-resolution graphics procedure, and execute it using minimum memory.

1. Boot OS-9. Then, create a graphics window by typing:

```
iniz w1 [ENTER]
wcreate /w1 -s=06 00 00 40 24 06 01 01 [ENTER]
merge sys/stdfonts>/w1 [ENTER]
shell i=/w1& [ENTER]
ex [ENTER]
```

2. The system stops, and you can no longer type or issue commands. Press [CLEAR] to move to the new window. Then, load BASIC09 by typing:

```
basic09 [ENTER]
```

3. Enter the edit mode, and type the following procedure:

```
PROCEDURE squeeze
□DIM XCOR,YCOR,X,Y:INTEGER; RESPONSE:STRING[1]
□RUN GFX2("CUROFF")
□XCOR=320 \ YCOR=95 \ X=300 \ FLAG=1
□PRINT CHR$(12)
□LOOP
□FOR Y=1 TO 100 STEP 2
□X=X-3
□GOSUB 10
□IF FLAG<1 THEN
□RUN GFX2("COLOR",0)
```

```
□ELSE
□RUN GFX2("COLOR",3)
□ENDIF
□RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
□FLAG=FLAG*-1
□NEXT Y
□RUN GFX2("COLOR",1)
□FOR Y=99 TO 1 STEP -2
□GOSUB 10
□X=X+3
□RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
□NEXT Y
□RUN GFX2("COLOR",0)
□ENDLOOP
10□RUN INKEY(RESPONSE)
□IF RESPONSE="" THEN
□RETURN
□ENDIF
10□PRINT CHR$(12)
□RUN GFX2("COLOR",0)
□RUN GFX2("CURON")
□END
```

4. When you have entered the procedure exactly as shown, exit the edit mode, and from the BASIC09 command mode, save Squeeze by typing:

   ```
   save squeeze [ENTER]
   ```

5. Compile Squeeze by typing:

   ```
   pack squeeze [ENTER]
   ```

   Squeeze is now an executable module saved in your current execution directory. The following steps assume your execution directory is /D0/CMDS.

6. Exit BASIC09 by typing:

   ```
   bye [ENTER]
   ```

7. Merge Squeeze, RUNB, INKEY, and GFX2 into one module. To do this, type:

   ```
   merge /d0/cmds/squeeze /d0/cmds/runb /d0/cmds/
   inkey gfx2 > /d0/cmds/yawn [ENTER]
   ```

8. MERGE does not set the new file Yawn as an executable file. Before you execute it, you must make the file executable by typing:

```
attr /d0/yawn e pe [ENTER]
```

9. To execute Yawn, type:

```
yawn [ENTER]
```

10. To terminate the procedure, press the space bar.

The merging procedure in Step 7 saves a considerable amount of memory. Every module you load uses one or more 8-kilobyte blocks of storage space. For instance, INKEY is only 94 bytes in length. However, if you load it as a separate module, it requires 8192 bytes. RUNB is 12185 bytes in length. This means that it requires two 8-kilobyte blocks, or 16384 bytes of memory. GFX2 is 2190 bytes in length, and Squeeze is 605 bytes in length. Loaded individually, they also require two memory blocks.

If you load all four modules independently, they use 40960 bytes. However, by combining them into one file, they load into two memory blocks, or 16384 bytes.

Using the information in this section, you can write and execute numerous BASIC09 procedures with only 128 kilobytes of memory. However, if your computer has 512 kilobytes of memory, you can bypass many of these steps. Also, the additional memory enables you to have several windows open at one time. For instance, you can create one window in which to write BASIC09 procedures, another window in which to execute your procedures, and a third window from which you can use OS-9 commands.

> **Note:** The remainder of this chapter assumes you have 512 kilobytes of memory. If you don't, you can still run many of the sample procedures by implementing the steps in this section.

Creating Windows from BASIC09

Using GFX2 routines, BASIC09 provides the means to create and manage windows. The steps for creating windows from BASIC09 are as follows:

1. DIM a variable to hold the path number to the window you want to create.

2. OPEN a path to the window.
3. SELECT the new window as the display window.
4. Send commands, data, or text to the window through the open path.
5. CLOSE the open path.
6. Use SELECT to return to your original window.

If you do not want to return immediately to the screen or window of origin, you can skip Steps 5 and 6.

The following sample procedure shows how to open Window 2 as a 320 x 192 graphics window, draw a circle, then return to the original screen when you press a key.

```
PROCEDURE make_win
DIM PATH:INTEGER
DIM RESPONSE:STRING[1]
OPEN #PATH,"/W2":WRITE
RUN GFX2 (PATH,"DWSET",08,00,00,40,24,03,02,02)
RUN GFX2 (PATH,"SELECT")
RUN GFX2 (PATH,"CIRCLE",200,90,80)
GET #1,RESPONSE
CLOSE #PATH
RUN GFX2 ("SELECT")
END
```

This procedure establishes a Format 8 window, beginning at Coordinates 0,0 and covering the total screen. The foreground color is green, the background color is black, and the border color is black.

Because this procedure does not INIZ the window it opens, the window automatically disappears when the procedure closes its path. To create a window that stays in the system, even after you close the path to it, use INIZ before the OPEN statement, like this:

```
SHELL "INIZ /W2"
```

After you create and define the window, view it by pressing CLEAR. To get back to the screen you are working on, press SHIFT CLEAR. If you intend to use a window more than once in a procedure, you do not need to close its path until the procedure no longer needs it.

Creating Overlay Windows

When you establish a window, you are initializing an OS-9 device. However, an overlay window is only a new screen for an existing window. An overlay screen can be the same size as its window, or it can be smaller. OS-9 automatically transfers to the overlay window any current procedures operating in the device window.

The process for creating overlay windows lets you select whether you want to save the contents of the screen covered by the new window. If you choose to save the contents, the previous screen is redisplayed when you end the overlay.

The following procedure provides an example of using overlay windows. It creates six overlays, each smaller than the preceding window. The procedure then waits for you to press a key. When you do, it removes the overlay windows.

```
PROCEDURE overwindows
□DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
□DIM RESPONSE:STRING[1]
□X=0 \Y=0
□X1=80 \Y1=24
□PLACE=33
□FOR T=1 TO 6
□IF T=2 OR T=6 THEN
□B=3
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□X=X+6 \Y=Y+2
□X1=X1-12 \Y1=Y1-4
□FOR J=1 TO 5
□PRINT TAB(PLACE); "Overlay Screen "; T
□NEXT J
□PLACE=PLACE-6
□NEXT T
□PRINT "Overlay Screen 6"
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 6
□RUN GFX2("OWEND")
□NEXT T
□END
```

The Graphics Cursor and the Draw Pointer

High-resolution graphics provide a text cursor, a graphics cursor, and a *draw pointer*. The text cursor and the graphics cursor can be either visible or invisible. The draw pointer is always invisible.

Text functions always begin at the current location of the text cursor. Whenever you *print* on the screen, the cursor automatically moves to the end of the text or to the beginning of the next line, depending on whether or not you use a semicolon after the print statement. You can reset the text cursor to any place on the screen with the CURXY function of GFX2.

Many BASIC09 graphics functions also begin operating at a location pointed to by the draw pointer. When you begin graphics, the draw pointer is located at coordinates 0,0. BASIC09 then updates the pointer as you execute certain graphics functions. For instance, the LINE function of GFX2 draws from the draw pointer position to the specified end coordinates. The draw pointer is left pointing to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the SETDPTR function to move the draw pointer to new locations.

The graphics cursor is for use with joystick or mouse operations. It provides a *pointer* for graphics applications. The system diskette provides patterns that can be loaded into the graphics cursor *buffer*. You can select from a variety of pointer images.

High-Resolution Text

When you create a graphics window, you can display either text characters, graphics characters, or both.

To display graphics, move the draw pointer to the location where you want the graphics to begin. Then, execute the graphics routines.

To display text, move the text cursor to the location where you want the text to begin. Then, use normal BASIC commands to *print* text.

Instructions for the draw pointer relate to a 640 x 192 grid, numbered 0-639 and 0-191. Instructions for the text cursor relate to the number of characters per line and the number of lines on the current screen format.

Using Fonts

OS-9 has built-in fonts (character sets). You can also create your own fonts and instruct BASIC09 to use them. If you create your own fonts, you can design any symbols or graphics characters you want to use.

To use fonts, you must be in a graphics window. See "Establishing a Graphics Screen" earlier in this chapter. Use the FONT function to tell OS-9 what font you want. BASIC09 has three fonts installed in Group 200, Buffers 1, 2, and 3. The following procedure uses characters in Buffer 3 to draw a border, then prints a message using the characters in Buffer 2. It then returns to Buffer 3 and asks you to press a key to end the procedure.

```
PROCEDURE borders
□DIM T,B,V,J,K:INTEGER
□DIM RESPONSE:STRING[1]
□B=199
□PRINT CHR$(12)
□RUN GFX2("FONT",200,3)
□RUN GFX2("COLOR",1,2)
□FOR T=0 TO 79
□PRINT CHR$(B);
□NEXT T
□FOR T=1 TO 21
□RUN GFX2("CURXY",0,T)
□PRINT CHR$(B); CHR$(B);
□RUN GFX2("CURXY",78,T)
□PRINT CHR$(B); CHR$(B);
□NEXT T
□RUN GFX2("CURXY",0,21)
□FOR T=0 TO 79
□PRINT CHR$(B);
□NEXT T
□RUN GFX2("FONT",200,2)
□RUN GFX2("COLOR",0,2)
□RUN GFX2("CURXY",45,9)
□PRINT "A Demonstration"
```

```
RUN GFX2("CURXY",50,10)
PRINT "Of A"
RUN GFX2("CURXY",43,11)
PRINT "Buffer Three Border"
RUN GFX2("CURXY",51,12)
PRINT "And"
RUN GFX2("CURXY",45,13)
PRINT "Buffer Two Text"
RUN GFX2("FONT",200,1)
RUN GFX2("COLOR",3,2)
RUN GFX2("CURXY",33,15)
PRINT "Press A Key...";
GET #1,RESPONSE
PRINT CHR$(12)
END
```

High-Resolution Quick Reference

High-resolution functions are all part of the GFX2 module. You call them in a BASIC09 procedure with the following syntax:

```
RUN GFX2([PATH],"FUNCTION"[,PARAMETER[,...]])
```

Path is an optional variable name that tells OS-9 the window in which you want the function performed. *Function* is the high-resolution task you want to perform. *Parameter* is an essential or optional value that affects the performance of the function. Different functions require or permit different numbers of parameters.

The following reference gives a brief description of the high-resolution graphics functions. This list is organized by function. Following the quick reference is a detailed reference organized alphabetically.

Window Commands

Command	Function
DWSet	Establishes a window and sets its location on the screen, its size, its background color, its foreground color, and its border color.
OWSet	Establishes an overlay window on a device window that already exists. The function also sets the overlay window size, background color, foreground color, and border color. When using this function, you can choose whether or not to save the contents of the original screen.
OWEnd	Deallocates the specified overlay window.
Select	Selects the window to display.
DWEnd	Deallocates an established window.
CWArea	Changes the size of a window. You can only reduce the working area of a window, not increase it.
DWProtectSw	Lets you unprotect a window and set other device windows over it. This might destroy the contents of either or both windows.

Drawing Commands:

Command	Function
Point	Sets the pixel under the draw pointer to the specified color or to the default color.
Line	Draws a line.
Box	Draws a rectangle outline.
Bar	Draws a filled rectangle.
Circle	Draws a circle.
Ellipse	Draws an ellipse.
Arc	Draws an arc.
Fill	Fills the area of the window the same color as the pixel under the draw pointer.
Clear	Clears the window.

Configuring Commands:

Command	Function
Color	Sets any of the foreground, background, or border colors.
DefCol	Sets palette registers to the default colors.
Border	Sets the border palette register.
Palette	Changes colors in the palette registers.
Pattern	Establishes a buffer from which BASIC09 gets a pattern for graphics functions.
Logic	Turns on AND, OR, or XOR logic functions for draw functions.
GCSet	Establishes a buffer from which BASIC09 gets the graphics cursor.
ScaleSw	Turns scaling on or off.
SetDPtr	Positions the draw pointer.
PutGC	Positions the graphics cursor.
Draw	Draws an image from directions provided in a draw string.

Get/Put Commands:

Command	Function
Get	Saves a specified portion of a window to a buffer.
Put	Places the image stored in a buffer onto a window.
DefBuff	Defines a buffer for storage.
GPLoad	Preloads a buffer from a disk file.
KillBuff	Deallocates a buffer.

Text/Cursor Handling Routines:

Command	Function
CurHome	Positions the cursor at coordinates 0,0.
CurXY	Positions the cursor at specified coordinates.
ErLine	Erases the line under the cursor.
ErEOLine	Erases from the cursor to the end of the line.
CurOff	Turns the graphics cursor off.
CurOn	Turns the graphics cursor on.
CurRgt	Moves the graphics cursor right one space.
Bell	Sounds the terminal bell.
CurLft	Moves the graphics cursor left one space.
CurUp	Moves the graphics cursor up one line.
CurDwn	Moves the graphics cursor down one line.

Font Handling Commands:

Command	Function
Font	Specifies the buffer from which BASIC09 selects its font characters.
TCharSw	Selects or deselects transparent characters.
BoldSw	Selects or deselects bold characters.
PropSw	Selects or deselects proportional characters.
ErEoWndw	Erases from the graphics cursor to the end of the window.
Clear	Erases window and homes the cursor.
CrRtn	Performs a carriage return by moving the cursor down one line and to the extreme left of the window.
ReVOn	Turns reverse video on.
ReVOff	Turns reverse video off.
UndlnOn	Turns the underline function on.
UndlnOff	Turns the underline function off.
BlnkOn	Turns blinking characters on (only for hardware text screens).
BlnkOff	Turns blinking characters off (only for hardware text screens).
InsLin	Inserts a blank line at the graphics cursor position.
DelLin	Deletes the line at the graphics cursor position.

ARC Draw an arc

Syntax: **RUN GFX2([*path*,]"ARC"[,*mx,my*], *xrad,yrad,xcor1,ycor1,xcor2, ycor2*)**

Function: Draws an arc at the current or specified draw position with the specified X and Y radius. If you specify the same radius for both X and Y, the function draws a circular arc, otherwise the arc is elliptical. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

ARC begins drawing from the point on the screen closest to the first set of coordinates (xcor1, ycor1). It stops at the portion of the screen closest to the second set of coordinates (xcor2, ycor2). You can determine on which side of the line ARC draws by selecting which set of coordinates is the beginning and which set is the end.

Parameters:

path	The route to the window in which you want to draw an arc.
mx,my	The X- and Y-coordinates for the center of the arc. If you do not specify *mx* and *my*, BASIC09 uses the current draw pointer position.
xrad	The radius of the arc's width.
yrad	The radius of the arc's height.
xcor1,ycor1 *xcor2,ycor2*	The beginning and ending coordinates for an imaginary line from which the function draws an arc. The line is relative to the center of the arc (the center point is at 0,0 for these coordinates) and extends through the two coordinates from one edge of the screen to the other.

Examples:

```
RUN GFX2("ARC",50,100,50,100,50,150)
```

Sample Program:

This procedure draws a series of diagonally-cut arcs on a graphics window screen.

```
PROCEDURE arcing
□DIM MX,MY,XRAD,YRAD,XCOR,YCOR,XCOR2,YCOR2:INTEGER
□DIM T,X,Y,Z:INTEGER
□PRINT CHR$(12)
□FOR T=1 TO 90 STEP 2
□RUN GFX2("ARC",318,95,150,T,0,1,0,1)
□RUN GFX2("ARC",324,95,150,T,1,0,1,1)
□NEXT T
```

BAR Fill a rectangle

Syntax: **RUN GFX2([*path*,]"BAR"[,*xcor1,ycor1*],*xcor2, ycor2*)**

Function: Fills a rectangular area defined by two sets of coordinates. BAR defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path	The route to the window in which you want to draw a bar.
xcor1,ycor1	The beginning coordinates of the line defining the area to fill. If you omit these coordinates, BAR uses the draw pointer position. See the previous section "The Graphics Cursor and The Draw Pointer." Also see SETDPTR.
xcor2,ycor2	The ending coordinates of the line defining the area to fill.

Examples:

```
RUN GFX2("BAR",200,100)

RUN GFX2("BAR",0,0,100,50)
```

Sample Program:

This procedure draws a bar chart on a window screen.

```
PROCEDURE OSgraf
□DIM COLOR,T,X,XCOR1,YCOR1,XCOR2,YCOR2:
INTEGER; RESPONSE:STRING[1]
□PRINT CHR$(12)
□RUN GFX2("DEFCOL")
□COLOR=13 \ XCOR1=10 \ YCOR1=180
□XCOR2=XCOR1+40
□RUN GFX2("CUROFF")
```

```
FOR T=1 TO 10
READ YCOR2
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",XCOR1,YCOR1,XCOR2,YCOR2)
RUN GFX2("COLOR",7)
RUN GFX2("BOX",XCOR1,YCOR1,XCOR2,YCOR2)
COLOR=COLOR+1 \ XCOR1=XCOR1+50 \ XCOR2=XCOR1+40
NEXT T
PRINT \ PRINT "    OS-9 Sales Chart"
RUN GFX2("BOX",0,0,510,180)
GET #1,RESPONSE
RUN GFX2("CURON")
PRINT CHR$(12)
END
DATA 170,150,140,130,110,90,70,60,50,30
```

BELL Ring the terminal bell

Syntax: **RUN GFX2("BELL")**

Function: Rings the terminal's *bell* (produces a beep through the speaker).

Parameters: None

Examples:

```
RUN GFX2("BELL")
```

BLNKON Character blink on
BLNKOFF Character blink off

Syntax: **RUN GFX2([*path*,]"BLNKON")**
RUN GFX2([*path*,]"BLNKOFF")

Function: Executing BLNKON causes all subsequent characters sent to a window on a *hardware* screen to blink. A hardware screen is one of the predefined device windows /W1 through /W7. Executing BLNKOFF cancels a previous blink command; characters already blinking continue to do so. Blink does not operate on graphics windows.

Parameters:

path	The route to the window in which you want to blink characters.

Examples:

```
RUN GFX2("BLNKON")

RUN GFX2("BLNKOFF")
```

BOLDSW Switch bold characters on or off

Syntax: **RUN GFX2([*path*,]"BOLDSW","*switch*")**

Function: Causes characters to display in either regular or bold typeface. The default is regular typeface. BOLD only works on graphics screens.

Parameters:

path	The route to the window in which you want bold characters.
switch	Can be either "ON" or "OFF." If *switch* is "ON," subsequent characters are bold. If *switch* is "OFF," subsequent characters are not bold.

Examples:

```
RUN GFX2("BOLDSW","ON")
```

Sample Program:

This procedure demonstrates the BOLDSW function by displaying both bold and normal text on a window screen.

```
PROCEDURE bold
□DIM LINE:STRING
□DIM LETTER:STRING[1]
□DIM T,J,K,FLAG:INTEGER
□RUN GFX2("CLEAR")
□FLAG=1
□FOR T=1 TO 8
□READ LINE
□FOR J=1 TO LEN(LINE)
□LETTER=MID$(LINE,J,1)
□IF LETTER<>"!" AND LETTER<>"#" THEN
□PRINT LETTER;
□ENDIF
□IF LETTER="!" THEN
□FLAG=FLAG*-1
```

```
IF FLAG>0 THEN
RUN GFX2("BOLDSW","OFF")
ELSE
RUN GFX2("BOLDSW","ON")
ENDIF
ENDIF
IF LETTER="#" THEN
PRINT CHR$(34);
ENDIF
NEXT J
PRINT
NEXT T
PRINT \ PRINT
END
DATA "This is a demonstration of"
DATA "the !Bold! function of"
DATA "BASIC09's GFX2 module."
DATA "Use the command"
DATA "!RUN GFX2(#BOLDSW#,#ON#)!"
DATA "to turn boldface on."
DATA "Use !RUN GFX2(#BOLDSW#,#OFF#)!"
DATA "to turn boldface off"
```

BORDER Set the border color

Syntax: **RUN GFX2([*path*,]"BORDER",*color*)**

Function: Resets the palette register that affects a window's border color (Register 0) to the specified color code. For information on the palette and on screen colors, see "The Palette" and Table 9.7 earlier in this chapter.

Parameters:

path	The route to the window in which you want to change border color.
color	One of the current palette colors. *Color* can be either a constant or a variable.

Examples:

```
RUN GFX2("BORDER",1)
```

Sample Program:

This procedure lets you select different border colors by pressing + or − to select higher or lower color codes. Press q to end the procedure.

```
PROCEDURE border
□DIM COLOR:INTEGER
□DIM KEY:STRING[1]
□COLOR=8
□RUN GFX2("CLEAR")
□WHILE KEY<>"q" AND KEY<>"Q" DO
□GET #1,KEY
□IF KEY="-" OR KEY="=" THEN
□COLOR=COLOR-1
□ENDIF
□IF KEY="+" OR KEY=";" THEN
□COLOR=COLOR+1
□ENDIF
```

```
□IF COLOR>15 OR COLOR<0 THEN COLOR=8
□ENDIF
□RUN GFX2("BORDER",COLOR)
□RUN GFX2("CURXY",0,0)
□ENDWHILE
□END
```

BOX Draw a rectangle

Syntax: **RUN GFX2([*path*,]"BOX"[,*xcor1,ycor1*], *xcor2,ycor2*)**

Function: Draws a rectangle. BOX defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. BOX does not reset the draw pointer. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path	The route to the window in which you want to draw a box.
xcor1,ycor1	The beginning coordinates for the line that defines the rectangle to drawn. If you omit these coordinates, BOX uses the draw pointer position.
xcor2,cor2	The ending coordinates for the line that defines the rectangular area to be drawn.

Examples:

```
RUN GFX2("BOX",200,100)

RUN GFX2("BOX",0,0,100,50)
```

Sample Program:

This procedure draws a series of progressively smaller boxes of different colors on a window screen. Then, it rapidly changes the colors of the boxes to produce a hypnotic effect.

```
PROCEDURE hypbox
□DIM X,Y,X1,Y1,T,R,COLOR:INTEGER
□DIM KEY:STRING[1]
□KEY=""
□X=18 \Y=6
□Y1=185 \X1=621
□RUN GFX2("CLEAR")
```

```
□FOR T=0 TO 15
□COLOR=T
□RUN GFX2("COLOR",3)
□RUN GFX2("BOX",X,Y,X1,Y1)
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",X-1,Y-1)
□X=X+18 \Y=Y+6
□X1=X1-18 \Y1=Y1-6
□NEXT T
□WHILE KEY="" DO
□RUN INKEY(KEY)
□FOR T=1 TO 16
□R=RND(65)
□RUN GFX2("PALETTE",T,R)
□NEXT T
□ENDWHILE
□RUN GFX2("DEFCOL")
□END
```

CIRCLE Draw a circle

Syntax: **RUN GFX2([*path*,]"CIRCLE"[,*xcor,ycor*], *radius*)**

Function: Draws a circle with a specified radius. If you specify coordinates, CIRCLE uses them for the center point. Otherwise, CIRCLE locates the center of the circle at the current draw pointer position. See "The Graphics Cursor and the Draw Pointer" earlier in this section. Also see SETDPTR.

Parameters:

path	The route to the window in which you want to draw a circle.
xcor,ycor	The coordinates for the circle's center. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.
radius	The radius of the circle.

Examples:

```
RUN GFX2("CIRCLE",100)

RUN GFX2("CIRCLE",100,200,50)
```

Sample Program:

This procedure uses circles to produce a geometric design.

```
PROCEDURE ciraround
□DIM T,X,Y:INTEGER
□PRINT CHR$(12)
□RUN GFX2("COLOR",1,2)
□FOR T=1 TO 130
□X=150*SIN(T)+320
□Y=25*COS(T)+96
□RUN GFX2("CIRCLE",X,Y,100)
□NEXT T
□RUN GFX2("COLOR",3,2)
□FOR T=1 TO 45
□X=150*SIN(T)+320
□Y=25*COS(T)+96
□RUN GFX2("CIRCLE",X,Y,100)
□NEXT T
□END
```

CLEAR Clear the screen

Syntax: **RUN GFX2([*path*,]"CLEAR")**

Function: Clears the current working area of a window. CLEAR does not change the location of the draw pointer but does set the text cursor and graphics cursor location to the upper left corner of the window.

Parameters:

path	The route to the window you want to clear.

Examples:

```
RUN GFX2("CLEAR")
```

COLOR Set screen colors

Syntax: **RUN GFX2([*path*,]"COLOR",**
***foreground*[,*background*][,*border*])**

Function: Changes any of the foreground, background, or the border colors. COLOR does not change the draw pointer position.

Parameters:

path	The route to the window in which you want to change one or more screen or text colors.
foreground	The register number for the foreground palette.
background	The register number for the background palette.
border	The register number for the border palette. Changing the border color for any window on a screen, changes the border color for all windows on the same screen.

Examples:

```
RUN GFX2("COLOR",1)
RUN GFX2("COLOR",1,2)
RUN GFX2("COLOR",1,2,1)
```

Sample Program:

This procedure fills a window screen with multicolored filled circles.

```
PROCEDURE bubbles
DIM X,Y,W,Z,T:INTEGER
Z=1
RUN GFX2("COLOR",1,0,0)
RUN GFX2("CLEAR")
FOR T=1 TO 80
X=RND(635)+4
Y=RND(185)+5
W=RND(50+5)
Z=Z+1
IF Z>3 THEN Z=1
ENDIF
RUN GFX2("CIRCLE",X,Y,W)
RUN GFX2("COLOR",Z)
RUN GFX2("FILL",X,Y)
NEXT T
RUN GFX2("COLOR",3,2,2)
END
```

CRRTN Carriage return

Syntax: **RUN GFX2([*path*,]"CRRTN")**

Function: Causes BASIC09 to send a carriage return to a window. The cursor moves down one line and to the extreme left of the window.

Parameters:

path	The route to the window in which you want a carriage return.

Examples:

```
RUN GFX2("CRRTN")
```

CURDWN Cursor down

Syntax: **RUN GFX2([*path*,]"CURDWN")**

Function: Moves the cursor down one text line. The X-coordinate, or column position, remains the same.

Parameters:

path	The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURDWN")
```

CURHOME Cursor home

Syntax: **RUN GFX2([*path*,]"CURHOME")**

Function: Moves the text cursor to the top left corner of the screen.

Parameters:

path	The route to the window where you want to reset the cursor

Examples:

```
RUN GFX2("CURHOME")
```

CURLFT Move cursor left

Syntax: **RUN GFX2([*path*,]"CURLFT")**

Function: Moves the cursor one character to the left.

Parameters:

path The route to the window where you want to move the cursor.

Examples:

```
RUN GFX2("CURLFT")
```

CUROFF Turn off cursor

Syntax: **RUN GFX2([*path*,]"CUROFF")**

Function: Makes the cursor invisible.

Parameters:

path The route to the window in which you want to turn the cursor off.

Examples:

```
RUN GFX2("CUROFF")
```

CURON Turn on cursor

Syntax: **RUN GFX2([*path*,]"CURON")**

Function: Makes the text cursor visible.

Parameters:

path	The route to the window in which you want to turn the cursor on.

Examples:

```
RUN GFX2("CURON")
```

CURRGT Move cursor right

Syntax: **RUN GFX2("[*path*,]CURRGT")**

Function: Moves the cursor one character to the right.

Parameters:

path	The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURRGT")
```

CURUP Move cursor up

Syntax: **RUN GFX2([*path*,]"CURUP")**

Function: Moves the cursor up one line.

Parameters:

path	The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURUP")
```

CURXY Set cursor position

Syntax: **RUN GFX2([*path*,]"CURXY",*column,row*)**

Function: Moves the cursor to the specified column and row position. The column and row coordinates are relative to the window's current character width and depth.

Parameters:

path	The route to the window in which you want to move the cursor.
column	The column (horizontal) position for the cursor.
row	The row (vertical) position for the cursor.

Examples:

```
RUN GFX2("CURXY",10,10)
```

CWAREA Change working area

Syntax: **RUN GFX2([*path*,]"CWAREA",*xcor,ycor,sizex, sizey*)**

Function: Restricts output in the window to the specified area. The new area must be the same or smaller than the previous working area. When a window's working area is changed, OS-9 scales graphic and text coordinates and graphic images to the new proportions. Text characters remain the same size.

Parameters:

path	The route to the window in which you want to change the working area.
xcor,ycor	The beginning coordinates (the upper left corner) for the new working area, relative to the original window. The coordinates are based on the character column and row size of the original window.
sizex	Designates the number of columns in the new working area.
sizey	The number of lines available in the new working area.

Examples:

```
RUN GFX2("CWAREA",10,0,40,10)
```

Sample Program:

This procedure makes the working area in a window progressively smaller, filling each area with a different color. It then changes the areas' colors rapidly to produce a hypnotic effect.

```
PROCEDURE hypnobox
□DIM X,Y,X1,Y1,T,R,COLOR:INTEGER
□DIM KEY:STRING[1]
□KEY=""
□X=3 \Y=1
□X1=80-(X+X) \Y1=24-(Y+Y)
□FOR T=0 TO 10
□RUN GFX2("COLOR",3,T)
□RUN GFX2("CLEAR")
□RUN GFX2("CWAREA",X,Y,X1,Y1)
□X=X+3 \Y=Y+1
□X1=80-(X+X) \Y1=24-(Y+Y)
□NEXT T
□RUN GFX2("COLOR",3,2)
□WHILE KEY="" DO
□RUN INKEY(KEY)
□FOR T=1 TO 16
□R=RND(65)
□RUN GFX2("PALETTE",T,R)
□NEXT T
□ENDWHILE
□RUN GFX2("DEFCOL")
□RUN GFX2("CWAREA",0,0,80,24)
□END
```

DEFBUFF Define GET/PUT buffer

Syntax: **RUN GFX2("DEFBUFF",*group,buffer,size*)**

Function: Defines a buffer for GET/PUT operations.

When you define a buffer, you do so by group number and buffer number. Each group you define allocates eight kilobytes of memory. The system needs 30 bytes of the block for overhead, leaving 8162 bytes free. Within the group, you can allocate one or more buffers. Select a group number and a buffer number as indicated in the following "Parameters" section. Use these numbers in future references to the buffer.

A GET/PUT buffer remains allocated until you use the KILLBUFF function to remove it from your system's memory. For more information on Get/Put buffers, see KILLBUFF, PUT, GET, and GPLOAD.

Parameters:

group	A number you select in the range 1-199.
buffer	A number (in the range 1-255) that you assign to the buffer you create.
size	The size of the buffer, in the range of 1 to 8192 bytes, depending on available memory in its group.

Notes:

One method of selecting a group number is to use SYSCALL and the Get ID (103F 0C) system call to obtain your user's process ID number. Then, use this ID number as a group number. Using this system for all GET/PUT buffer operations, ensures against group number overlapping. See the SYSCALL command for more information.

Examples:

```
RUN GFX2("DEFBUFF",1,5,4000H0)
```

DEFCOL Set default colors

Syntax: **RUN GFX2([*path*,]"DEFCOL")**

Function: Sets the palette registers back to their default values. The type of monitor you have determines the actual hues. See "The Palette" and Table 9.7 earlier in this section.

Parameters:

path	The route to the window in which you want to restore the original palette registers.

Examples:

```
RUN GFX2("DEFCOL")
```

DELLIN Delete current line of text

Syntax: **RUN GFX2([*path*,]"DELLIN")**

Function: Deletes the line on which the cursor is resting and closes the space. DELLIN operates on both text and graphics screens.

Parameters:

path	The route to the window in which you want to delete a line.

Examples:

```
RUN GFX2("DELLIN")
```

Sample Program:

This procedure draws a series of various colored concentric circles, then produces a lemon shape by removing slices of the circle with DELLIN.

```
PROCEDURE slice
□DIM X,Y,R,T,COLOR:INTEGER
□RUN GFX2("CLEAR")
□COLOR=0
□X=320
□Y=96
□FOR T=185 TO 10 STEP -10
□RUN GFX2("CIRCLE",X,Y,T)
□NEXT T
□FOR T=140 TO 320 STEP 10
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",T,96)
□COLOR=COLOR+1
□NEXT T
□RUN GFX2("CURXY",0,8)
□FOR T=1 TO 8
□RUN GFX2("DELLIN")
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

DRAW Draw a polyline figure

Syntax: **RUN GFX2([*path*,]"DRAW",*option list*)**

Function: Draws in the directions specified, and for the distances specified, in an option list. The option list is a string of characters and numbers. You can separate options with spaces or commas. You must include commas between the two coordinates for the B and U options.

Parameters:

path	The route to the window in which you want to draw.
option list	A string consisting of one or more of the following options:

Options:

N*num*	draws north (up) *num* units.
S*num*	draws south (down) *num* units.
E*num*	draws east (right) *num* units.
W*num*	draws west (left) *num* units.
NE*num*	draws northeast (up and right) *num* units.
NW*num*	draws northwest (up and left) *num* units.
SE*num*	draws southeast (down and right) *num* units.
SW*num*	draws southwest (down and left) *num* units.
A*val*	rotates the draw axis. Possible values are: 0 = normal 1 = 90 degrees 2 = 180 degrees 3 = 270 degrees
U*xcor,ycor*	draws a relative vector to the specified coordinates. *Xcor* and *ycor* are relative to the current draw pointer position. The draw pointer location does not change. *Xcor* and *ycor* must be separated by a comma.

B*xcor,ycor* produces a blank line (moves the cursor but does not draw). The *xcor* and *ycor* coordinates are relative to the current draw pointer location. If you specify relative coordinates located offscreen, you cannot see subsequent lines.

Examples:

```
RUN GFX2("DRAW","N10,E10,S10,W10")
```

Sample Program:

```
PROCEDURE drawing
DIM T,X,Y,COLOR:INTEGER
COLOR=0
RUN GFX2("CLEAR")
FOR T=1 TO 96 STEP 6
RUN GFX2("SETDPTR",320,96)
FOR Y=0 TO 3
COLOR=MOD(Y,2)
RUN GFX2("COLOR",COLOR)
FOR X=1 TO 4
READ DR$
DR$="A"+STR$(Y)+DR$+STR$(T)
RUN GFX2("DRAW",DR$)
NEXT X
NEXT Y
RESTORE
NEXT T
RUN GFX2("COLOR",3)
END
DATA "N","E","S","W"
```

DWEND Device window end

Syntax: RUN GFX2([*path*,]"DWEND")

Function: Deallocates the device window you initialized with DWSET and INIZ. If the window deallocated is the last device window on the screen, BASIC09 returns the screen memory to the system. DWEND automatically positions you in the next device window, a result similar to pressing CLEAR. You can use this function with DWSET to redefine a device window to a different type.

Parameters:

path	The path number of the window you wish to end. Path can be a constant or variable.

Examples:

```
RUN GFX2("DWEND")
RUN GFX2(PATH,"DWEND")
RUN GFX2(3,"DWEND")
```

Sample Program:

From /TERM, this procedure temporarily opens a path to Window 3, displays the new window, draws a design, then returns to the /TERM screen and closes the path.

```
PROCEDURE decorate
□DIM PATH,T,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□RUN GFX2(PATH,"COLOR",3,2)
□FOR T=1 TO 185 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2(PATH,"COLOR",1,2)
□FOR T=185 TO 1 STEP -6
```

```
RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
IF INT(T/3)=T/3 THEN
Y=Y+1
ENDIF
NEXT T
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"DWEND")
CLOSE #PATH
END
```

DWPROTSW Device window protect switch

Syntax: **RUN GFX2([*path*,]"DWPROTSW","*switch*")**

Function: Lets you *unprotect* one device window and set other device windows on top of it.

OS-9 on the Color Computer 3 normally uses a protected windowing system that does not allow window devices to overlap. Removing the window protection with DWPROTSW lets one device window exist on the same screen area as another window device. Because this might destroy the contents of an unprotected window, you need to use care with this function.

Parameters:

path	The route to the window you want to unprotect.
switch	Either OFF to turn off protection, or ON to turn on protection. The default is ON.

Examples:

```
RUN GFX2("DWPROTSW",OFF)
```

DWSET Device window set

Syntax: **RUN GFX2([*path*,]"DWSET",*format,xcor,ycor, width,length,foreground,background,border*)**

Function: Defines a device window. Normally, you first open a path to a window, then use DWSET to set the window format, location, size, and colors.

Parameters:

path	The route to the window you are defining.
format	The code for the type of screen you want to establish. See Table 9.6 at the beginning of this section for the formats available.
xcor,ycor	The coordinates (character column and row) of the upper left corner of the screen you want to create.
width	The width (in characters) of the new window.
length	The depth (in lines) of the new window.
foreground	The code for the window's foreground color.
background	The code for the window's background color.
border	The code for the window's border color.

Examples:

```
RUN GFX2("DWSET",06,50,100,50,10,20,12,9)
```

Sample Program:

This procedure opens a path to Window 3, uses DWSET to define the new window, displays the new window, and draws a graphic *lemon* shape. It then uses SELECT to return to the /TERM window or screen, deallocates Window 3, and closes the path.

```
PROCEDURE lemon
□DIM PATH,T,X,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□RUN GFX2(PATH,"COLOR",0,2)
□FOR T=1 TO 185 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□X=T
□RUN GFX2(PATH,"COLOR",3,2)
□FOR T=62 TO 1 STEP -3
□RUN GFX2(PATH,"ELLIPSE",320,96,X,T)
□IF INT(T/3)=T/3 THEN
□X=X+1
□ENDIF
□NEXT T
□RUN GFX2(1,"SELECT")
□RUN GFX2(PATH,"DWEND")
□CLOSE #PATH
□END
```

ELLIPSE Draw an ellipse

Syntax: **RUN GFX2([*path*,]"ELLIPSE"[,*xcor*,*ycor*], *xrad*,*yrad*)**

Function: Draws an ellipse with the center at the current draw pointer position or at the specified X,Y coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path	The route to the window in which you want to draw.
xcor,*ycor*	The coordinates for the ellipse's center. If you omit these coordinates, ELLIPSE uses the current draw pointer position.
xrad,*yrad*	The radii of the ellipse's length and height.

Examples:

```
RUN GFX2("ELLIPSE",100,50)

RUN GFX2("ELLIPSE",100,125,100,10)
```

Sample Program:

This program uses ELLIPSE to draw a graphic design shaped like a Christmas tree decoration.

```
PROCEDURE xbulb
□DIM T,Y:INTEGER
□Y=1
□RUN GFX2("COLOR",3,2)
□RUN GFX2("CLEAR")
□FOR T=1 TO 180 STEP 3
□Y=Y+1
□RUN GFX2("ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2("COLOR",1,2)
□FOR T=180 TO 1 STEP -6
```

```
□RUN GFX2("ELLIPSE",320,96,T,Y)
□IF INT(T/3)=T/3 THEN
□Y=Y+1
□ENDIF
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

EREOLINE Erase to end of line

Syntax: **RUN GFX2([*path*,]"EREOLINE")**

Function: Deletes the portion of the current line from the cursor to the right side of the window.

Parameters:

path	The route to the window in which you want to erase a portion of a line.

Examples:

```
RUN GFX2("EREOLINE")
```

Sample Program:

This procedure uses EREOLINE to produce a series of steps down the screen.

```
PROCEDURE steps
□DIM T,J,K:INTEGER
□RUN GFX2("COLOR",2,3)
□RUN GFX2("CLEAR")
□RUN GFX2("COLOR",3,2)
□FOR T=0 TO 22
□J=T*3
□RUN GFX2("CURXY",J,T)
□RUN GFX2("EREOLINE")
□NEXT T
```

EREOWNDW Erase to end of window

Syntax: **RUN GFX2([*path*,]"EREOWNDW")**

Function: Deletes all the lines in a window from the line on which the cursor is positioned to the bottom of the window.

Parameters:

path	The route to the window in which you want to delete screen contents.

Examples:

```
RUN GFX2("EREOWNDW")
```

ERLINE Delete current line of text

Syntax: **RUN GFX2([*path*,]"ERLINE")**

Function: Deletes the current line (on which the cursor is resting) from the window but does not close the space.

Parameters:

path	The route to the window in which you want to remove the contents of a screen line.

Examples:

```
RUN GFX2("ERLINE")
```

Sample Program:

This procedure draws a bull's-eye design, then slices it with the ERLINE function.

```
PROCEDURE cut
□DIM X,Y,R,T,COLOR:INTEGER
□COLOR=0
□X=320
□Y=96
□RUN GFX2("CLEAR")
□COLOR=0
□FOR T=185 TO 10 STEP -10
□RUN GFX2("CIRCLE",X,Y,T)
□NEXT T
□FOR T=140 TO 320 STEP 10
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",T,96)
□COLOR=COLOR+1
□NEXT T
□FOR T=2 TO 22 STEP 2
□RUN GFX2("CURXY",0,T)
□RUN GFX2("ERLINE")
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

FILL Fill (paint) window

Syntax: **RUN GFX2([*path*,]"FILL",[*xcor,ycor*])**

Function: Paints an area with the current foreground color. Paint fills the portion of the window that is the same color as the pixel under the draw pointer.

Parameters:

path	The route to the window in which you want to use the FILL function.
xcor,ycor	Are optional X- and Y-coordinates to reposition the draw pointer before FILL begins. If you omit these coordinates, BASIC09 uses the current draw position.

Examples:

```
RUN GFX2("FILL",100,100)
```

Sample Program:

This procedure draws and fills 100 boxes on a window.

```
PROCEDURE colorbox
□DIM A,B,C,D,T,COLOR:INTEGER
□COLOR=0
□RUN GFX2("CLEAR")
□FOR T=1 TO 100
□A=RND(560)
□B=RND(151)
□C=A+RND(80)
□D=B+RND(40)
□COLOR=COLOR+1
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BOX",A,B,C,D)
□RUN GFX2("FILL",A+1,B+1)
□NEXT T
```

FONT Define font buffer

Syntax: **RUN GFX2([*path*,]"FONT",*group*,*buffer*)**

Function: Defines a buffer from which BASIC09 gets the character font (style) for the current screen. Use the text/cursor handling functions referenced in this section with the font you load. When you merge the Stdfonts file in your SYS directory with a graphics window, you have the choice of three fonts from Buffers 1, 2, and 3, located in Group 200. You can also create your own fonts. FONT works only on graphics screen. See "Using Fonts" earlier in this chapter.

You must load the font you want to use into the defined buffer before using FONT.

Parameters:

path	The route to the window in which you want to use an alternate font.
group	The group number of the buffer containing the font to use.
buffer	The number of the buffer containing the font to use.

Examples:

```
RUN GFX2("FONT",200,2)
```

GCSET Set graphics cursor

Syntax: **RUN GFX2("GCSET",*group,buffer*)**

Function: Defines a buffer from which BASIC09 gets the graphics cursor. This lets you define your own cursor for graphics operations. To turn the graphics cursor off, use a group Number 0. You must execute this command to display a graphics cursor. Before using GCSET, you must merge the Stdcur file in the SYS directory to the window.

Parameters:

group	The group number of the buffer containing the cursor image to use. See *OS-9 Windowing System* for information on the group to use.
buffer	The number of the buffer that contains the cursor image to use. See *OS-9 Windowing System* for information on the buffer to use.

Examples:

```
RUN GFX2("GCSET",1,5)
```

GET Get a block from the window

Syntax: RUN GFX2([*path*,]"GET",*group,buffer,xcor, ycor,xsize,ysize*)

Function: Saves a window area Get/Put buffer. Use PUT to replace the image to the window. If you did not previously define the buffer, BASIC09 creates it. If you store the window data in a predefined buffer, the data must be the same size or smaller than the buffer. If not, BASIC09 truncates the data to the size of the buffer. (Also see PUT and DEFBUFF.)

Parameters:

path	The route to the window where you want to save an image.
group	The group number of the Get buffer (1-199).
buffer	The Get buffer number (1-255).
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window image to save. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.
xsize	The horizontal size of the window section to save.
ysize	The vertical size of the window section to save.

Examples:

```
RUN GFX2("GET",1,5,0,0,10,15)
```

Sample Program:

This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation.

```
PROCEDURE puttdown
□DIM T,J:INTEGER
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,90)
□RUN GFX2("BAR",305,100,335,104)
□RUN GFX2("GET",1,1,288,85,50,23)
□RUN GFX2("GET",1,2,1,1,50,23)
□RUN GFX2("PUT",1,2,288,85)
□J=10
□FOR T=20 TO 559 STEP 6
□J=J+2
□RUN GFX2("PUT",1,1,T,J)
□NEXT T
□RUN GFX2("KILLBUFF",1,1)
□RUN GFX2("CURON")
□END
```

GPLOAD Load data into Get/Put buffer

Syntax: **RUN GFX2("GPLOAD",*group,buffer,format, xdim,ydim,size*)**

Function: Loads a buffer with image data that PUTBLK can use for window displays. If the Get/Put buffer is not created, BASIC09 creates it. If it is defined, the size of the data should not be larger than the buffer.

Parameters:

group	The group number you select, in the range 1-199, to let you group buffers.
buffer	A number in the range 1-255 that you assign to the buffer you create.
format	The type code of the screen format. (See Table 9.4.)
xdim	The X (horizontal) dimension of the stored block.
ydim	The Y (vertical) dimension of the stored block.
size	The size of the buffer in bytes. A buffer size can be in the range of 1 to 8 kilobytes, depending on available memory.

Examples:

```
RUN GFX2("DEFBUFF",1,5,06,100,50,5000)
```

INSLIN Insert line

Syntax: **RUN GFX2([*path*,]"INSLIN")**

Function: Moves the window lines at and below the cursor down one line.

Parameters:

path	The route to the window in which you want a blank line.

Examples:

```
RUN GFX2("INSLIN")
```

Sample Program:

This procedure draws a round face on the screen, then uses INSLIN and DELLIN to make a mouth appear to move.

```
PROCEDURE chomp
□DIM X,Y,T:INTEGER
□DIM RESPONSE:STRING[1]
□RESPONSE=""
□RUN GFX2("CLEAR")
□RUN GFX2("CIRCLE",320,96,80)
□RUN GFX2("COLOR",0,2)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",2)
□RUN GFX2("CIRCLE",285,80,12)
□RUN GFX2("CIRCLE",355,80,12)
□RUN GFX2("FILL",285,80)
□RUN GFX2("FILL",355,80)
□RUN GFX2("CIRCLE",315,96,3)
□RUN GFX2("CIRCLE",325,96,3)
□RUN GFX2("ARC",320,92,14,3,3,1,1,1)
□RUN GFX2("COLOR",3,2)
□RUN GFX2("CIRCLE",289,77,3)
□RUN GFX2("CIRCLE",359,77,3)
□RUN GFX2("CURXY",0,14)
□REPEAT
```

```
RUN GFX2("INSLIN")
FOR X=1 TO 100
NEXT X
RUN GFX2("DELLIN")
RUN INKEY(RESPONSE)
UNTIL RESPONSE>""
END
```

KILLBUFF Deallocate Get/Put buffer

Syntax: RUN GFX2("KILLBUFF",*group,buffer*)

Deallocates the indicated Get/Put buffer. You select group and buffer numbers when you define a buffer or when you load or get a window image. For more information on Get/Put buffers, see DEFBUFF, PUT, GET, and GPLOAD.

Parameters:

group	The group number of the buffer you want to deallocate, in the range 1-199. Buffer Group Numbers 0 and 200-255 are reserved for OS-9 system use.
buffer	The number of the buffer to deallocate, in the range 1-255.

Examples:

```
RUN GFX2("KILLBUFF",1,5)
```

Sample Program:

This procedure draws a figure on a window screen, loads it into a buffer, then repeatedly places it in new locations on the screen. Each new PUT erases the previous image.

```
PROCEDURE putdown
□DIM X,Y,T,J:INTEGER
□RUN GFX2("CUROFF")
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,90)
□RUN GFX2("BAR",305,100,335,104)
□RUN GFX2("GET",1,1,288,85,50,23)
□RUN GFX2("GET",1,2,1,1,50,23)
□RUN GFX2("PUT",1,2,288,85)
```

```
J=10
FOR T=20 TO 559 STEP 6
J=J+2
RUN GFX2("PUT",1,1,T,J)
NEXT T
RUN GFX2("KILLBUFF",1,1)
RUN GFX2("CURON")
END
```

LINE Draw a line

Syntax: RUN GFX2([*path*,]"LINE"[,*xcor1,ycor1*],*xcor2, ycor2*)

Function: Draws a line in one of the following ways:

- From the current draw pointer to the specified X- and Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X- and Y-coordinates.

Parameters:

path	The route to the window in which you want to draw a line.
xcor1,ycor1	The optional beginning X- and Y-coordinates for the line.
xcor2,ycor2	The ending X- and Y-coordinates for the line.

Examples:

```
RUN GFX2("LINE",192,128)

RUN GFX2("LINE",0,0,192,128)
```

Sample Program:

This procedure draws a sine wave of vertical lines across a window.

```
PROCEDURE waves
□DIM A,X,Y,Z:INTEGER
□CALC=0
□A=100
□RUN GFX2("CLEAR")
□RUN GFX2("COLOR",3,2)
□FOR X=0 TO 638 STEP 1
□CALC=CALC+.05
□Y=A-SIN(CALC)*15
□Z=Y+25
```

```
RUN GFX2("LINE",X,Y,X,Z)
NEXT X
END
```

LOGIC Perform logic function

Syntax: **RUN GFX2("LOGIC","*function*")**

Function: Causes BASIC09 to perform the specified logic function on all data bits used by subsequent drawing functions. Once set, the logic function remains in effect until you turn LOGIC off.

Parameters:

function can be one of the following logical functions:

OFF	— no logic
AND	— performs AND logic
OR	— performs OR logic
XOR	— performs XOR logic

Examples:

```
RUN GFX2("LOGIC","AND")

RUN GFX2("LOGIC","XOR")
```

Sample Program:

This procedure uses LOGIC to draw a horizontal bar across a background of multicolored vertical bars. Using XOR logic, the procedure causes the horizontal bar to change the color of each vertical bar.

```
PROCEDURE logic
□DIM A,Z,T,X,Y,COLOR:INTEGER
□RUN GFX2("LOGIC","OFF")
□RUN GFX2("CLEAR")
□COLOR=0
□FOR T=0 TO 619 STEP 20
□COLOR=COLOR+1
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",T,0,T+20,190)
□NEXT T
□RUN GFX2("COLOR",3,2)
□RUN GFX2("LOGIC","XOR")
```

```
FOR T=1 TO 10
RUN GFX2("BAR",0,80,639,112)
NEXT T
RUN GFX2("LOGIC","OFF")
END
```

OWSET Establish an overlay window

Syntax: **RUN GFX2([*path*,]"OWSET",*save switch,xpos, ypos,xsize,ysize,foreground,background*)**

Function: Creates an overlay window on a previously existing device window. Reconfigures the current device window paths to use a new area of the screen as the current device window.

Parameters:

path	The route to the window in which you want to set an overlay.
save switch	Either 0 or 1. A value of 0 tells BASIC09 not to save the overlaid area. A value of 1 tells BASIC09 to save the overlaid area and restore it when the new window closes.
xpos	The character column in which to start the new window (upper left corner).
ypos	The character row in which to start the new window (upper left corner).
xsize	The width of the new window in characters.
ysize	The depth of the new window in rows.
foreground	The foreground color of the new window.
background	The background color of the new window.

Examples:

```
RUN GFX2("OWSET",00,44,10,32,8,00,06)
```

Sample Program:

This procedure creates six progressively smaller overlay windows, labeling each. It then waits for you to press a key, after which it erases all the windows and leaves the original window intact.

```
PROCEDURE overwin
□DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
□DIM RESPONSE:STRING[1]
□X=0 \Y=0
□X1=80 \Y1=24
□PLACE=33
□FOR T=1 TO 6
□IF T=2 OR T=6 THEN
□B=3
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□X=X+6 \Y=Y+2
□X1=X1-12 \Y1=Y1-4
□FOR J=1 TO 5
□PRINT TAB(PLACE); "Overlay Screen "; T
□NEXT J
□PLACE=PLACE-6
□NEXT T
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 6
□RUN GFX2("OWEND")
□NEXT T
□END
```

PALETTE Set color for palette registers

Syntax: **RUN GFX2([*path*,]"PALETTE",*register*,*color*)**

Function: Sets palette colors. Lets you *install* any of the Color Computer's 64 colors in the palette for use with text and graphics.

Parameters:

path	The route to the window where you want to change palette colors.
register	The number of the register in which you want to install a new color.
color	The code of the new color you want to install.

Examples:

```
RUN GFX2("PALETTE",13,32)
```

Sample Program:

This procedure draws a series of bars and circles, then repeatedly changes their colors using PALETTE.

```
PROCEDURE palette
□DIM T,K,J,X,Y,COLOR:INTEGER
□DIM RESPONSE:STRING[1]
□RUN GFX2("COLOR",3,2,2)
□COLOR=0
□RUN GFX2("CLEAR")
□RUN GFX2("CUROFF")
□FOR Y=0 TO 23 STEP 3
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",0,Y,639,Y+3)
□COLOR=COLOR+1
□IF COLOR=2 THEN
□COLOR=COLOR+1
□ENDIF
□NEXT Y
□FOR Y=164 TO 185 STEP 3
```

```
RUN GFX2("COLOR",COLOR)
RUN GFX2("BAR",0,Y,639,Y+3)
COLOR=COLOR+1
NEXT Y
COLOR=0
FOR K=45 TO 170 STEP 48
FOR T=100 TO 580 STEP 100
RUN GFX2("COLOR",3)
RUN GFX2("CIRCLE",T,K,30)
RUN GFX2("COLOR",COLOR)
RUN GFX2("FILL",T,K)
COLOR=COLOR+1
IF COLOR=2 THEN
COLOR=COLOR+1
ENDIF
NEXT T
NEXT K
REPEAT
X=RND(63)
REPEAT
Y=RND(16)+1
UNTIL Y<>2
RUN GFX2("PALETTE",Y,X)
RUN INKEY(RESPONSE)
UNTIL RESPONSE>""
RUN GFX2("DEFCOL")
RUN GFX2("CURON")
END
```

PATTERN Select pattern buffer

Syntax: **RUN GFX2([*path*,]"PATTERN",*group,buffer*)**

Function: Selects the contents of a preloaded Get/Put buffer as a pattern for graphics functions. Although PATTERN can use a buffer of any size, it uses a specific number of bytes, depending on the screen format in use:

Color Mode	Pattern Array Size	Bits Per Pel
02	4 bytes x 8 bytes = 32 bytes	1
04	8 bytes x 8 bytes = 64 bytes	2
16	16 bytes x 8 bytes = 128 bytes	4

The pattern array is a 32 x 8 pel representation of graphics memory. It takes the current color mode into consideration to define the number of bits per pel and pels per byte. If the buffer is larger than the number of bytes required, PATTERN ignores the extra bytes. BASIC09 uses the selected pattern with all draw commands until you change the pattern or turn off the pattern function by specifying a group and buffer number of 0.

Parameters:

path The route to the window in which you want to use a new graphics pattern.

group The group number of the buffer you want to use for a graphics pattern.

buffer The buffer number that you want to use for a graphics pattern.

Examples:

```
RUN GFX2("PATTERN",1,3)
```

Sample Program:

This procedure loads the current window data at location 0,0 into a buffer to use as a draw pattern. It then draws a circle and fills the circle with the pattern in the buffer.

```
PROCEDURE pattern
□DIM X,Y,T:INTEGER
□RUN GFX2("GET",1,1,0,0,5,5)
□RUN GFX2("COLOR",4)
□RUN GFX2("CLEAR")
□RUN GFX2("CIRCLE",320,96,100)
□RUN GFX2("FILL",320,96)
□RUN GFX2("PATTERN",1,1)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,96)
□RUN GFX2("PATTERN",0,0)
□END
```

POINT Mark a point

Syntax: **RUN GFX2([*path*,]"POINT"[,*xcor,ycor*])**

Function: Sets the pixel at the current draw pointer position or at the specified coordinates to the current foreground color. If you do not specify coordinates, POINT sets the pixel at the draw pointer.

Parameters:

path	The route to the window in which you want to turn on the specified pixels.
xcor,ycor	Optional coordinates for the POINT function. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("POINT")

RUN GFX2("POINT",192,128)
```

Sample Program:

This procedure uses POINT to produce a *swirl* design on a window screen.

```
PROCEDURE point
□BASE 0
□DIM X(20),Y(20):INTEGER
□DIM T,R,J,K:INTEGER
□J=0
□K=0
□RUN GFX2("CUROFF")
□RUN GFX2("CLEAR")
□FOR T=1 TO 288 STEP 3
□J=J+1
□FOR R=0 TO 11
□X(R)=INT(T*SIN(30*R+K))+320
□Y(R)=INT(J*COS(30*R+K))+96
□RUN GFX2("POINT",X(R),Y(R))
```

```
□K=K+1
□NEXT R
□NEXT T
□RUN GFX2("CURON")
□END
```

PROPSW Proportional space switch

Syntax: **RUN GFX2([*path*,]"PROPSW","*switch*")**

Function: Enables or disables the automatic proportional spacing of characters on graphic screens.

Parameters:

path	The route to the window in which you want to use proportional character spacing.
switch	Either OFF to turn proportional spacing off, or ON to turn proportional spacing on. The default setting of the switch is OFF.

Examples:

```
RUN GFX2("PROPSW","ON")
```

Sample Program:

This procedure produces a demonstration of the BASIC09 proportional spacing function.

```
PROCEDURE proport
□DIM LINE:STRING
□DIM LETTER:STRING[1]
□DIM T,J,K,FLAG:INTEGER
□RUN GFX2("CLEAR")
□FLAG=1
□FOR T=1 TO 12
□READ LINE
□FOR J=1 TO LEN(LINE)
□LETTER=MID$(LINE,J,1)
□IF LETTER<>"!" AND LETTER<>"#" THEN
□PRINT LETTER;
□ENDIF
□IF LETTER="!" THEN
□FLAG=FLAG*-1
□IF FLAG>0 THEN
□RUN GFX2("PROPSW","OFF")
```

```
□ELSE
□RUN GFX2("PROPSW","ON")
□ENDIF
□ENDIF
□IF LETTER="#" THEN
□PRINT CHR$(34);
□ENDIF
□NEXT J
□PRINT
□NEXT T
□PRINT \ PRINT
□END
□DATA "This is a demonstration of"
□DATA "!Proportional Spacing! using"
□DATA "BASIC09's GFX2 module."
□DATA ""
□DATA "!The quick brown fox jumped...!"
□DATA "The quick brown fox jumped..."
□DATA ""
□DATA "Use the command"
□DATA "!RUN GFX2(#PROPSW#,#ON#)!"
□DATA "to turn proportional spacing on."
□DATA "Use !RUN GFX2(#PROPSW#,#OFF#)!"
□DATA "to turn proportional spacing off"
```

PUT Put a saved data block on the window

Syntax: RUN GFX2([*path,*]"PUT",*group,buffer, xcor,ycor*)

Function: Places the image in the specified Get/Put buffer on the window. PUT requires only the group and buffer numbers and the window coordinates for the upper left corner of the image. The GET function saves the dimensions of the block in the buffer. PUT automatically handles window format conversion.

Parameters:

path	The route to the window where you want to place a pre-saved image.
group	The group number of the buffer in which to save the window data.
buffer	The buffer number in which to save the window data.
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window position. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("PUT",1,5,100,50)
```

Sample Program:

This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation.

```
PROCEDURE putdown
□DIM X,Y,T,J:INTEGER
□RUN GFX2("CUROFF")
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,90)
□RUN GFX2("BAR",305,100,335,104)
□RUN GFX2("GET",1,1,288,85,50,23)
□RUN GFX2("GET",1,2,1,1,50,23)
□RUN GFX2("PUT",1,2,288,85)
□J=10
□FOR T=20 TO 559 STEP 6
□J=J+2
□RUN GFX2("PUT",1,1,T,J)
□NEXT T
□RUN GFX2("KILLBUFF",1,1)
□RUN GFX2("CURON")
□END
```

PUTGC Put graphics cursor

Syntax: **RUN GFX2([*path*,]"PUTGC",*xcor,ycor*)**

Function: Places and displays the graphics cursor at the specified location. Use screen relative coordinates for this function, not window relative coordinates. The horizontal range is 0-639. The vertical range is 0-191.

Parameters:

path	The route to the window where you want to display a graphics cursor.
xcor,ycor	The screen coordinates for the cursor location. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Examples:

```
RUN GFX2("PUTGC",100,5)
```

Sample Program:

This procedure displays the available graphic cursors stored in group 202. Before this procedure can work, you must merge the Stdptrs file in the SYS directory of your system disk with the window you are using. For instance, if your system diskette is in Drive /D0, merge Stdptrs with Window 1, by typing:

```
merge /d0/sys/stdptrs > /w1 ENTER
```

```
PROCEDURE viewcur
□DIM T,Z:INTEGER
□RUN GFX2("CLEAR")
□FOR T=1 TO 7
□RUN GFX2("GCSET",202,T)
□RUN GFX2("PUTGC",320,96)
□FOR Z=1 TO 6000
□NEXT Z
□NEXT T
□RUN GFX2("GCSET",0,0)
□END
```

REVON Reverse video on
REVOFF Reverse video off

Syntax: **RUN GFX2([*path*,]"REVON")**
RUN GFX2([*path*,]"REVOFF")

Function: Enables or disables reverse video characters. Once set, reverse video remains in effect until you execute the reverse video off function.

Parameters:

path	The route to the window in which you want to display reverse characters.

Examples:

```
RUN GFX2("REVON")
RUN GFX2("REVOFF")
```

SCALESW Enable/disable scaling

Syntax: RUN GFX2([*path*,]"SCALESW","*switch*")

Function: Enables or disables scaling when drawing on variously formatted windows. Scaling in windows is normally on. If scaling is off, coordinates are relative to the window origin coordinates. Scaling does not affect text.

Parameters:

path	The route to the window where you want to turn scaling off or on.
switch	Either OFF (disable scaling) or ON (enable scaling).

Examples:

```
RUN GFX2("SCALESW","OFF")
```

Sample Program:

This procedure runs a routine of drawing a design in overlay windows twice. The routine runs once with scaling off and once with scaling on. After the first routine pauses, press the space bar to see the second demonstration.

```
PROCEDURE scale
□DIM X,Y,X1,Y1,T,B,J,R,W,Z:INTEGER
□DIM RESPONSE:STRING[1]
□RUN GFX2("CLEAR")
□FOR J=1 TO 2
□IF J=1 THEN
□RUN GFX2("SCALESW","OFF")
□ELSE
□RUN GFX2("SCALESW","ON")
□ENDIF
□X=0 \Y=0 \X1=80 \Y1=24
□FOR T=1 TO 4
□IF T=2 OR T=6 THEN
□B=3
```

```
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□FOR R=1 TO 35
□W=40*SIN(R)+170
□Z=25*COS(R)+45
□RUN GFX2("CIRCLE",W,Z,30)
□NEXT R
□X=X+6 \Y=Y+2 \X1=X1-12 \Y1=Y1-4
□NEXT T
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 4
□RUN GFX2("OWEND")
□NEXT T
□NEXT J
□END
```

SELECT Select next window

Syntax: **RUN GFX2([*path*],"SELECT")**

Function: SELECT causes a window to display if the procedure is operating in the active window. If the procedure is not in the active window, the newly selected window displays when you press CLEAR. If you do not specify a path, BASIC09 selects the device using the standard input, standard output, and standard error paths, Paths 0, 1, and 2.

Parameters:

path	The path to the window to select.

Examples:

```
RUN GFX2("SELECT")

RUN GFX2(1,"SELECT")

RUN GFX2(PATH,"SELECT")
```

Sample Program:

From /TERM, this procedure temporarily opens a path to Window 3, creates the window format, and uses SELECT to display the new window. It draws a design, then returns to the /TERM screen and closes the path.

```
PROCEDURE design
□DIM PATH,T,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",5,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□FOR T=1 TO 200 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2(PATH,"COLOR",1,2)
□FOR T=200 TO 1 STEP -6
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
```

```
IF INT(T/3)=T/3 THEN
Y=Y+1
ENDIF
NEXT T
RUN GFX2(1,"SELECT")
RUN GFX2(PATH,"DWEND")
CLOSE #PATH
END
```

SETDPTR Set draw pointer

Syntax: **RUN GFX2([*path*,]"SETDPTR",*xcor,ycor*)**

Function: Places the draw pointer at the specified coordinates. The draw pointer selects the beginning point of the next graphics draw function (such as CIRCLE, LINE, BOX, and so on), if you do not supply other coordinates.

Parameters:

path	The route to the screen where you want to set the draw pointer.
xcor,ycor	The screen coordinates for the draw pointer location. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("SETDPTR",100,5)
```

Sample Program:

This procedure uses coordinates from a DATA statement for setting the draw pointer to create a series of star shapes.

```
PROCEDURE star
□DIM X,Y,T,J:INTEGER
□PRINT CHR$(12)
□FOR J=1 TO 10
□READ X,Y
□RUN GFX2("SETDPTR",X+J,Y+J+J)
□FOR T=1 TO 5
□READ X,Y
□RUN GFX2("LINE",X+J,Y+J+J)
□NEXT T
□NEXT J
□DATA 320,46,440,146,200,84,440,84,200,146,320,46
□END
```

UNDLNON Underline characters on
UNDLNOFF Underline characters off

Syntax: **RUN GFX2([*path*,]"UNDLNON")**
RUN GFX2([*path*,]"UNDLNOFF")

Function: Enables or disables character underline. After you execute UNDLNON, all characters displayed are underlined until you execute UNDLNOFF. The default is UNDLNOFF.

Parameters:

path The route to the window where you want to use underline characters.

Examples:

```
RUN GFX2("UNDLNON")
RUN GFX2("UNDLNOFF")
```

BASIC09 Quick Reference

This chapter contains a quick reference of all BASIC09 commands, statements, and functions. It includes commands for programming, editing, and debugging, as well as the Commands mode commands.

The following chart lists all BASIC09 keywords that you can use in a procedure.

Statements and Functions

Command	Description
ABS	Returns the absolute value of a number.
ACS	Calculates the arccosine of a number.
ADDR	Returns an integer value which is the absolute memory address of a variable, array, or structure in a process's address space.
AND	Generates the logical AND of two Boolean values.
ASC	Returns the ASCII code of the first character in a string.
ASN	Calculates the arcsine of a number.
ATN	Calculates the arctangent of a number.
BASE	Sets the lowest array or data structure subscript in a procedure to either 0 or 1.
BYE	Ends execution of a procedure and terminates BASIC09.
CHAIN	Executes a module, passing arguments if appropriate.
CHD	Changes the current data directory.
CHR$	Returns the ASCII character represented by a specified integer.
CHX	Changes the current execution directory.
CLOSE	Deallocates the specified path to a file or device.
COS	Calculates the cosine of a number.

Command	Description
CREATE	Opens a path and establishes a new file on disk.
DATE$	Returns the computer's current date and time.
DEG	Causes BASIC09 to calculate angles in degrees.
DATA	Stores data in a procedure to be accessed by the READ statement.
DELETE	Deletes a file from disk.
DIM	Declares simple variables, arrays or complex data structure for size and type.
DO	See WHILE/DO/ENDWHILE.
ELSE	See IF/THEN/ELSE/ENDIF.
END	Terminates execution of a procedure. Returns to the calling procedure or to BASIC09's command mode. Displays the specified text.
ENDEXIT	See EXITIF/ENDEXIT.
ENDIF	See IF/THEN/ELSE/ENDIF.
ENDLOOP	See LOOP/ENDLOOP.
ENDWHILE	See WHILE/DO/ENDWHILE.
EOF	Tests for the end of a disk file.
ERR	Returns the error code of the most recent error.
ERROR	Generates the specified error.
EXITIF/ ENDEXIT	Tests conditions in a loop. The procedure exits the loop if the condition is true.
EXP	Calculates e (2.71828183) raised to the specified value.
FALSE	A Boolean function that always returns FALSE.
FIX	Rounds a real number and converts it to an integer.
FLOAT	Converts a byte or integer value to a real number.

Command	Description
FOR/NEXT	Creates a program loop of a specified number of repetitions.
GET	Reads an element or a data structure from a binary file or a device.
GOSUB/ RETURN	Transfers program control to a specified subroutine. RETURN sends execution back to the calling routine.
IF/THEN/ELSE/ ENDIF	Evaluates an expression and performs an operation if the conditions are met. Including ELSE causes an alternate operation if the conditions are false.
INKEY	Stores the character of a keypress in a string variable.
INPUT	Causes a procedure to accept input from the keyboard or other specified device.
INT	Returns the largest whole number less than or equal to the specified value.
KILL	*Unlinks* a procedure. (Removes it from BASIC09's directory.)
LAND	Performs a bit-by-bit logical AND on two-byte, or integer, values.
LEFT$	Returns the specified number of characters, from the leftmost portion of a string.
LEN	Returns the length of the specified string.
LET	Assigns a value to a variable.
LNOT	Performs a bit-by-bit logical NOT function on two-byte, or integer, values.
LOG	Calculates the natural logarithm.
LOG10	Calculates a base 10 logarithm.
LOOP/ ENDLOOP	Establishes a loop. Use EXITIF and ENDEXIT to test the loop and exit when a specified condition is true.
LOR	Performs a bit-by-bit logical OR on two-byte, or integer, values.

Command	Description
LXOR	Performs a bit-by-bit logical EXCLUSIVE OR on two-byte, or integer, values.
MID$	Returns the specified number of characters, beginning at the specified position in a string.
MOD	Returns the modulus (remainder) of a division operation.
NEXT	See FOR/NEXT.
NOT	Returns the logical complement of a Boolean value.
ON ERROR/ GOTO	Traps errors and transfers control to the specified line number.
ON/GOSUB	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execution transfers to the selected line.
ON/GOTO	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execute jumps to the selected line.
OPEN	Opens an I/O path to an existing file or device.
OR	Performs a logical OR on two Boolean values.
PARAM	Describes the parameters a called procedure expects from a calling procedure.
PAUSE	Suspends execution of a procedure, and enters the Debug mode.
PEEK	Returns the byte value of a memory address.
PI	Represents the constant 3.14159265.
POKE	Stores a byte value at a specified memory address.
POS	Returns the current character position of the print buffer.

Command	Description
PRINT	Sends the specified characters or values to the display.
PRINT USING	Sends characters or values to the display, using the specified format.
PRINT#	Sends the specified characters or values to the specified path.
PRINT# USING	Sends characters or values to the specified path using the specified format.
PUT	Writes data to a random access file.
RAD	Causes BASIC09 to calculate angles in radians.
READ	Accesses data from procedure DATA lines or from files or devices.
REM	Indicates that the following characters in a procedure line are comments and are not to be executed. Also use (* *), or (*.
REPEAT/UNTIL	Establishes a loop that executes until the specified condition is met.
RESTORE	Restores the DATA pointer to the first data item or to a specified line.
RETURN	See GOSUB/RETURN.
RIGHT$	Returns the number of characters specified, from the rightmost portion of a string.
RND	Returns a random number from a specified range.
RUN	Calls another procedure for execution.
SEEK	Changes the file pointer address.
SGN	Determines the sign of a number.
SHELL	Calls an OS-9 command or program for execution.
SIN	Calculates the sine of a specified value.
SIZE	Returns the number of bytes assigned to a variable, array, or complex data structure.
SQ	Calculates a value raised to the power of two.

Command	Description
SQR/SQRT	Calculates the square root of a positive number.
STEP	Sets the size of increment in a FOR/NEXT loop.
STOP	Terminates the execution of all procedures and returns to the BASIC09 Command mode.
STR$	Converts numeric data to string data.
SUBSTRING	Returns the starting position of a sequence of characters in a string.
SYSCALL	Executes an OS-9 System Call.
TAB	Begins a print operation at the specified column.
TAN	Calculates the tangent of a value.
TRIM$	Strips trailing spaces from the specified string.
TRON/TROFF	Turn the trace mode on and off.
TRUE	Returns the Boolean value of TRUE.
TYPE	Defines a new data type.
UNTIL	See REPEAT/UNTIL.
USING	See PRINT USING.
VAL	Converts a string to an integer.
WHILE/DO/ ENDWHILE	Executes a loop as long as a specified condition is true.
WRITE	Writes data in ASCII format to a file or device.
XOR	Performs a logical EXCLUSIVE OR on two Boolean values.

Commands by Type

Statements

BASE 0	DIM	GOSUB	OPEN	RETURN
BASE 1	ELSE	GOTO	PARAM	RUN
BYE	END	IF/THEN	PAUSE	SEEK
CHAIN	ENDEXIT	INPUT	POKE	SHELL
CHD	ENDIF	KILL	PRINT	STOP
CHX	ENDLOOP	LET	PUT	TROFF
CLOSE	ENDWHILE	LOOP	RAD	TRON
CREATE	ERROR	NEXT	READ	TYPE
DATA	EXITIF/THEN	ON ERROR/GOTO	REM	UNTIL
DEG	FOR/TO/STEP	ON/GOSUB	REPEAT	WHILE/DO
DELETE	GET	ON/GOTO	RESTORE	WRITE

Transcendental Functions

ACS	COS	LOG10	SIN
ASN	EXP	PI	TAN
ATN	LOG		

Numeric Functions

ABS	LAND	MOD	SQ
FIX	LNOT	RND	SQR
FLOAT	LOR	SGN	SQRT
INT	LXOR		

String Functions

ASC	LEFT$	RIGHT$	TRIM$
CHR$	LEN	STR$	VAL
DATE$	MID$	SUB	STR
INKEY			

Miscellaneous Functions

ADDR	FALSE	SIZE	SYSCALL
EOF	PEEK	TAB	
ERR	POS	TRUE	

Data Types

The following list shows the BASIC09 data type you can specify when defining a variable.

Type	Function
BOOLEAN	Returns TRUE or FALSE
BYTE	Specifies that a numeric variable is to store single-byte values.
INTEGER	Specifies that a numeric variable is to store integer (two-byte) values.
REAL	Specifies that a numeric variable is to store real (five-byte) values.
STRING	Specifies that a variable is to store ASCII characters.

Types of Access for Files

You can use the following parameters with the CREATE and OPEN commands. Check the individual commands for information on which parameter to use with which command.

Parameter	Function
DIR	Lets BASIC09 access a directory-type file for reading. Do not use with UPDATE or WRITE.
EXEC	Lets BASIC09 access the current execution directory rather than the current data directory.
READ	Sets the file access mode for reading.
WRITE	Sets the file access mode for writing.
UPDATE	Sets the file access mode for both reading and writing.

Command Mode

The following chart lists the commands available from the BASIC09 Commands mode:

Command	Function
$	Calls the shell command interpreter to execute an OS-9 command.
BYE or CTRL BREAK	Returns you to the OS-9 system or to the program that called BASIC09.
CHD	Changes the current data directory.
CHX	Changes the current execution directory.
DIR	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or E	Enters the procedure editor/compiler mode.
KILL	Removes one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a file into the workspace.
MEM	Displays current workspace size or reserves a specified amount of memory for the workspace.
PACK	Performs a second compilation and stores the resulting file in the execution directory.
RENAME	Changes a procedure's name.
RUN	Causes a procedure to execute.
SAVE	Writes one or more procedures to disk.

Edit Commands

The following chart lists the commands available from the Edit mode:

Command	Function
ENTER	Moves the edit pointer to the next line.
+*num*	Moves the edit pointer forward a specified number of lines.
+*	Moves the edit pointer past the last line.
−*num*	Moves the edit pointer back a specified number of lines.
−*	Moves the edit pointer to the first line.
text	A space followed by text inserts an unnumbered line before the current line.
line	Typing a line number with or without text following it inserts the line into the procedure.
line ENTER	Moves the edit pointer to the line *line*.
c/*str1*/*str2*/	Changes the text *str1* to the text *str2*.
c*/*str1*/*str2*	Changes all occurrences of *str1* to *str2*.
d	Deletes the current line.
d*	Deletes all the lines in the procedure.
l	Lists the current line.
l*	Lists all the lines in the current procedure.
q	Terminates the edit session.
r	Renumbers lines from the first line number, in increments of 10.
r*	Renumbers all numbered lines in increments of 10. The first line number is 100.
r *line*	Renumbers lines from *line* in increments of 10.
r *line num*	Renumbers lines from *line*, in increments of *num*.
s/*str*	Searches for the first occurrences of *str*.
s*/*str*	Searches for all occurrences of *str*.

Debug Commands

The following table lists all the Debug commands and what they accomplish:

Command	Function
$*command*	Tells BASIC09 to execute the specified OS-9 command or program.
BREAK	Sets a breakpoint at the specified procedure.
CONT	Causes procedure execution to continue.
DEG/RAD	Selects either degrees or radians as the unit of angle measurement for trigonometric functions.
DIR	Displays the procedures in the workspace.
Q	Leaves the Debug mode for the System mode.
LET	Assigns a new value to a variable.
LIST	Displays a source listing of the suspended procedure.
PRINT*var*	Displays the value of the specified variable.
STATE	Lists the *nesting* order of all active procedures.
STEP*num*	Causes execution of the suspended procedure in specified increments.
TRON/TROFF	Turns the trace function on and off.

BASIC09 Command Reference

BASIC09 is made of keywords (functions and statements) that you use, with their parameters, to instruct the computer to perform certain operations.

This chapter is a complete reference for all of BASIC09's keywords.

Keyword Format

The reference to each keyword is organized in this manner:

- The keyword.
- The proper *syntax* (spelling and form) for using the keyword.
- A brief description of the keyword's purpose or effect.
- Descriptions of any parameters or arguments for the keyword.
- Notes about special features or requirements of the keyword, when appropriate.
- One or more examples for using the keyword.
- One or more sample procedures.

This format can vary slightly, depending on the complexity of each keyword. For instance, some keywords require parameters or arguments, and others do not. Some keywords are self-explanatory and do not require a sample procedure.

The Syntax Line

The second line in each command or keyword reference is the syntax line. This line uses keyword *constants* and keyword *variables* to show you how to construct a command line. Constants are words, numbers, or symbols that you type exactly as they appear. Variables are words that only represent the actual words, numbers, or symbols that you must supply for the command.

All variables are italic. When you see an italicized word, you know that you must supply some other word, name, symbol, or value in place of that word. If a word, symbol, or value is not italicized, type it exactly the way it appears in the syntax line.

The syntax line also uses symbols to help you understand how to construct a command line. These symbols are:

[ ] Words, names, value, or symbols contained between right and left brackets are optional. You can use them or not, depending on what you want to accomplish with the command.

... Ellipsis indicates that the last parameter can be repeated.

The following syntax line for DELETE requires only one parameter, the variable *pathname*.

```
DELETE "pathname"
```

Because *pathname* is italicized, you know that you must replace it with other text—in this case the pathlist to the file you want to delete. If you wanted to delete a file named Test from the ROOT directory of Drive /D1, this syntax line tells you that you must type:

```
delete "/d1/test"
```

Other syntax lines are more complex, such as the line for CREATE:

```
CREATE #path,"pathlist" [access mode]
[+access mode][+...]
```

This line tells you how to create a path to a file or device. Because the number symbol (#) is not italicized, you type it after the blank space following the keyword. However, *path*,*pathlist*, and *access mode* are all italicized. You must replace them with other names or values.

The *access mode* variable is contained within brackets. This tells you that it is optional. You can include an access mode, or not. If you don't, BASIC09 opens the path in the Update Mode.

The second *access mode* shows that the command allows two access mode parameters, preceded by a plus symbol. The ellipsis show that you can have even more *access mode* parameters.

Other syntax lines show that no parameters are required, such as:

```
DATE$
```

This command returns the current date. There is nothing it requires, and you can do nothing else with it.

Sample Programs

The sample programs in this chapter are complete. That is, you can type them, run them, and get a result. The procedures let you see the syntax and form of a command, as well as showing you how it might be used in a program.

Because the programs are executable, the manual shows unformatted listings (without relative address, indented control structures, and so on). This helps eliminate confusion for you when you type the program. You can type it exactly as it appears, exit the editor, and run the procedure.

ABS Return absolute value

Syntax: **ABS(*number*)**

Function: Computes the absolute value of *number*. A number's absolute value is its magnitude without regard to its sign. Absolute values are always positive or zero.

Parameters:

number	Any positive or negative number.

Examples:

```
PRINT ABS(-66)

X=ABS(Y)
```

Sample Program:

The following procedure asks you to type the temperature, and makes an appropriate comment. It uses ABS to get the absolute value of the temperature.

```
PROCEDURE temperature
□DIM TEMP:INTEGER
□INPUT "What's the temperature outside? (Degrees
F)...",TEMP
□IF TEMP<0 THEN
□PRINT "That's "; ABS(TEMP); " below
zero!□□□Brrrrrrr!"
□END
□ENDIF
□IF TEMP=0 THEN
□PRINT "Zero degrees? That's mighty cold!"
□END
□ENDIF
□PRINT TEMP; " degrees above zero? That's kind of
balmy..."
□END
```

ACS Return arccosine

Syntax: **ACS(*number*)**

Function: Calculates the arccosine of *number*. Use the DEG or RAD commands to tell BASIC09 if *number* is in degrees or radians. If you do not specify degrees or radians, the default is radians.

Parameters:

number	The number for which you want to compute the arccosine.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

The procedure calculates the arccosine of a value you type and expresses the result in degrees.

```
PROCEDURE arccosine
□DEG
□DIM NUM:REAL
□INPUT "Enter a number between -1 and 1",NUM
□PRINT "The arccosine of "; NUM; " is---";
ACS(NUM)
□END
```

ADDR Return the location of a variable

Syntax: **ADDR(*name*)**

Function: Returns the absolute location in a process's address space of the variable, array, or data structure assigned to *name*. The address returned is that of the first character in the variable. If the variable is numeric, one or more of the locations might contain zero.

For instance, if you use ADDR to obtain the address of an integer variable that contains the value 44, the first address location (byte) contains 0, and the second location contains 44.

Parameters:

name	The name of a string, a numeric variable, an array, or a data structure.

Examples:

This procedure displays the memory address where a variable named X resides:

```
PRINT ADDR(X)
```

Sample Program:

This procedure uses ADDR to tell you the memory location of the variable that stores your keyboard entry.

```
PROCEDURE address
□DIM A:INTEGER
□DIM TEST:STRING
□INPUT "Type a string of characters...",TEST
□A=ADDR(TEST)
□PRINT "The string you typed is stored at address
"; A
□PRINT "This is what it contains:..."
□FOR T=A TO A+LEN(TEST)
□PRINT CHR$(PEEK(T));
□NEXT T
□PRINT
□END
```

AND Performs a logical AND operation

Syntax: *operand1* AND *operand2*

Function: Performs the logical AND operation on two or more values, returning a value of either TRUE or FALSE.

Parameters:

operand1 *operand2*	Can be either numeric or string values.

Examples:

```
PRINT A>3 AND B>3

PRINT A$="YES" AND B$="YES"
```

Sample Program:

The following program calculates an insurance premium rate that is based on the answers to some lifestyle questions. Every time you press [Y], the premium rate goes up. The procedure uses AND to increase the rate by two percent if you both smoke and drink.

```
PROCEDURE policy
DIM POLICY_VALUE,RATE:REAL
DIM SMOKE,DRINK:STRING[1]
POLICY_VALUE=1000000.
RATE=.001
INPUT "Do you smoke? (Y/N)...",SMOKE
INPUT "Do you drink? (Y/N)...",DRINK
IF SMOKE="Y" AND DRINK="Y" THEN RATE=RATE+.02
ELSE
IF SMOKE="Y" THEN RATE=RATE+.01
ENDIF
IF DRINK="Y" THEN RATE=RATE+.01
ENDIF
ENDIF
PRINT "Your premium is "; RATE*POLICY_VALUE
END
```

ASC Returns ASCII code

Syntax: **ASC(*string*)**

Function: Returns the ASCII code for the first character of *string*.

ASC returns the value as a decimal number. If *string* is null (contains no characters) BASIC09 returns Error 67 (Illegal Argument).

Parameters:

string	Any string type variable or constant.

Examples:

```
PRINT ASC("Hello")

X = ASC(A$)
```

Sample Program:

The following procedure determines whether the first character you enter is a hexadecimal digit. To do this, it gets the ASCII value of the character and compares it to the ranges for characters between 1 and 0 and A and F.

```
PROCEDURE hexcheck
□DIM A:INTEGER
□DIM HEXNUM:STRING
□LOOP
□INPUT "Enter a hexadecimal value...",HEXNUM
□A=ASC(HEXNUM) \          (* GET THE ASCII CODE *)
□EXITIF A<48 OR A>57 AND A<65 OR A>70 THEN
□PRINT "Not a hex number."
□END
□ENDEXIT
□PRINT "Ok."
□ENDLOOP
□END
```

ASN Returns arcsine

Syntax: **ASN(*number*)**

Function: Calculates the arcsine of *number*. ASN expresses its result in radians unless you specify otherwise (see DEG).

Parameters:

number	The number for which you want to calculate the arcsine.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

The following program calculates the arcsine of a number you enter and expresses the result in degrees.

```
PROCEDURE arcsine
□DIM NUM:REAL
□DEG
□INPUT "Enter a number (-1 to 1) ",NUM
□PRINT "The arcsine of a "; NUM; " is---";
ASN(NUM)
□END
```

ATN Returns arctangent

Syntax: **ATN(*number*)**

Function: Calculates the arctangent of *number*.

Parameters:

number	The number for which you want to find the arctangent.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

This procedure calculates arcsine, arccosine, and arctangent for a value you enter.

```
PROCEDURE anglecalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number ",NUM
□PRINT
□PRINT " ","Arcsine","Arccosine","Arctangent"
□PRINT "Number","Degrees","Degrees","Degrees"
□PRINT "----------------------------------------
--------------------"
□IF NUM>1 OR NUM<-1 THEN
□PRINT NUM,"UNDEF","UNDEF",ATN(NUM)
□PRINT
□END
□ENDIF
□PRINT NUM,ASN(NUM),ACS(NUM),ATN(NUM)
□PRINT
□END
```

BASE Set array base

Syntax: **BASE 0**
BASE 1

Function: Sets a procedure's lowest array or data structure index to either 0 or 1. If you want to have the first elements in arrays set to 0, you must include `BASE Ø` at the beginning of the procedure.

The BASE statement does not affect string operations such as MID$, RIGHT$, and LEFT$. BASIC09 always indexes the first character of a string as 1.

Parameters:

0 or 1	If you do not indicate a BASE setting in a procedure, BASIC09 uses a default of 1.

Examples:

```
BASE Ø
```

Sample Program:

This procedure determines how many times RND selects each number between 0 and 11 out of 1000 selections. It stores the results in an array of 12 elements. Because it specifies BASE 0, one of the elements in the array is 0. Whenever the procedure picks a random number, it increments the value in the corresponding array number by one.

```
PROCEDURE randomtest
□BASE Ø                          (* set the array base at Ø.
□DIM RND__ARRAY(12),X,R:INTEGER (* dimension array to hold results.

□FOR X=Ø TO 11
□RND__ARRAY(X)=Ø                 (* initialize array elements at zero.
□NEXT X
□SHELL "TMODE -PAUSE"            (* turn off screen pause.

□FOR X=1 TO 1ØØØ
□R=RND(11)                       (* select random number 1ØØØ times.
```

```
□RND__ARRAY(R)=RND__ARRAY(R)+1
                                    (* add 1 to appropriate element.
□PRINT 1001-X                        (* count down from 1000 to 1.
□NEXT X

□FOR X=0 TO 11
□PRINT "RND selected "; X; " "; RND__ARRAY(X); "
times."                              (*display array
□NEXT X
□SHELL "TMODE PAUSE"                 (* turn scroll lock back on.
□END
```

BYE End procedure, terminate BASIC09

Syntax: BYE

Function: Ends execution of a procedure and terminates BASIC09. The statement closes any open files, but you lose any unsaved procedures or data.

Use BYE to exit packed programs that you call from OS-9 and especially programs that you call from procedure files.

Parameters: None

Examples:

```
INPUT "Press ENTER to return to the system.";Z$
BYE
```

Sample Program:

This procedure calculates the payments and interest of a loan. When it is through, it exits the procedure and BASIC09 with a BYE statement.

```
PROCEDURE loan
□DIM PRIN,LENG,RATE,MONPAY:REAL
□DIM RESPONSE:STRING[1]
□REPEAT
□PRINT "Amortization Program"
□INPUT "How much do you want to borrow?...",PRIN
□INPUT "For how many months?...",LENG
□INPUT "At what interest rate?...",RATE
□A=RATE/1200 .
□B=1-1/(1+A)^LENG
□MONPAY=PRIN*A/B
□MONPAY=INT(MONPAY*100+.5)/100
□PRINT "Monthly payments are...$";
□PRINT USING "R12.2<",MONPAY
□PRINT "The total interest to pay is...$";
□PRINT USING "r12.2<",MONPAY*LENG-PRIN
□PRINT
□INPUT "Do another calculation?...",RESPONSE
□PRINT
□PRINT
□UNTIL RESPONSE<>"Y"
□BYE
□END
```

CHAIN Execute another module

Syntax: **CHAIN "*module* [*parameters*][...]"**

Function: CHAIN performs an OS-9 chain operation, passing *module* as the name of a program to execute. If you include other parameters, CHAIN passes them to the executing module. The module must be programmed to expect parameters of the type you provide.

CHAIN exits BASIC09, unlinks BASIC09, and returns the freed memory to OS-9.

CHAIN can begin execution of any module, not only BASIC09 modules. It executes the module indirectly through the shell in order to take advantage of the shell's parameter processing. This has the side effect of leaving the initiated shells active. Programs that repeatedly chain to each other eventually fill memory with waiting shells. To prevent this, use the EX option to initialize a shell.

BASIC09 does not close files that are open when you execute CHAIN. However, the OS-9 FORK call passes only the standard I/O paths (0, 1, and 2) to a child process. Therefore, if you need to pass an open path to another program segment, use the EX shell option.

Parameters:

module	The name of the procedure module you want BASIC09 to execute.
parameters	String data passed to the *chained* module.

Examples:

```
CHAIN "ex BASIC09 menu"

CHAIN "BASIC09 #10k sort (""datafile"",
""tempfile"")"

CHAIN "DIR /D0"

CHAIN "Dir; Echo *** Copying Directory ***; ex
basic09 copydir"
```

Sample Program:

This procedure chains to two others to display a directory or a file. It uses CHAIN to call the procedures.

```
PROCEDURE chaining
□DIM RESPONSE:BYTE
□PRINT USING "S26^","- MENU -" (* print menu title.
□PRINT
□PRINT "1. List current data directory" (* print menu.
□PRINT "2. Display a file"
□PRINT "3. Exit to system"
□PRINT
□INPUT "Select a function (1-3) ",RESPONSE (* function you want.
□ON RESPONSE GOTO 100,200,300 (* select appropriate function.
100□CHAIN "EX BASIC09 dirlook" (* chain to list directory.
200□CHAIN "EX BASIC09 display" (* chain to list file.
300□BYE

PROCEDURE dirlook
□REM Lists the specified directory

□SHELL "DIR"          (* execute dir command.
□CHAIN "EX BASIC09 chaining" (* chain back to calling proc.
□END

PROCEDURE display
□REM Lists the specified file.

□DIM FILE,JOB:STRING
□INPUT "Path of file to display...",FILE
□JOB="LIST "+FILE
□SHELL JOB          (* list specified file.
□CHAIN "EX BASIC09 chaining" (* chain back to calling proc.
□END
```

CHD Change data directory
CHX Change execution directory

Syntax: **CHD *dirpath***
CHX *dirpath*

Function: Changes the current data or execution directory.

Parameters:

dirpath An existing data or execution directory.

Examples:

```
CHD "/D1/ACCOUNTS/RECEIVABLE"

CHX "/D1/CMDS"

CHD ".."
```

Sample Program:

This procedure creates a directory, and makes it the data directory. Then, it creates a file in the new directory, exits the new directory, and deletes the file and the directory.

```
PROCEDURE chdtest
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"            (* create new directory named TEST.
□CHD "TEST"                     (* make TEST the data directory.

□CREATE #PATH,"samplefile":WRITE (* create a file in TEST.
□REM      Write data into the new file
□WRITE #PATH,"This file is for testing only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH

□SHELL "LIST samplefile"        (* list the new file.
□CHD ".."                       (* make the ROOT the data directory.
□SHELL "DEL TEST/samplefile"    (* delete the file.
□SHELL "DELDIR TEST"            (* delete the directory.
□END
```

CHR$ Return ASCII character

Syntax: **CHR$(*code*)**

Function: Returns the ASCII character for the value of *code*. CHR$ is the inverse of the ASC function, which returns the ASCII code for a given character. For a complete listing of ASCII codes, see Chapter 9.

Parameters:

code	The ASCII value for a keyboard character or special block graphics character.

Examples:

```
PRINT CHR$(88)
```

Sample Program:

By increasing by one the ASCII values of characters you type, the following program creates a secret code. It uses CHR$ to display the secret code.

```
PROCEDURE secret
□DIM TEXT,SECRETLINE:STRING[80]
□DIM T,CODECHAR:INTEGER
□TEXT=""
□SECRETLINE=""

□PRINT "Type a line to code in capital letters..."
□INPUT TEXT                    (* you type a line.
□FOR T=1 TO LEN(TEXT)
□CODECHAR=ASC(MID$(TEXT,T,1)) (* look at each character in line.
□IF CODECHAR=90 THEN           (* is it "Z"? If yes then
□CODECHAR=64                   (* make it one less than "A".
□ENDIF
□IF CODECHAR=32 THEN           (* is character a space? If yes then
□CODECHAR=31                   (* decrease its value by one.
□ENDIF

□SECRETLINE=SECRETLINE+CHR$(CODECHAR+1) (* add 1 to characters.
□NEXT T
□PRINT SECRETLINE              (* print the secret code.
□END
```

CHX Change execution directory
CHD Change data directory

Syntax: **CHX *dirpath***
CHD *dirpath*

Function: Changes the current execution or data directory.

Parameters:

dirpath	An existing execution or data directory.

Examples:

```
CHX "/D1/CMDS"
CHD "/D1/ACCOUNTS/RECEIVABLE"
CHD ".."
```

CLOSE Deallocate file or device path

Syntax: **CLOSE *#pathnum***

Function: Deallocates the file or device path specified by *pathnum*.

When you OPEN or CREATE a file, BASIC09 allocates a path number to the variable you supply in the OPEN or CREATE command. The system then *knows* the path by that number. If the path you CLOSE is to a non-shareable device (such as a printer), the system releases the device for other use. Do not close paths 0, 1, and 2 (the standard I/O paths) unless you immediately open a new path to take over the standard path number.

Parameters:

pathnum	The name of variable containing the path number or the actual number of the path to a file or device.

Examples:

```
CLOSE #FILEPATH, #PRINTERPATH, #TERMPATH

CLOSE #5, #6, #7

CLOSE #1     \ (* closes the standard output path *)
OPEN #PATH,"/T1"     \ (* redirects standard output *)
```

Sample Program:

This procedure creates a directory named TEST and changes it to the data directory. It then creates a file named Samplefile and writes data to the file. Finally it changes back to the parent directory and deletes Samplefile and TEST.

```
PROCEDURE close
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"
□CHD "TEST"
□CREATE #PATH,"samplefile":WRITE (* create a new file.
□WRITE #PATH,"This file is for testing only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□CHD ".."
□SHELL "DELDIR TEST"
□END
```

COS Return cosine

Syntax: COS(*number*)

Function: Calculates the cosine of *number*. Unless you specify DEG, COS interprets the value of *number* in radians.

Parameters:

number	The number for which you want to find the cosine.

Examples:

```
PRINT COS(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent of a value you enter.

```
PROCEDURE ratiocalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number...",NUM
□PRINT
□PRINT "Number","SINE","COSINE","TAN"
□PRINT "----------------------------------------
--------------------"
□PRINT ANGLE,SIN(NUM),COS(NUM),TAN(NUM)
□PRINT
□END
```

CREATE Establish a disk file.

Syntax:
CREATE *#path,"pathlist"* [*access mode*]
[+*access mode*][+...]

Function: Creates a file on a disk. When you create a file, you can select one or more of the following access modes for the file:

Mode	Function
READ	Lets you read (receive) data from a file but does not let you write (send) data to the file.
WRITE	Lets you write data to a file but does not let you read data from a file.
UPDATE	Lets you both read from and write to a file.

Parameters:

path	The name of the variable in which BASIC09 stores the number of the opened path.
pathlist	The route to the file or device to be opened, including the filename, if appropriate.
access mode	The type of access to be allowed for the file or device. Use plus symbols to allow more than one type of access with a single file.

Notes:

- You can access files either sequentially or randomly. With random access, you must establish the filing system you want for a particular application.
- Files are byte-addressed, and you are not restricted by explicit record lengths. You can read the data one byte at a time, or in whatever size portions you want.

- A new file has a size of zero. OS-9 then expands the file automatically when PRINT, WRITE, or PUT statements write beyond the current end-of-file.

Examples:

```
CREATE #TRANS,"transportation":UPDATE

CREATE #SPOOL,"/user4/report":WRITE

CREATE #OUTPATH,name$:UPDATE+EXEC
```

Sample Program:

This procedure CREATEs a directory named TEST and makes it the data directory. It creates a file in TEST named Samplefile, writes data to the file, then resets the parent directory as the data directory. Finally, it deletes Samplefile and TEST.

```
PROCEDURE close
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"
□CHD "TEST"
□CREATE #PATH,"samplefile":WRITE (* create a file.
□WRITE #PATH,"This file is for testing purposes only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□CHD ".."
□SHELL "DELDIR TEST"
□END
```

DATA Store numeric and string information

Syntax: DATA "*item*"[,"*item*",...]

Function: Stores numeric and string constants to be accessed by a READ statement. A DATA line can contain up to 254 characters. Each item in the list must be separated by commas.

You can place DATA statements anywhere in a procedure that is convenient. BASIC09 reads sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement.

The following rules apply to data items:

- You must place all string data between quotation marks.
- To include quotes in string-type data, use consecutive quotation marks, like this: `DATA "He said, ""go home"" to me".`
- You can use RESTORE to reset the data *pointer*. Using RESTORE without an argument resets the pointer to the beginning of the data items. Using RESTORE with a line number, resets the pointer to the first item in the specified line.
- The READ statement can support a list of one or more variable names of various types. The data types in DATA statements must match the variable types used in the corresponding READ statements.
- You can include arithmetic expressions in data items. READ causes the expressions to be evaluated and returns the result of the expression as the data item.

Parameters:

item	Numeric or string characters. Enclose string characters in quotation marks.

Examples:

```
DATA 1.1,1.5,9999,"CAT","DOG"
DATA SIN(TEMP/25), COS(TEMP*PI)
DATA TRUE,FALSE,TRUE,TRUE,FALSE
DATA "The rain in spain","falls mainly on the
plain"
```

Sample Program:

This procedure calculates the day of the week for a date you enter. A data statement contains the names of the weekdays.

```
PROCEDURE weekday
□DIM X,DAY,MONTH,YEAR,CALC:INTEGER
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,INUM:
INTEGER
□DIM WEEKDAY(7):STRING[9]
□PRINT USING "S60^","Day of the Week Program"
□PRINT USING "S60^","For any year after 1752"
□PRINT
□INPUT "Enter day of the month as two digits, such
as 08...",DAY
□INPUT "Enter month as two digits, such as
12...",MONTH
□INPUT "Enter year as four digits, such as
1986...",YEAR
□FOR X=1 TO 7
□READ WEEKDAY(X)
□NEXT X
□ANUM=INT(.6+1/MONTH)
□BNUM=YEAR-ANUM
□CNUM=MONTH+12*ANUM
□DNUM=BNUM/100
□ENUM=INT(DNUM/4)
□FNUM=INT(DNUM)
□GNUM=INT(5*BNUM/4)
□HNUM=INT(13*(CNUM+1)/5)
□INUM=HNUM+GNUM-FNUM+ENUM+DAY-1
□INUM=INUM-7*INT(INUM/7)+1
□PRINT
□PRINT "The day of the week on "; DAY; "/"; MONTH;
□PRINT "/"; YEAR; " is..."; WEEKDAY(INUM)
□DATA "Sunday","Monday","Tuesday","Wednesday",
"Thursday"
□DATA "Friday","Saturday"
□END
```

DATE$ Provide date and time

Syntax: DATE$

Function: Returns the date and time. The OS-9 internal date is kept in the format:

year/month/day hour:minutes:seconds

If your OS-9 Startup file contains the SETIME command, the system asks you to enter the date and time whenever it boots. If it does not contain the SETIME command, the date and time start from 86/09/01:00:00:00.

You can use the normal string functions to access the data contained in DATE$, but you cannot use functions or operations that attempt to change or append to its values. To reset the date or time or both, use the SHELL command, such as:

```
SHELL "SETIME"
```

Parameters: None

Examples:

```
PRINT DATE$
```

Sample Program:

This program is essentially the same as the sample program for the DATA statement, except that it gets the day, month, and year from DATE$.

```
PROCEDURE date
□DIM X,DAY,MONTH,YEAR,CALC:INTEGER
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,INUM:INTEGER
□DIM WEEKDAY(7):STRING[9]
□MONTH=VAL(MID$(DATE$,4,2)) (* get month from DATE$.
□DAY=VAL(MID$(DATE$,7,2)) (* get day from DATE$.
□YEAR=VAL("19"+LEFT$(DATE$,2)) (* get year from DATE$.

□FOR X=1 TO 7
□READ WEEKDAY(X)
```

```
□NEXT X
□ANUM=INT(.6+1/MONTH)
□BNUM=YEAR-ANUM
□CNUM=MONTH+12*ANUM
□DNUM=BNUM/100
□ENUM=INT(DNUM/4)
□FNUM=INT(DNUM)
□GNUM=INT(5*BNUM/4)
□HNUM=INT(13*(CNUM+1)/5)
□INUM=HNUM+GNUM-FNUM+ENUM+DAY-1
□INUM=INUM-7*INT(INUM/7)+1
□PRINT
□PRINT "Today is "; WEEKDAY(INUM)

□DATA "Sunday","Monday","Tuesday","Wednesday","Thursday","Friday"
□DATA "Saturday"
□END
```

DEG Return trigonometric calculations in degrees

Syntax: DEG

Function: Causes a procedure to calculate trigonometric values in degrees. If you do not include the DEG statement, procedures produce radian values.

Parameters: None

Examples:

```
DEG
```

Sample Program:

This procedure calculates the sine, cosine, and tangent for a value you enter. Because it uses the DEG statement, it displays the results in degrees.

```
PROCEDURE degcalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number...",NUM
□PRINT
□PRINT "Number","SINE","COSINE","TAN"
□PRINT "-----------------------------------------
--------------------"
□PRINT NUM,SIN(NUM),COS(NUM),TAN(NUM)
□PRINT
□END
```

DELETE Erase a disk file

Syntax: **DELETE "*pathname*"**

Function: DELETE removes a file from disk storage and releases the portion of the disk on which it resides. When you DELETE a file, it is permanently lost.

Parameters:

pathname The complete pathlist to the file you want to delete, including the drive and one or more directories, if appropriate. You must surround the pathlist with quotation marks.

Examples:

```
DELETE "myfile"

DELETE "/D1/ACCOUNTS/receivables"
```

Sample Program:

This procedure creates a file named Samplefile, writes data to the file, then closes it. It then lists the file before deleting it.

```
PROCEDURE close
□DIM PATH:BYTE
□CREATE #PATH,"samplefile":WRITE (* create a file.
□WRITE #PATH,"This file is for testing purposes only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□DELETE "samplefile"
□END
```

DIM Assign variable storage

Syntax: DIM ***variable***[,...][:***type***][;***variable***][,...][:***type***][...]

Function: Assigns storage space and declares types for variables, arrays, or complex data structures.

Parameters:

variable	A simple variable, an array structure, or a complex data structure.
type	BYTE, INTEGER, REAL, BOOLEAN, STRING, or user defined.

Notes:

- You declare simple arrays with DIM by using the variable name, without a subscript. If you do not explicitly declare variables, the system makes them type real unless they are followed by a dollar sign ($). The system dimensions variables ending with a dollar sign ($) as strings, with a length of 32 bytes. You must declare types of all other simple variables as to type.
- You can declare several variables of the same type by separating them with commas. To separate variables of different types, follow each type group with a colon, the type name, and then a semicolon.
- Define a maximum length for a string variable by enclosing the length in brackets following the type, like this:

```
DIM name:string[25]
```

If you do not define a maximum length, BASIC09 uses a default length of 32 characters. You can declare a shorter length or a longer length, up to the capacity of BASIC09's memory. If you try to extend a string beyond its declared length, or beyond the default length, the system ignores all extra characters. Thus the following:

```
DIM name:string[10]
name = "Abbernathinsky"
```

produces the string:

```
Abbernathi
```

- Arrays can have one, two, or three dimensions. The DIM format for dimensioned arrays is the same as for simple variables, except that you must follow each array name with a subscript, enclosed in parentheses, to indicate its size. The maximum array size is 32767.

  Arrays can be either of the standard BASIC09 type or of a user-defined type. For information on creating your own types for simple variables, arrays, and complex data structures, see TYPE.

Examples:

```
DIM logical:BOOLEAN

DIM a,b,c:INTEGER

DIM name,address,zip:STRING

DIM name:STRING[25]; address:STRING[30];
zip:INTEGER

DIM no1,no2,no3:REAL;no4,no5,no6:INTEGER;
no7:BYTE
```

Sample Program:

This procedure randomly selects letters and vowels to create six-letter words that might look like alien names. It first DIMs nine string variables to contain the letters selected for each name. It DIMs two integer variables to provide a loop counter and to store the number of names you request.

When asked, type the number of names you want to have the procedure generate.

```
PROCEDURE alien
□DIM B,BEGIN,F,FINISH:STRING
□DIM VOWELS,VOWEL1,VOWEL2:STRING
□DIM MID1,MID2:STRING
□DIM T,RESPONSE:INTEGER
□VOWELS="aeiouy"

□INPUT "How many alien names do you want to
see?...",RESPONSE
□BEGIN="ABCDFGHJKLMNPRSTVWXZ"
□FINISH="ehlmnprstvwyz"

□FOR T=1 TO RESPONSE
□B=MID$(BEGIN,RND(19)+1,1)
□F=MID$(FINISH,RND(12)+1,1)
□MID1=CHR$(RND(25)+97)
□MID2=CHR$(RND(25)+97)
□VOWEL1=MID$(VOWELS,RND(5)+1,1)
□VOWEL2=MID$(VOWELS,RND(5)+1,1)
□PRINT B; VOWEL1; MID1; MID2; VOWEL2; F,
□NEXT T

□PRINT
□END
```

DO Execute procedure lines in a loop

Syntax: **WHILE *expression* DO**
proclines
ENDWHILE

Function: Establishes a loop that executes the procedure lines between DO and ENDWHILE as long as the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless *expression* is true.

Parameters:

expression — A Boolean expression (produces a result of True or False).

proclines — Are program lines to execute if the expression is true.

See WHILE/DO/ENDWHILE for more information.

ELSE Execute alternate action

Syntax: **IF *condition* THEN**
action
ELSE
secondary action
ENDIF

Function: ELSE provides access to a secondary action within an IF/THEN test. When the condition tested by IF is **not** true, BASIC09 executes the *secondary action* preceded by ELSE.

Parameters:

condition	A Boolean expression (produces a result of True or False).
action	A line number to which the procedure is to transfer execution, or a program statement. If *action* is a line number, do not include the ENDIF statement in the IF test.
secondary action	One or more program statements.

For more information, see IF/THEN/ELSE

END Terminate a procedure

Syntax: **END ["*text*"]**

Function: Ends procedure execution and returns to the calling procedure, or to the highest level procedure. If you provide output text for END, it functions in the same manner as PRINT. You can use END several times in the same procedure. END is not required as the last statement in a procedure.

Parameters:

text	A literal string or a string-type variable.

Examples:

```
END "Program Terminated"

LAST$="Session over"
END LAST$
```

Sample Program:

This procedure calculates a loan's term, using END to terminate routines.

```
PROCEDURE loaner
□DIM YOUPAY,PRINCIPLE,INTEREST,NUMPAY,YEARS,
MONTHS:REAL
□DIM RESPONSE:STRING[1]
□REPEAT
□PRINT
□PRINT USING "S45^","Loan Terms"
□PRINT
□INPUT "   Amount of Regular Payments...",YOUPAY
□INPUT "   Enter the Principle...",PRINCIPLE
□INPUT "   Enter the Annual Interest Rate...",
INTEREST
□INPUT "   Enter the Number of Payments
Yearly...",NUMPAY
```

```
□YEARS=-(LOG(1-PRINCIPLE*(INTEREST/100)/
(NUMPAY*YOUPAY))/(LOG(1+INTEREST/100/NUMPAY)*
NUMPAY))
□MONTH=INT(YEARS*12+.5)
□YEARS=INT(MONTH/12)
□MONTH=MONTH-YEARS*12
□PRINT "   The Term of Your Loan is "; YEARS; "
years and "; MONTH; " months."
□INPUT "Calculate another or Quit (C/Q)?...",
RESPONSE
□UNTIL RESPONSE<>"C" AND RESPONSE<>"c"
□END "Goodbye...I hope I helped you."
```

ENDEXIT Leave loop if a condition is True

Syntax: **EXITIF *condition* THEN**
proclines
ENDEXIT

Function: ENDEXIT terminates an EXITIF test. You always use EXITIF/THEN/ENDEXIT inside a procedure loop. If the Boolean expression tested by EXITIF is true, BASIC09 executes the program statements between THEN and ENDEXIT and then transfers program operation outside the loop. If the condition tested by EXITIF is not true, loop execution continues at the statement following ENDEXIT.

Parameters:

condition A comparison operation that returns either True or False, such as A=B, A<B, or A=B=C.

proclines One or more statements to perform if the Boolean expression tested by EXITIF is True.

For more information, see EXITIF/THEN/ENDEXIT

ENDIF Close IF statement

Syntax: **IF *condition* THEN**
action
[ELSE
***secondary action*]**
ENDIF

Function: ENDIF terminates an IF/THEN condition test. If the condition tested by IF is true, BASIC09 executes the statements between THEN and ENDIF. If the condition tested by IF is *not* true, BASIC09 transfers execution to the procedure line following ENDIF or (optionally) executes the statements following ELSE.

Parameters:

condition	A Boolean expression (produces a result of True or False).
action	A line number to which the procedure is to transfer execution. *Action* can also be a program statement. If *action* is a line number, do not include the ENDIF statement in the IF test.
secondary action	A program statement.

For more information, see IF/THEN/ELSE/ENDIF.

ENDLOOP Close LOOP statement

Syntax: **LOOP**
statement(s)
ENDLOOP

Function: ENDLOOP terminates a procedure loop established by the LOOP command. BASIC09 endlessly executes all procedure statements between LOOP and ENDLOOP repeatedly unless a condition test within the loop (such as EXITIF/THEN/ENDEXIT, or IF/THEN) transfers execution outside of the loop.

Parameters:

statement(s) One or more procedure lines that execute within the loop.

For more information, see LOOP/ENDLOOP.

ENDWHILE Close WHILE statement

Syntax: **WHILE *condition* DO**
proclines
ENDWHILE

Function: Forms the bottom of a WHILE loop. WHILE causes the procedure lines between DO and ENDWHILE to execute as long as the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless the expression is true.

Parameters:

condition A Boolean expression (produces results of True or False).

proclines Are program lines to execute if the expression is true.

For more information, see WHILE/DO/ENDWHILE.

EOF Test for end-of-file

Syntax: **EOF(*path*)**

Function: Tests for the end of a disk file. The function returns a value of True when it encounters an end-of-file; otherwise, it returns False. Use EOF with a READ or GET statement.

Parameters:

path	The number of the path you are accessing. BASIC09 automatically stores a path number into the variable you specify during a CREATE or OPEN operation.

Examples:

```
IF EOF(#PATH) THEN
CLOSE #PATH
ENDIF
```

Sample Program:

This procedure redirects a listing of the current directory into a file named Dirfile. It then lists Dirfile to the screen. EOF tells the WHILE/ENDWHILE loop when the READ operation reaches the end of the file.

```
PROCEDURE readfile
□DIM A:STRING[80]
□DIM PATH:BYTE
□SHELL "DIR > dirfile"
□OPEN #PATH,"dirfile":READ
□WHILE NOT EOF(#PATH) DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

ERR Return error code

Syntax: **ERR**

Function: Returns the error code of the most recent error. BASIC09 automatically sets the ERR code to zero after you reference it. ERR is only useful when used in conjunction with BASIC09's ON ERROR error trapping functions.

See Appendix A for a list of all BASIC09 error codes.

Parameters: None

Examples:

```
ERRNUM = ERR
IF ERRNUM = 218 THEN
PRINT "File already exists. Please use another
filename."
ENDIF
```

Sample Program:

This procedure displays the contents of a file you select. If the file doesn't exist (Error 216, Pathname not found), the procedure uses ERR to tell you. If an error other than Error 216 occurs, the procedure displays `I can't handle error` *xx*, where *xx* is the code of the error.

```
PROCEDURE readfile
□DIM READFILE:STRING; A:STRING[80]; PATH:BYTE
10□INPUT "Type the pathlist of the file to read...",READFILE
□ON ERROR GOTO 100  (* if an error occurs, skip to line 100.
□OPEN #PATH,READFILE:READ
□WHILE EOF(#PATH)<>TRUE DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
100□ERRNUM=ERR (* store the error code in ERRNUM.
□IF ERRNUM=216 THEN (* if file doesn't exist say so.
□PRINT "I can't find the file...Please try again."
□ON ERROR
□GOTO 10
□ENDIF
□PRINT "Sorry, I can't handle error number "; ERRNUM (* other error.
□CLOSE #PATH
□END
```

ERROR Simulate an error

Syntax: **ERROR** ***code***

Function: Simulates the error specified by *code*. You would mainly use this command to test ON ERROR GOTO routines. When BASIC09 encounters an ERROR statement, it proceeds as if the error corresponding to the specified code has occurred. Refer to Appendix A for a listing of error codes and their meanings.

Parameters:

code	The code of the error you want to simulate.

Examples:

```
ERROR 207
ERRNUM = ERR
IF ERRNUM = 207 THEN
PRINT "Memory is full. The current data is being
saved to disk."
ENDIF
```

Sample Program:

This program creates a file named Test1. Before creating the file, it checks to see if it already exists. If the file exists, the procedure deletes it. An error trap catches any error that might occur. To test if the trap works for Error 216, "Pathname not found", the statement ERROR 216 is inserted as the fourth line. After testing the trap to make sure it works, delete this line to use the procedure.

```
PROCEDURE errortest
□DIM PATH,ERRNUM:BYTE; RESPONSE:STRING[1]
□BASE 0
□ON ERROR GOTO 10     (* set error trap
□ERROR 216            (* simulate error
□DELETE "test1"
□GOTO 100
10□ERRNUM=ERR
□IF ERRNUM=216 THEN
□INPUT "File doesn't exist...continue?
(Y/N)",RESPONSE
□IF RESPONSE="N" THEN
□END "Procedure terminated at your request..."
□ENDIF
□ENDIF
□ON ERROR             (* turn off error trap.
100□CREATE #PATH,"test1":WRITE
□END
```

EXITIF/THEN/ENDEXIT

Exit from loop if a condition is true

Syntax: **EXITIF *condition* THEN**
statement
ENDEXIT

Function: Use these statements with loop constructions (particularly LOOP and ENDLOOP) to provide an exit for what is otherwise an endless loop. EXITIF performs a test of a Boolean expression, such as A<B. The THEN statement precedes any operation you want to execute if the expression is true. You must always follow EXITIF with an ENDEXIT.

If the Boolean expression following an EXITIF is false, execution of the program transfers to the statement immediately following the body of the loop (after the ENDEXIT statement). Otherwise, BASIC09 executes the statement(s) between EXITIF and ENDEXIT, then transfers control to the statement following the body of the loop.

You can also use EXITIF and ENDEXIT with types of loop constructions other than LOOP/ENDLOOP.

Parameters:

Boolean expression	A comparison operation that returns either True or False, such as A=B, A<B, or A=B=C.
statement	An operation to be performed if the Boolean expression tested by EXITIF is True, such as: `PRINT A is less than B.`

Examples:

```
LOOP
COUNT=COUNT+1
EXITIF COUNT>100 THEN
DONE = TRUE
ENDEXIT
PRINT COUNT
X = COUNT/2
ENDLOOP
```

Sample Program:

This procedure simulates a gambling machine by randomly selecting among several fruit names and displaying them. It gives you a starting stake of $25 and, depending on the combination of fruit selected, it adds or subtracts from your stake.

If your stake drops to zero, an EXITIF statement ends the procedure and tells you that you're broke.

```
PROCEDURE onearm
□DIM FRUIT1,FRUIT2,FRUIT3,STAKE:INTEGER; FRUIT(8):
STRING[6]
□STAKE=25
□PRINT  \ PRINT "You have $"; STAKE; " to play
with."
□FOR T=1 TO 8
□READ FRUIT(T)
□NEXT T
□LOOP
□FRUIT1=RND(7)+1 \FRUIT2=RND(7)+1 \FRUIT3=RND(7)+1
□PRINT FRUIT(FRUIT1); " "; FRUIT(FRUIT2); " ";
FRUIT(FRUIT3)
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) AND FRUIT(FRUIT1)=
FRUIT(FRUIT3) THEN STAKE=STAKE+10
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) OR FRUIT(FRUIT1)=
FRUIT(FRUIT3) OR
□FRUIT(FRUIT2)=FRUIT(FRUIT3) THEN
□STAKE=STAKE+1
□ELSE
□STAKE=STAKE-1
□ENDIF
□ENDIF
```

```
□REM exit play loop is stake is less than $1.
□EXITIF STAKE<1 THEN
□PRINT
□PRINT "You're Busted...Better go home."
□ENDEXIT
□PRINT "Your stake is now $"; STAKE; "."
□PRINT
□PRINT
□INPUT "Press ENTER to pull again...",Z$
□ENDLOOP
□END
□DATA "ORANGE","APPLE","CHERRY","LEMON","BANANA"
□DATA "PEAR","PLUM","PEACH"
```

EXP Return natural exponent

Syntax: **EXP(*number*)**

Function: Returns the natural exponent of *number*, that is, e (2.71828183) to the power of *number*. *Number* must be positive.

This function is the inverse of the LOG function. Therefore, *number* = EXP(LOG(*number*)).

Parameters:

number A positive value.

Examples:

```
PRINT EXP(2)
```

Sample Program:

This procedure calculates the exponent of values in the range 0-1.

```
PROCEDURE exprint
□FOR T=0 TO 1 STEP .03
□PRINT EXP(T),EXP(T+.01),EXP(T+.02)
□NEXT T
□END
```

FALSE Assign Boolean value

Syntax: *variable*=FALSE

Function: FALSE is a Boolean function that always returns False. You can use FALSE and TRUE to assign values to Boolean variables.

Parameters: None

Examples:

```
DIM TEST:BOOLEAN
TEST=FALSE
```

Sample Program:

The procedure uses a Boolean variable to store True or False, depending on whether you answer some questions correctly or incorrectly.

```
PROCEDURE quiz
□DIM REPLY,VALUE:BOOLEAN; ANSWER:STRING[1];
QUESTION:STRING[80]
□FOR T=1 TO 5
□READ QUESTION,VALUE
□PRINT QUESTION
□PRINT "(T) = TRUE□□□□□□□(F) = FALSE"
□PRINT "Select T or F:□□";
□GET #1,ANSWER
□IF ANSWER="T" THEN
□REPLY=TRUE
□ELSE
□REPLY=FALSE
□ENDIF
□IF REPLY=VALUE THEN
□PRINT  \ PRINT "That's Correct...Good Show!"
□ELSE
□PRINT "Sorry, you're wrong...Better Luck next
time."
□ENDIF
□PRINT  \ PRINT  \ PRINT
```

```
□NEXT T
□DATA "In computer talk, CPU stands for Central
Packaging Unit.", FALSE
□DATA "The actual value of 64K is 65536
bytes.",TRUE
□DATA "The bits in a byte are normally numbered 0
through 7?",TRUE
□DATA "BASIC09 has four data types.",FALSE
□DATA "The LAND function is a Boolean type
operator.",FALSE
□END
```

FIX Round a real number

Syntax: **FIX(*value*)**

Function: Rounds a real number to the nearest whole number and converts it to an integer-type number. Fix performs a function that is the opposite of the FLOAT function.

Parameters:

value	Any real number.

Examples:

```
A=RND(10)
PRINT FIX(A)
```

Sample Program:

This procedure displays the FIXed values of seven constants.

```
PROCEDURE printfix
□PRINT FIX(1.2)
□PRINT FIX(1.3)
□PRINT FIX(1.5)
□PRINT FIX(1.8)
□PRINT FIX(99.566666)
□PRINT FIX(50.1)
□PRINT FIX(.7654321)
□PRINT FIX(-12.44)
□PRINT FIX(-9.99)
□END
```

FLOAT Convert from integer or byte to real

Syntax: **FLOAT(*value*)**

Function: Converts an integer- or byte-type value to real type. FLOAT performs a function that is the opposite of the FIX function.

Parameters:

value	An integer- or byte-type number.

Examples:

```
DIM TEST:INTEGER
TEST=44
PRINT FLOAT(TEST)/3
```

Sample Program:

This procedure uses FLOAT to produce a real number result of an inch to centimeter conversion.

```
PROCEDURE convert
□DIM T:INTEGER; MEASURE:STRING[11]
□FOR T=1 TO 10
□IF T=1 THEN
□MEASURE="centimeter "
□ELSE
□MEASURE="centimeters"
□ENDIF
□PRINT T; " "; MEASURE; " is "; FLOAT(T)*.3937;
" inches."
□NEXT T
□END
```

FOR/NEXT/STEP Establish a loop

Syntax:
FOR ***variable*** **=** ***init val*** **TO** ***end val*** **[STEP** ***value*****]**
[*procedure statements*]
NEXT ***variable***

Function: Establishes a procedure loop that lets BASIC09 execute one or more procedure statements a specified number of times. The variables you use can be either integer or real type and can be negative, positive, or both. Loops using integer values execute faster than loops using real values.

BASIC09 executes the lines following the FOR statement until it encounters a NEXT statement. Then it either increases or decreases the initial value by one (the default) or by the value given STEP.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to be executed within the loop.

Notes:

- If you provide an initial value that is greater than the final value, BASIC09 skips the program loop entirely unless you specify a negative STEP value. Specifying a negative value for STEP causes the loop to decrement from the initial value to the end value.

- When execution reaches the NEXT statement in a positive stepping loop, and the step value is less than or equal to the end value, BASIC09 branches back to the line after FOR and repeats the process. When the step value is greater than the end value, BASIC09 transfers execution to the statement following the NEXT statement.
- When execution reaches the NEXT statement in a negative stepping loop, and the step value is greater than or equal to the end value, BASIC09 branches back to the line after FOR and repeats the process. When the step value is less than the end value, execution continues following the NEXT statement.

Examples:

```
FOR COUNTER = 1 to 100 STEP .5
PRINT COUNTER
NEXT COUNTER

FOR X = 10 TO 1 STEP -1
PRINT X
NEXT X

FOR TEST = A TO B STEP RATE
PRINT TEST
NEXT TEST
```

Sample Program:

This procedure uses two *nested* FOR/NEXT loops to produce a multiplication table.

```
PROCEDURE multable
□PRINT USING "S45^","MULTIPLICATION TABLE"
□PRINT
□DIM I,J:INTEGER
□FOR I=1 TO 9
□FOR J=1 TO 9
□IF J>1 THEN
□PRINT I*J; TAB(5*J);
□ELSE PRINT I*J; "| ";
□ENDIF
□NEXT J
□IF I=1 THEN
□PRINT ""
```

```
□PRINT "----------------------------------------
---";
□ENDIF
□PRINT
□NEXT I
□END
```

GET Read a direct-access file record

Syntax: GET *#path,varname*

Function: Reads a fixed-size binary data record from a file or device. Use GET to retrieve data from random access files.

Although you usually use GET with files, you can also use it to receive data for any outputting device, such as a keyboard or another computer. By dimensioning a string variable to the length of input you want, you can use GET to read a specified number of keystrokes, then continue program execution without requiring ENTER to be pressed.

For information about storing data in random access files, see Chapter 8, "Disk Files." Also see PUT, SEEK, and SIZE.

Parameters:

path	A variable name you choose in which BASIC09 stores the number of the path it opens to the device you specify or one of the standard I/O paths (0, 1, or 2).
varname	The variable in which you want to store the data read by the GET statement.

Examples:

```
GET #PATH,DATA$

GET #1,RESPONSE$

GET #INPUT,INDEX(X)
```

Sample Program:

This procedure directs a directory listing to a file named Dirfile. GET then reads the file, one character at a time in order to determine which characters are valid filename characters. The procedure creates a file containing all the filenames in the directory.

```
PROCEDURE filenames
□DIM DIRECTORY,FILENAME:STRING; CHARACTER:STRING[1]; FILES(125):STRING[15];
PATH,COUNT,T:INTEGER
□COUNT=0
□FILENAME=""
□FOR T=1 TO 125 (* initialize array elements to null.
□FILES(T)=""
□NEXT T
□INPUT "Pathlist of directory to read...",DIRECTORY (* dir to copy.
□ON ERROR GOTO 10
□DELETE "dirfile" (* if dirfile already exists, delete it.
10□ON ERROR
□SHELL "DIR "+DIRECTORY+" > dirfile" (* copy directory into file.
□OPEN #PATH,"dirfile":READ (* open the file for reading.
□REPEAT
□REM Get characters from the file until the first carriage return - the
beginning of the first filename.
□GET #PATH,CHARACTER (* get characters from the file.
□UNTIL CHARACTER=CHR$(13)
□REM
20□LOOP
□EXITIF EOF(#PATH) THEN
□GOTO 200 (* quit when end of file.
□ENDEXIT
□REM get a character from the file until it finds a non-valid filename
character.
□GET #PATH,CHARACTER
□REM
□EXITIF CHARACTER<=" " OR CHARACTER>"z" THEN
□GOTO 100
□ENDEXIT
□FILENAME=FILENAME+CHARACTER (* build the filename.
□ENDLOOP
100□WHILE NOT(EOF(#PATH)) DO
□GET #PATH,CHARACTER (*-check for non-valid filename characters.
□EXITIF CHARACTER>" " AND CHARACTER<="z" THEN (* check if valid char.
□COUNT=COUNT+1
□FILES(COUNT)=FILENAME (* store filename in array.
□PRINT FILENAME, (* display the extracted filename.
□FILENAME="" (* set variable to NULL.
□FILENAME=FILENAME+CHARACTER (* last character begins new filename.
□GOTO 20 (* go get the rest of filename.
□ENDEXIT
□ENDWHILE
200□CLOSE #PATH
```

```
□DELETE "dirfile" (* names are all in array so delete file.
□CREATE #PATH,"dirfile":WRITE (* create the file again.
□FOR T=1 TO COUNT
□WRITE #PATH,FILES(T) (* fill the file with individual filenames.
□NEXT T
□CLOSE #PATH
□PRINT
□PRINT "□□□□□□□*The directory has "; COUNT; " entries"
□PRINT□"□□□□□□□□They are now stored in a file named Dirfile."
□END
```

GOSUB/RETURN

Jump to subroutine/ Return from subroutine

Syntax: **GOSUB** ***linenumber***

Function: Branches program execution to the specified line number.

BASIC09 lets you write programs with line numbers or without. You can also mix numbered and un-numbered lines within a single procedure. This means that, to use GOSUB, you need to number only the first line of the subroutine to which you want to branch.

Every subroutine you access with GOSUB must contain a RETURN statement. You can call a subroutine in this manner as many times as you want. When BASIC09 encounters the RETURN, it transfers program execution to the line following the GOSUB statement.

You can precede GOSUB with a test statement, such as IF or WHEN, that makes branching conditional.

You can nest GOSUB statements to any depth, depending on your computer's free memory.

Parameters:

linenumber	The number of the line where procedure execution is to continue.

Examples:

```
GOSUB 100
```

Sample Program:

The following procedure asks you for two numbers and an operator. It determines the line to jump to by the position of the operator in a table. GOSUB sends the procedure to execute the proper routine. RETURN sends the execution back to the main routine. To quit, enter a negative value.

```
PROCEDURE calc
□DIM NUM1,NUM2:REAL; OP:STRING[1]; A:INTEGER
1□INPUT "NUMBER 1 ";NUM1
□IF NUM1<0 THEN
□END
□ENDIF
□INPUT "NUMBER 2 ";NUM2
□INPUT "OPERATOR ";OP
□A=SUBSTR(OP,"+-*/^")
□ON A GOSUB 10,20,30,40,50
□GOTO 1
10□PRINT NUM1+NUM2 \ RETURN
20□PRINT NUM1-NUM2 \ RETURN
30□PRINT NUM1*NUM2 \ RETURN
40□PRINT NUM1/NUM2 \ RETURN
50□PRINT NUM1 NUM2 \ RETURN
□END
```

IF/THEN/ELSE/ENDIF

Test a Boolean expression

Syntax: **IF *condition* THEN *linenumber***
[ELSE
secondary action
ENDIF]

IF *condition* THEN
action
[ELSE
***secondary action*]**
ENDIF

Function: Tests a Boolean expression and executes *action* if the expression is true. Optionally, the statements execute a secondary action if the expression is not true. Each IF statement must be accompanied by THEN. If *action* is a line number, you can omit the ENDIF statement. For instance, both of the following statements operate in the same manner:

```
IF T=5 THEN 10
```

```
IF T=5 THEN
GOTO 10
ENDIF
```

Parameters:

condition	A Boolean expression (produces True or False).
linenumber	A line to which the procedure is to transfer execution if *condition* is true.
action	One or more procedure statements to be executed if *condition* is true.
secondary action	One or more procedure statements to execute if *condition* is false.

Examples:

```
IF A>B THEN 100

IF A<B THEN 100
ELSE
A=A-1
ENDIF

IF TEST=TRUE THEN
PRINT "The test is a success..."
ENDIF

IF A < B THEN
PRINT "A is less than B"
ELSE
PRINT "B is less than A"
ENDIF
```

Sample Program:

The following procedure is a *purge* procedure. Use it only with the GET Sample Program to delete one or more files from your current directory.

The Filenames procedure (see GET) stores the current directory's filenames in Dirfile. This procedure reads Dirfile, displays all the filenames, then asks you for a *wildcard*. Type in characters that identify a group of files you want to delete. The program deletes **all** files that contain, in the same order and case, the characters you type.

For instance, if you have four files named Test, File1, File2, and File3, and you type a wildcard of "File," the procedure deletes File1, File2, and File3, but does not delete Test. Delete all of the files in a directory by typing "*" as the wildcard.

Use this program carefully. Be sure you are in the right directory and that the wildcard characters you type are not contained in filenames other than the ones you want to delete. You might want to add a prompt to the procedure that lets you confirm each deletion before it happens.

```
PROCEDURE purge
□DIM PATH:INTEGER
□DIM NAME(100):STRING
□DIM WILDCARD:STRING
□X=0
□OPEN #PATH,"dirfile":READ
□WHILE NOT(EOF(#PATH)) DO
□X=X+1
□READ #PATH,NAME(X)
□ENDWHILE
□FOR T=1 TO X
□PRINT NAME(T),
□NEXT T
□INPUT "Wildcard Characters...",WILDCARD
□FOR T=1 TO X
□ON ERROR GOTO 100
□IF SUBSTR(WILDCARD,NAME(T))>0 OR WILDCARD="*"
THEN
□PRINT "DELETING "; NAME(T); " ......"
□DELETE NAME(T)
□ENDIF
10□NEXT T
□END
100□PRINT "* * * ERROR * * * "; NAME(T); " cannot
be deleted..continuing."
□GOTO 10
```

INKEY Read a keypress

Syntax: **RUN INKEY(*string*)**

Function: Reads a keypress, and stores the character of the key in the specified string variable.

Parameters:

string is a string variable into which INKEY stores the character you press.

Examples:

```
DIM CHAR:STRING[1]
CHAR=""
WHILE CHAR="" DO
RUN INKEY(CHAR)
ENDWHILE
PRINT ASC(CHAR)
```

Sample Program:

```
PROCEDURE Calculate
□DIM CHAR:STRING[1]
□DIM LOOKUP:STRING[7]
□DIM FIRST,SECOND:REAL
□DIM FLAG:INTEGER
□LOOKUP="+-*/^<>"
1 FLAG=0 \CHAR=""
□PRINT "Enter the first number to evaluate...";
□INPUT FIRST
□IF FIRST=0 THEN
□GOTO 100
□ENDIF
□PRINT "Enter the second number to evaluate...";
□INPUT SECOND
□PRINT "Press the key of the operator you want to
use..."
□PRINT " + - * / ^ < > ...";
□WHILE CHAR="" DO
□RUN INKEY(CHAR)
□ENDWHILE
□PRINT
□FLAG=SUBSTR(CHAR,LOOKUP)
□ON FLAG GOTO 10,20,30,40,50,60,70
10 PRINT FIRST+SECOND \ GOTO 1
20 PRINT FIRST-SECOND \ GOTO 1
30 PRINT FIRST*SECOND \ GOTO 1
40 PRINT FIRST/SECOND \ GOTO 1
50 PRINT FIRST^SECOND \ GOTO 1
60 PRINT FIRST<SECOND \ GOTO 1
70 PRINT FIRST>SECOND \ GOTO 1
100 PRINT "Procedure Terminated Due to 0
Input..."
□END
```

INPUT Get data from a device path

Syntax: **INPUT [*#path,*] [*prompt,*] *variable* [*,variable...*]**

Function: INPUT accepts input from the specified path. (The default is the keyboard.) When a procedure encounters INPUT, it displays a question mark and awaits data from the specified path. If you provide a string type prompt for INPUT, it displays the text of the prompt, rather than a question mark.

INPUT stores the data it collects in the variable you specify. The type of the receiving variable must match the type of data received.

Because INPUT sends data (the question mark prompt or the user-specified string prompt), it is really both an input and an output statement. This means that, if you use a path other than the standard input path, you should not use the UPDATE mode. If you do, the prompts produced by INPUT write to the file specified by the path number.

If the data received does not match the type of data INPUT expects, it displays the message:

```
**INPUT ERROR - RETYPE**
```

followed by a new prompt. You must then enter the entire input line, of the correct type, to satisfy INPUT. For more information, see GET.

Parameters:

path — Either a variable containing the path number, or the absolute path number to the file or device from which you want to receive input. If you want to receive input from the keyboard, do not include a path number.

prompt — Text you type as a message to be displayed when BASIC09 executes an INPUT statement.

variable — The variable name in which you want to store the data received by INPUT. The type of variable must match the type of input.

Examples:

```
INPUT NUMBER,NAME$,LOCATION

INPUT #PATH,X,Y,Z

INPUT "What is your selection: ";CHOICE

INPUT #HOST,"What's your ID number? ",IDNUM
```

Sample Program:

This procedure calculates the day of the week for a specified date. It asks you for the date using the INPUT command.

```
PROCEDURE weekday
□DIM X,Y,D,M,CALC:INTEGER; DAY,MONTH:STRING[2];
YEAR:STRING[4]; WEEKDAY (7):STRING[9]
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,
INUM:INTEGER
□PRINT USING "S80 ","Day of the Week Program","For
any year after 1752"
□PRINT
□PRINT "Enter day (e.g. 08): ";  \ INPUT DAY
□PRINT "  Enter month (e.g. 12): ";  \ INPUT MONTH
□PRINT "  Enter year (e.g. 1986): ";  \ INPUT YEAR
□Y=VAL(YEAR) \M=VAL(MONTH) \D=VAL(DAY)
□FOR X=1 TO 7
□READ WEEKDAY(X)
□NEXT X
□ANUM=INT(.6+1/M)
□BNUM=Y-ANUM
```

```
CNUM=M+12*ANUM
DNUM=BNUM/100
ENUM=INT(DNUM/4)
FNUM=INT(DNUM)
GNUM=INT(5*BNUM/4)
HNUM=INT(13*(CNUM+1)/5)
INUM=HNUM+GNUM-FNUM+ENUM+D-1
INUM=INUM-7*INT(INUM/7)+1
PRINT
PRINT "The day of the week on "; M; "/"; D;
"/"; Y; " is..."; WEEKDAY (INUM)
DATA "Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday"
END
```

INT Convert real number to whole number

Syntax: **INT(*value*)**

Function: Converts a real number to a whole number by truncating any fractional part of the real number.

Parameters:

value	Any negative or positive real number.

Examples:

```
PRINT INT(77.89)

PRINT INT(NUM)

PRINT INT(-8.12)
```

Sample Program:

The RND function produces real numbers. This procedure uses INT to convert the real RND output to integer values.

```
PROCEDURE integer
□DIM T:INTEGER
□FOR T=1 TO 10
□R=RND(50)-25
□PRINT R,INT(R)
□NEXT T
□END
```

KILL Remove a procedure from memory

Syntax: **KILL *procedure***

Function: Unlinks (removes) an external procedure from the BASIC09 procedure directory. If the procedure is not external, but resides in BASIC09's workspace, KILL has no effect.

Use KILL to remove auto-loaded (packed) procedures that are called by RUN or CHAIN. You can also use KILL with auto-loading procedures as a method to overlay programs within BASIC09.

> **Warning:** Be certain you do not KILL an active procedure. Also be certain that when you use RUN and KILL together, that both statements use the same string variable that contains the name of the procedure to RUN and KILL.

Parameters:

procedure — The name of the external procedure you want to KILL. *Procedure* can either be a name or a variable containing the procedure name.

Examples:

```
PROCEDURENAME$ = "AVERAGE"
RUN PROCEDURENAME$
KILL PROCEDURENAME$

INPUT "Which test do you want to run? ",TEST$
RUN TEST$
KILL TEST$
```

Sample Program:

This procedure calls a procedure named Show to display ASCII values on the screen. When it no longer needs the Show procedure, it removes Show from memory using KILL.

```
PROCEDURE produce
□DIM T,U:INTEGER
□DIM NUM,NUM1,NUM2,TABLE,PROCNAME:STRING
□PROCNAME=SHOW
□TABLE="123456789ABCDEF"
□FOR T=8 TO 15
□FOR U=1 TO 15
□NUM1=MID$(TABLE,T,1)
□NUM2=MID$(TABLE,U,1)
□NUM=NUM1+NUM2   (* parameter to pass to Show.
□RUN PROCNAME(NUM)
□NEXT U
□NEXT T
□KILL "PROCNAME" (* remove Show from the workspace.
□END

PROCEDURE SHOW
□PARAM NUM:STRING
□SHELL "DISPLAY "+NUM
□END
```

LAND Returns the logical AND of two numbers

Syntax: **LAND(*num1,num2*)**

Function: Performs the logical AND function on a byte- or integer-type value. The operation involves a bit-by-bit logical AND of the two numbers you specify. For instance, if you LAND 5 and 6, the logic is like this:

Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

```
      0101
AND   0110
-----------
=     0100   = 4 Decimal
```

Parameters:

num1	A byte- or integer-type number.
num2	A byte- or integer-type number.

Examples:

```
PRINT LAND(11,12)

PRINT LAND($20,$FF)
```

Sample Program:

The following procedure asks eight questions and uses the eight bits of one byte (contained in the variable STORAGE) to indicate either a "yes" or "no" answer. If the answer is "yes," it sets a corresponding bit to 1. If the answer is "no," it sets a corresponding bit to 0, using LAND. This procedure operates in conjunction with the sample program for LXOR.

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER; X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X) (* OR STORAGE if yes.
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X)) (* LAND STORAGE with NOT value if no.
□ENDIF
□X=X*2
□NEXT T
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word processing"
□DATA "Do you use your Color Computer for business applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the office"
□DATA "Do you use your Color Computer more than two hours a day"
□DATA "Do you share your Color Computer with others"
```

LEFT Returns characters from the left portion of a string

Syntax: **LEFT$(*string,length*)**

Function: Returns the specified number of characters from the specified string, beginning at the leftmost character. If *length* is the same as or greater than the number of characters in *string*, then LEFT$ returns all the characters in the string.

Parameters:

string	A sequence of ASCII characters or a string variable name.
length	The number of characters you want to access.

Examples:

```
PRINT LEFT$("HOTDOG",3)

PRINT LEFT$(A$,6)
```

Sample Program:

The following procedure extracts the first name from a list of ten names with the LEFT$ function.

```
PROCEDURE firstname
□DIM NAMES:STRING; FIRSTNAME:STRING[10]
□PRINT "Here are the first names:"
□FOR T=1 TO 10
□READ NAMES
□POINTER=SUBSTR(" ",NAMES) (* find space between first and last names.
□FIRSTNAME=LEFT$(NAMES,POINTER-1) (* extract first name.
□PRINT FIRSTNAME (* print first name.
□NEXT T
□END
□DATA "Joe Blonski","Mike Marvel","Hal Skeemish","Fred Laungly"
□DATA "Jane Mistey","Wendy Paston","Martha Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
```

LEN Returns the length of a string

Syntax: **LEN(*string*)**

Function: Returns the number of characters in a string. Counts blanks or spaces as characters.

Parameters:

string	A literal string or a variable containing string characters.

Examples:

```
PRINT LEN("ABCDEFGHIJKLM")

PRINT LEN(NAME$)

NAME$ = "JOE"
ADDRESS$ = "2244 LANCASTER"
TOTALLEN = LEN(NAME$)+LEN(ADDRESS$)
```

Sample Program:

The following procedure uses LEN to determine which name in a list is longest.

```
PROCEDURE longname
□DIM NAMES,LNAME:STRING; LONGEST,LENGTH:INTEGER
□NAMES="" \LNAME="" \LENGTH=0 \LONGEST=0
□FOR T=1 TO 10
□READ NAMES
□LENGTH=LEN(NAMES)
□IF LONGEST<LENGTH THEN
□LONGEST=LENGTH
□LNAME=NAMES
□ENDIF
□NEXT T
□PRINT "The longest name is "; LNAME; " with "; LONGEST; " characters."
□END
□DATA "Joe Blonski","Mike Marvel","Hal Skeemish","Fred Laungly"
□DATA "Jane Misty","Wendy Paston","Martha Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
```

LET Assigns a variable's value

Syntax: [LET] ***variable*** **=** ***expression***

Function: Assigns a value to a variable. BASIC09 does not require the LET statement to assign values but does accept it in order to be compatible with versions of BASIC that do require it.

Parameters:

variable	The variable to which you want to assign a value.
expression	Either a numeric or string constant or a numeric or string expression.

Notes:

- The result of the LET expression must be of the same type as, or compatible with, the variable in which it is stored.
- BASIC09's assignment function accepts either = or := as assignment operators. The := form helps to distinguish assignment operations from comparisons (test for equality) and is compatible with Pascal programming.
- Use BASIC09's assignment function to copy entire arrays or complex data structures to another array or complex data structure. The data structures do not need to be of the same type or *shape*, but the size of the destination structure must be the same as or larger than the source structure. This means the assignment function can perform unusual type conversions. For example, you can copy a string variable of 80 characters into a one-dimensional array of 80 bytes.

Examples:

```
LET A = 5

LET A := B

ANSWER = A * B

LET NAME$ := "JOE"

NAME $ = FIRSTNAME$ + " " + LASTNAME$
```

Sample Program:

This procedure uses LET to assign a random value to the variable R.

```
PROCEDURE getint
□DIM T:INTEGER
□FOR T=1 TO 10
□LET R=RND(50)-25
□PRINT R,INT(R)
□NEXT T
□END
```

LNOT Performs a logical NOT on a number

Syntax: LNOT(*value*)

Function: Performs the logical NOT function on an integer or byte type number. The operation involves a bit-by-bit logical complement operation of the number you specify. For instance, if *value* is 188, the logic looks like this:

188 Decimal = 10111100 Binary

NOT 10111100
= 01000011

01000011 Binary = 67 Decimal

LNOT changes each bit in a binary number to its complementary binary value—all 1 values become 0 and all 0 values become 1. LNOT returns an integer result; it is not a Boolean operator.

Parameters:

value	Any decimal or hexadecimal integer or byte number. Precede hexadecimal numbers with $.

Examples:

```
PRINT LNOT(88)

A = LNOT(B)

A = LNOT($44)
```

Sample Program:

This procedure uses one byte (contained in the variable STORAGE) to indicate the results of eight questions. Each bit in the byte indicates a Yes or No answer (Yes = 1 and No = 0). The combination logic of LAND and LNOT masks the byte X so that it affects only the appropriate bit of STORAGE to set it to 0 if the answer is No. LOR sets the appropriate bit to 1 if the answer is Yes. The procedure operates in conjunction with the LXOR sample program.

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER; X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X) (* Answer is yes, set bit to 1.
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X)) (* Answer is no, set bit to 0.
□ENDIF
□X=X*2
□NEXT T
□PRINT STORAGE
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word processing"
□DATA "Do you use your Color Computer for business applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the office"
□DATA "Do you use your Color Computer more than two hours a day"
□DATA "Do you share your Color Computer with others"
```

LOG Returns natural logarithm

Syntax: **LOG(*number*)**

Function: Computes the natural logarithm of a number that is greater than zero. BASIC09 returns the logarithm as a real type result.

Parameters:

number	Any integer, byte, or real number.

Examples:

```
PRINT LOG(3.14159)

LOGVALUE = LOG(88/PI)
```

Sample Program:

This procedure calculates the natural log and the log to base 10 of the values 1-7.

```
PROCEDURE logs
□DIM NUM,T:INTEGER
□FOR T=1 TO 7
□PRINT "The LOG of "; T; " to the natural base =
"; LOG(T)
□PRINT "The LOG of "; T; " to base 10 = ";
LOG10(T)
□PRINT
□NEXT T
□END
```

LOG10 Returns base 10 logarithm

Syntax: LOG10(*number*)

Function: Calculates the base 10 logarithm of a number. BASIC09 returns the logarithm as a real number.

Parameters:

number Any byte, integer, or real value.

Examples:

```
PRINT LOG10($45)

PRINT LOG10(A)

PRINT LOG10(A/12)
```

Sample Program:

This procedure calculates the natural log and the log to base 10 of the values 1-7.

```
PROCEDURE logs
□DIM NUM,T:INTEGER
□FOR T=1 TO 7
□PRINT "The LOG of "; T; " to the natural base =
"; LOG(T)
□PRINT "The LOG of "; T; " to base 10 = ";
LOG10(T)
□PRINT
□NEXT T
□END
```

LOOP/ENDLOOP

Establishes/Closes a loop

Syntax: **LOOP**
statement(s)
ENDLOOP

Function: Establishes a loop in which you can install EXITIF tests at any location. The LOOP and ENDLOOP statements define the body of the loop. EXITIF tests for a condition which, if TRUE, causes alternate actions, the transfer of procedure execution to another routine, or both.

If you do not include an EXITIF statement, the loop cannot terminate.

Parameters:

statement(s) One or more procedure lines to execute within the loop.

Examples:

```
LOOP
COUNT = COUNT+1
EXITIF COUNT > 100 THEN
DONE = TRUE
ENDEXIT
PRINT COUNT
X=COUNT/2
ENDLOOP
```

```
INPUT X,Y
LOOP
PRINT
EXITIF X<0 THEN
PRINT "X became 0 first"
END
ENDEXIT
X = X-1
EXITIF Y=0 THEN
PRINT "Y became 0 first"
```

```
END
ENDEXIT
Y=Y-1
ENDLOOP
```

Sample Program:

This procedure simulates a gambling machine that awards cash returns depending on a random selection of kinds of fruits. You begin with a stake of $25 and win or lose according to random selections of the procedure.

The program uses LOOP/ENDLOOP to keep operating until you run out of cash.

```
PROCEDURE bandit
□DIM FRUIT1,FRUIT2,FRUIT3,STAKE:INTEGER;
FRUIT(10):STRING[6]
□STAKE=25
□PRINT  \ PRINT "You have $"; STAKE; " to play
with."
□FOR T=1 TO 10
□READ FRUIT(T)
□NEXT T
□LOOP
□FRUIT1=RND(9)+1 \FRUIT2=RND(9)+1 \FRUIT3=RND(9)+1
□PRINT FRUIT(FRUIT1); " "; FRUIT(FRUIT2); " ";
FRUIT(FRUIT3)
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) AND FRUIT(FRUIT1)=
FRUIT(FRUIT3) THEN
□STAKE=STAKE+10
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) OR FRUIT(FRUIT2)=
FRUIT(FRUIT3) THEN
□STAKE=STAKE+2
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT3) THEN
□STAKE=STAKE+1
□ELSE STAKE=STAKE-1
□ENDIF
□ENDIF
□ENDIF
□EXITIF STAKE<1 THEN
□PRINT
□PRINT "You're Busted...Better go home."
```

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER;
X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X)
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X))
□ENDIF
□X=X*2
□NEXT T
□PRINT STORAGE
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word
processing"
□DATA "Do you use your Color Computer for business
applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the
office"
□DATA "Do you use your Color Computer more than
two hours a day"
□DATA "Do you share your Color Computer with
others"
```

LXOR Returns logical XOR of two numbers

Syntax: LXOR(*value1,value2*)

Function: Performs the logical XOR function on two-byte, or integer-type, values. For instance, if you LXOR the numbers 5 and 6 the logic is like this:

```
Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

        0101
LXOR    0110
-------------
=       0011   = 3 Decimal
```

If one bit or the other bit in the evaluation is 1, but not both, LXOR returns a result of 1. Otherwise, LXOR returns a result of 0.

Parameters:

value1	A byte or integer number.
value2	A byte or integer number.

Examples:

```
PRINT LXOR(11,12)

PRINT LXOR($20,$FF)
```

Sample Program:

The following program summarizes the results of the sample program for LOR. The LOR program stored the answers to eight questions in a single byte. This procedure reads the byte and displays appropriate comments. LXOR checks to see if two of the answers are "yes" or "no."

```
□ENDEXIT
□PRINT "Your stake is now $"; STAKE; "."
□PRINT
□PRINT
□INPUT "Press ENTER to pull again...",Z$
□ENDLOOP
□END
□DATA "ORANGE","APPLE","CHERRY","LEMON","BANANA"
□DATA "PEAR","PLUM","PEACH","GRAPE","APRICOT"
```

LOR Returns logical OR of two numbers

Syntax: **LOR(*value1,value2*)**

Function: Performs the logical OR function on a byte- or integer-type value. The operation involves a bit-by-bit logical OR operation on two values. For instance, if you LOR the numbers 5 and 6, the logic is like this:

Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

```
     0101
OR   0110
-----------
=    0111   = 7 Decimal
```

If one bit or the other bit is 1, LOR returns a result of 1. Otherwise, LOR returns a result of 0.

Parameters:

value1	A byte or integer number.
value2	A byte or integer number.

Examples:

```
PRINT LOR(11,12)

PRINT LOR($20,$FF)
```

Sample Program:

This procedure stores the answers to eight "yes" or "no" questions in one byte, named STORAGE. If you answer "yes" to a prompt, the procedure sets a corresponding bit to 1. If you answer "no" to a prompt, the procedure sets a corresponding bit to 0. The procedure uses LOR to set bits to 1 by *masking* all bits except the one it needs to set. The procedure operates in conjunction with the LXOR sample program.

```
PROCEDURE summary
□DIM T:INTEGER; A,B,X,TEST,TEST2:BYTE; SUMMARY:
STRING[50]
□PARAM STORAGE:BYTE
□A=0 \B=0
□PRINT  \ PRINT
□PRINT "The following is a summary of the
questionnaire answers:"
□PRINT
□PRINT "The surveyee: "
□X=1
□FOR T=1 TO 8
□TEST=LAND(STORAGE,X)
□READ SUMMARY
□IF TEST>0 THEN
□PRINT TAB(10); SUMMARY
□ENDIF
□X=X*2
□NEXT T
□IF LAND(STORAGE,128)>0 THEN
□A=1
□ENDIF
□IF LAND(STORAGE,64)>0 THEN
□B=1
□ENDIF
□TEST2=LXOR(A,B)
□IF TEST2=1 THEN
□PRINT "This computer owner either uses the
computer"
□PRINT "more than two hours a day or shares it
with others."
□PRINT "This is a heavy use situation."
□ENDIF
□TEST2=LAND(A,B)
□IF TEST2=1 THEN
□PRINT "This computer user uses the computer more
than two"
□PRINT "hours per day and shares it with others.
This is a"
□PRINT "super heavy use situation."
□ENDIF
□END
□DATA "Uses more than one computer"
□DATA "Plays games"
```

```
□DATA "Uses the computer for word processing"
□DATA "Uses the computer for business"
□DATA "Keeps a Color Computer at home"
□DATA "Keeps a Color Computer at the office"
□DATA "Uses the computer more than two hours a
day"
□DATA "Shares the computer with others"
```

MID$ Returns characters from within a string

Syntax: **MID$(*string,begin,length*)**

Function: Returns a substring *length* characters long, beginning at *begin*. Use MID$ to "take apart" a string consisting of a number of elements.

Parameters:

string	A sequence of string type characters or a string type variable.
begin	The position (an integer value) in *string* of the first character to retrieve.
length	The number of characters you want to retrieve.

Examples:

```
NAME$ = "JONES, JOHN M."
LASTNAME$ = MID$(NAME$,8,6)
FIRSTNAME$ = MID$(NAME$,1,5)
INITIAL$ = MID$(NAME$,15,2)
```

Sample Program:

This procedure reverses a word or phrase you type. MID$ reads each character in your phrase from the end to the beginning.

```
PROCEDURE reverse
□DIM PHRASE:STRING; T,BEGIN:INTEGER
□PRINT "Type a word or phrase you want to
reverse:";
□PRINT
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□END
```

MOD Returns modulus of a division

Syntax: **MOD(*number1,number2*)**

Function: Returns the modulus (remainder) of a division. MOD divides *number1* by *number2* and calculates the remainder. You can use MOD to put a limit on a numeric variable. For instance, regardless of the value of X, MOD(X,3) produces numbers only in the range 0 through 2. MOD(X,5) produces numbers only in the range of 0 through 4.

You can use MOD to cause repeating sequences. For instance, in a loop, MOD(X,3) produces a repeating sequence of 0, 1, 2, where X increases by 1 in each step of the loop.

Parameters:

number1	A byte, integer or real number dividend.
number2	A byte, integer or real number divisor.

Examples:

```
PRINT MOD(99,5)
```

Sample Program:

This procedure uses MOD to execute repeatedly routines that display asterisks on the screen. There are eight subroutines that the MOD function selects over and over through 100 passes.

```
PROCEDURE stardown
□DIM T:INTEGER
□SHELL "TMODE -PAUSE"
□FOR T=1 TO 100
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

NEXT Causes repetition in a FOR loop

Syntax: **FOR *variable* = *init val* TO *end val* [STEP *value*]**
[*procedure statements*]
NEXT *variable*

Function: NEXT forms the bottom end of a FOR/NEXT loop. Any program statements between FOR and NEXT are executed once for each repetition of the loop, from the initial value to end value.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to execute within the loop.

For more information, see FOR/NEXT/STEP.

NOT Returns the complement of a value

Syntax: **NOT(*value*)**

Function: Returns the logical complement of a Boolean value or expression.

Parameters:

value	A Boolean value (True or False), or an expression resulting in a Boolean value.

Examples:

```
DIM TEST:BOOLEAN
WHILE NOT(TEST) DO
A=A+1
TEST=A=B
ENDWHILE
```

Sample Program:

This procedure redirects the current directory listing to a file named Dirfile. It then opens Dirfile and reads the contents, displaying each line on the screen. It uses NOT in a WHILE/ENDWHILE loop to make sure that the end of the file has not been reached before trying to read another entry.

```
PROCEDURE readfile
□DIM A:STRING[80]
□DIM PATH:BYTE
□SHELL "DIR > dirfile"
□OPEN #PATH,"dirfile":READ
□WHILE NOT EOF(#PATH) DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

ON ERROR/GOTO

Establishes an error trap

Syntax: **ON ERROR [GOTO *linenum*]**

Function: Sets an error *trap* that transfers control to the specified line number in a procedure. This lets your program recover from an error and continue execution. To use these commands, your program must have at least one numbered line—the line to branch to in the event of an error.

Parameters:

linenum	The line to which you want BASIC09 to branch should an error occur.

Notes:

- ON ERROR GOTO is effective only with non-fatal, run-time errors. If such an error occurs without a preceding ON ERROR GOTO statement, BASIC09 enters the DEBUG mode. You must specify ON ERROR GOTO before an error occurs.
- You turn on error trapping by specifying ON ERROR GOTO *linenum*. You turn off error trapping by specifying ON ERROR without a line number.
- Use ON ERROR GOTO with the ERR function (that returns the code of the last error) to specify a particular action for a particular error. You can also use ERROR to simulate an error to test error trapping. For more information on this, see ERROR.

Examples:

```
□DIM FILENAME:STRING
□DIM PATH:INTEGER
10□INPUT "Name of file to create? ",FILENAME
□ON ERROR GOTO 100
□CREATE #PATH,FILENAME:UPDATE
□END
100□PRINT "That file already exists...please
choose another name..."
□GOTO 10
□END
```

Sample Program:

If you created a directory file with the GET sample program, you can use this procedure to delete files from the original directory using key characters. For instance, you might type XX as key characters. This means that any filename containing the character group XX is deleted. You can select any key characters you wish, but **be sure they apply only to files you want to delete.**

If you want to delete all the files in the directory, type an asterisk (*) when asked for key characters.

This procedure uses ON ERROR to let the procedure continue, even if a directory entry cannot be deleted—if an entry is a subdirectory. Without the ON ERROR function, the procedure would produce an error and cease execution when it tried to delete a subdirectory.

```
PROCEDURE purge
□REM Use caution with this procedure
□REM Be sure to specify key characters
□REM that exist only in the files you
□REM want to delete!

□DIM PATH:INTEGER
□DIM NAME(100):STRING
□DIM WILDCARD:STRING
□X=0
□OPEN #PATH,"dirfile":READ
□WHILE NOT(EOF(#PATH)) DO
□X=X+1
□READ #PATH,NAME(X)
```

```
□ENDWHILE
□FOR T=1 TO X
□PRINT NAME(T),
□NEXT T
□INPUT "Wildcard Characters...",WILDCARD
□FOR T=1 TO X
□ON ERROR GOTO 100
□IF SUBSTR(WILDCARD,NAME(T))>0 OR WILDCARD="*"
THEN
□PRINT "DELETING "; NAME(T); " ......"
□DELETE NAME(T)
□ENDIF
10□NEXT T
□END
100□PRINT "*□*□*□ERROR,□"; NAME(T); "□cannot be
deleted...continuing."
□GOTO 10
□END
```

ON/GOSUB Jumps to subroutine on a specified condition

Syntax: **ON *pos* GOSUB *linenum* [,*linenum*,...]**

Function: Transfers procedure control to the line number located at position *pos* in the list of line numbers immediately following the GOSUB command. For example, if *pos* equals 1, BASIC09 branches to the first line number it encounters in the list. If *pos* equals 2, BASIC09 branches to the second line number it encounters in the list. If *pos* is greater than the number of items in the list, execution continues with the next command line. To use ON/GOSUB you must have numbered lines to match the line numbers in your list. End the routines accessed by ON/GOSUB with a RETURN statement.

Parameters:

pos	An integer value pointing to a line number in a list of line numbers.
linenum	Any numbered line in the procedure.

Examples:

```
PRINT "You can now: (1) End the program (2) Print
the results"
PRINT "             (3) Try again       (4) Start
a new program"
INPUT "Type the letter of your choice: ",CHOICE
ON CHOICE GOSUB 100, 200, 300, 400
```

Sample Program:

This procedure uses MOD to execute repeatedly a sequence of GOSUB commands. A loop of index of 80 causes execution to jump to each line number in the list 10 times.

```
PROCEDURE repeat
□SHELL "TMODE -PAUSE"
□DIM T:INTEGER
□FOR T=1 TO 80
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

ON/GOTO Jump to line number on a specified condition

Syntax: ON *pos* GOTO *linenum* [,*linenum*,...]

Function: Transfers procedure control to the line number located at position *pos* in the list of line numbers immediately following the GOTO command. For example, if *pos* equals 1, BASIC09 branches to the first line number it encounters in the list. If *pos* equals 2, BASIC09 branches to the second line number it encounters in the list. If *pos* is greater than the number of items in the list, execution continues with the next command line. To use ON/GOTO you must have numbered lines to match the line numbers in the list.

Parameters:

pos	An integer value in a range from 1 to the number of items in the list following GOTO.
linenum	Any numbered line in the procedure.

Examples:

```
PRINT "You can now: (1) End the program (2) Print
the results"
PRINT "              (3) Try again        (4) Start
a new program"
INPUT "Type the letter of your choice: ",choice
ON CHOICE GOTO 100, 200, 300, 400
```

Sample Program:

This procedure converts decimal numbers to binary. It uses ON GOTO to execute the operation you select from a menu: Convert a number, display the result of all conversions, or end the program.

```
PROCEDURE bicalc
□DIM□NUMBER,NUM,X,STORAGE:INTEGER;□BI:STRING;
□ARRAY(50,2):STRING
□COUNT=0
```

```
10 BI="" \NUMBER=0 \NUM=0 \X=0 \STORAGE=0
 INPUT "Number to convert to binary ",NUMBER
 IF NUMBER=0 THEN END
 ENDIF
 NUM=LOG10(NUMBER)/.3
 NUM=2^NUM \STORAGE=NUMBER
 REPEAT
 X=NUMBER/NUM
 IF X>0 THEN BI=BI+"1"
 NUMBER=MOD(NUMBER,NUM)
 ELSE BI=BI+"0"
 ENDIF
 NUM=NUM/2
 UNTIL NUM<=1
 IF NUMBER>0 THEN
 BI=BI+"1"
 ELSE^BI=BI+"0"
 ENDIF
 PRINT STORAGE; " = "; BI; " in binary."
 PRINT
 COUNT=COUNT+1
 ARRAY(COUNT,1)=STR$(STORAGE)
 ARRAY(COUNT,2)=BI
12 PRINT "Do you want to: (1) Convert another
number."
 PRINT "                (2) Display all calculations
thus far."
 PRINT "                (3) End the program."
 INPUT "Enter 1, 2, or 3...",choice
 ON choice GOTO 10,20,30
 END
20 FOR T=1 TO COUNT
 PRINT ARRAY(T,1); " = "; ARRAY(T,2)
 NEXT T
 GOTO 12
30 PRINT  \ PRINT " Program Terminated"
 END
```

OPEN Opens a path to a device

Syntax: **OPEN** ***#path,"pathlist"*** **[*access mode*][+*access mode*][+...]**

Function: Opens an input/output path to a disk file or to a device. When you open a file, you can select one or more of the following access modes:

Mode	Function
READ	Lets you read (receive) data from a file or device but does not allow you to write (send) data.
WRITE	Lets you write data to a file or device but does not allow you to read data.
UPDATE	Lets you both read from and write to a file or device.
EXEC	Specifies that the file you want to access is in the current execution directory.
DIR	Specifies that the file you want to access is a directory-type file.

Parameters:

path	The variable in which BASIC09 stores the number of the newly opened path.
pathlist	The route to the file or device to be opened, including the filename if appropriate.
access mode	The type of access the system is to allow for the file or device. Use a plus symbol to specify more than one type of access.

Notes:

- The access mode defines the direction of I/O transfers.
- Because OS-9 files are byte-addressed and are unformatted, you can set up the filing system you want for a particular application. Your system can read the data contained in a file as single bytes or in groups of any size you want.
- You can expand a file using PRINT, WRITE, or PUT statements to write beyond the current end-of-file.

Examples:

```
OPEN #TRANS,"transportation":UPDATE

OPEN #SPOOL,"/user4/report":WRITE

OPEN #OUTPATH,name$:UPDATE+EXEC
```

Sample Program:

This procedure opens a path to both the SYS directory on Drive /D0 and the error message file.

```
PROCEDURE readerr
□DIM A:STRING[80]
□DIM PATH:BYTE
□OPEN #PATH,"/D0/SYS/ERRMSG":READ
□WHILE EOF(#PATH)<>TRUE DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

OR Performs a Boolean OR operation

Syntax: ***operand1*** **OR** ***operand2***

Function: Performs an OR operation on two or more values, returning a Boolean value of either TRUE or FALSE.

Parameters:

operand1 *operand2*	Either numeric or string values.

Examples:

```
PRINT A>3 OR B>3

PRINT A$="YES" or B$="YES"
```

Sample Program:

This procedure asks you to type a word or phrase, then converts all lowercase characters to uppercase. It uses OR to test for a character in your word or phrase that is outside of the ASCII values for lowercase letters. If it is, the character does not need converting.

```
PROCEDURE uppercase
□DIM PHRASE,NEWSTRING:STRING[80]; CHARACTER:
STRING[1]; T,X:INTEGER
□NEWSTRING="" \PHRASE=""
□PRINT "Type a phrase in lowercase and I will make
it uppercase."
□INPUT PHRASE
□FOR T=1 TO LEN(PHRASE)
□CHARACTER=MID$(PHRASE,T,1)
□X=ASC(CHARACTER)
□IF X<97 OR X>122 THEN
□NEWSTRING=NEWSTRING+CHARACTER
□ELSE
□X=X-32
□NEWSTRING=NEWSTRING+CHR$(X)
□ENDIF
□NEXT T
□PHRASE=NEWSTRING
□NEWSTRING=""
□PRINT PHRASE
□END
```

PARAM Establishes variables to receive from another procedure

Syntax: **PARAM *variable*[,...][:*type*][;*variable*][,...][:*type*] [...]**

Function: Defines the parameters that a *called* procedure expects to receive from the procedure that calls it. When using PARAM, be sure that the total size of each parameter in the calling procedure's RUN statement is the same as the defined size in the called procedure's PARAM statement.

Parameters:

variable	A simple variable, an array structure, or a complex data structure.
type	Byte, Integer, Real, Boolean, String, or user defined.

Notes:

- BASIC09 checks the size of each parameter to prevent accidental access to storage other than that assigned to the parameter. However, BASIC09 does not check that parameters are of the proper type. In most cases you must be sure that types evaluated in RUN statements match the types defined in the PARAM statements.

  However, because BASIC09 does not perform type checking, it is possible to perform useful but normally illegal type conversions of identically-sized data structures. For example, you could pass a string of 80 characters to a procedure expecting a byte array of 80 elements. Each character in the string is assigned a corresponding position in the array.

- You declare simple arrays by using the variable name, without a subscript, in a PARAM statement.

- You can declare several variables of the same type by separating them with commas. To separate variables of different types, follow each type group with a colon, the type name, and then a semicolon.
- If you do not include a maximum length for a string variable enclosed in brackets following the type, like this:

  ```
  DIM name:string[25]
  ```

  BASIC09 uses a default length of 32 characters for strings. You can declare shorter or longer lengths, to the capacity of BASIC09's memory.
- Arrays can have one, two, or three dimensions. The PARAM format for dimensioned arrays is the same as for simple variables except you must follow each array name with a subscript, enclosed in parentheses, to indicate its size. The maximum array size is 32767.

  Arrays can be either of the standard BASIC09 type, or of a user-defined type. To create your own data types for simple variables, arrays, and complex data structures, see TYPE.

Examples:

```
PARAM NUMBER:INTEGER

PARAM NAME:STRING[25];ADDRESS:STRING[30];ZIP:
INTEGER

PARAM NO1,NO2,NO3:REAL;NO4,NO5,NO6:INTEGER;NO7:
BYTE
```

Sample Program:

The first procedure asks you to enter a decimal number. Then, it asks you to choose whether you want to convert the number to binary or hexadecimal. Depending on your choice, the procedure calls (using RUN) either a procedure named Binary or a procedure named Hex. It passes the number you typed to the appropriate procedure for conversion.

```
PROCEDURE convert
□DIM NUMBER,CHOICE:INTEGER
□PRINT USING "S80^"; "Hexadecimal - Binary
Conversion Program"
□PRINT
10□INPUT "Number to convert...",NUMBER
□IF NUMBER=0 THEN
□END
□ENDIF
□INPUT "Choose: (1) Binary or (2) Hex...",CHOICE
□ON CHOICE GOTO 20,30
20□RUN BINARY(NUMBER)
□GOTO 10
30□RUN HEX(NUMBER)
□GOTO 10
□END

PROCEDURE binary
□DIM NUM,X,STORAGE:INTEGER; BI:STRING;
ARRAY(50,2):STRING
□PARAM NUMBER:INTEGER
□COUNT=0
□BI="" \NUM=0 \X=0 \STORAGE=0
□NUM=LOG10(NUMBER)/.3
□NUM=2^NUM \STORAGE=NUMBER
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN
□BI=BI+"1"
□NUMBER=MOD(NUMBER,NUM)
□ELSE
□BI=BI+"0"
□ENDIF
□NUM=NUM/2
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□BI=BI+"1"
□ELSE
□BI=BI+"0"
□ENDIF
□PRINT STORAGE; " = "; BI; " in binary."
□PRINT
□END
```

```
PROCEDURE hex
□DIM NUM,X,STORAGE:INTEGER; TABLE,HX:STRING;
ARRAY(50,2):STRING
□PARAM NUMBER:INTEGER
□TABLE="123456789ABCDEF"
□HX="" \NUM=0 \X=0 \STORAGE=0
□NUM=LOG10(NUMBER)/1.2
□NUM=16^NUM \STORAGE=NUMBER
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN
□HX=HX+MID$(TABLE,X,1)
□NUMBER=MOD(NUMBER,NUM)
□ELSE HX=HX+"0"
□ENDIF
□NUM=NUM/16
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□HX=HX+MID$(TABLE,NUMBER,1)
□ELSE
□HX=HX+"0"
□ENDIF
□PRINT STORAGE; " = "; HX; " in hexadecimal."
□PRINT
□END
```

PAUSE Suspends execution and enters Debug

Syntax: **PAUSE** ***text***

Function: Suspends the execution of a procedure and causes BASIC09 to enter the DEBUG mode. If you include text with the PAUSE command, it is displayed on the screen.

Place PAUSE statements in a program temporarily to observe the way in which the procedure operates and to track down programming errors. When the procedure is operating correctly, remove the PAUSE statement.

After using DEBUG, you can continue execution of the *paused* procedure with the CONT command.

Parameters:

text	A message you want PAUSE to display on the screen when BASIC09 executes the statement.

Examples:

```
PAUSE

PAUSE The array is now full.
```

PEEK Returns the value in a memory location

Syntax: **PEEK(*mem*)**

Function: Returns the value of a memory byte as a decimal integer. The value returned is in the range 0 to 255. PEEK is the complement of the POKE statement.

See also ADDR.

Parameters:

mem	An integer value representing the location of the memory byte you want to examine. The memory byte is relative to the current process's address space.

Examples:

```
PRINT PEEK(15250)

MEMVAL = PEEK(4450)
```

Sample Program:

This procedure asks you to type a phrase in uppercase characters. It then uses ADDR to locate the area in memory where BASIC09 stores the phrase. Next, it reads each character from memory with PEEK, converts it to lowercase if necessary, and pokes the new value back into the same location. When the procedure displays the contents of the phrase, it is all lowercase.

```
PROCEDURE lowercase
□DIM LOC,T:INTEGER; PHRASE:STRING[80]
□PRINT "Type a phrase in UPPERCASE and I'll make
it lowercase."
□INPUT PHRASE
□LOC=ADDR(PHRASE)
□FOR T=LOC TO LOC+LEN(PHRASE)
□X=PEEK(T)
□IF X>32 AND X<91 THEN
□X=X+32
□POKE T,X
□ENDIF
□NEXT T
□PRINT PHRASE
□END
```

PI Returns the value of pi

Syntax: PI

Function: Returns the constant value 3.14159265.

Parameters: None

Examples:

```
PRINT "The area of a circle with a radius of 6
inches is ";PI*6^2
```

Sample Program:

This procedure uses the formula (PI+2)/15 as a basis for calculating a screen position. Taking the sine of the formula, it prints a sine wave of asterisks down the screen.

```
PROCEDURE picalc
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 100
□CALCULATE=CALCULATE+FORMULA
□POSITION=INT(SIN(CALCULATE)*10+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

POKE Stores a value in a memory location

Syntax: **POKE** ***mem,value***

Function: Stores a value at the specified memory address, relative to the current process's address space. *Mem* is an absolute address at which BASIC09 stores a byte type value. POKE is the complement of the PEEK statement.

You should use care when using POKE. Because it changes the value in memory, a POKE to the wrong portion of memory could cause OS-9, BASIC09, or your procedures to malfunction until you reboot the system.

See also ADDR.

Parameters:

mem	An integer value representing the location of the memory byte you want to change.
value	The value to store in the specified memory location.

Examples:

```
POKE 15250,13
```

Sample Program:

This procedure asks you to type a phrase in uppercase characters. It then uses ADDR to locate the area in memory where BASIC09 stores the phrase. Next, it reads each character from memory, converts it to lowercase if necessary, and uses POKE to store the new value back in the same location. When the procedure next displays the contents of the phrase, it is all lowercase.

```
PROCEDURE lowercase
□DIM LOC,T:INTEGER; PHRASE:STRING[80]
□PRINT "Type a phrase in UPPERCASE and I'll make
it lowercase."
□INPUT PHRASE
□LOC=ADDR(PHRASE)
□FOR T=LOC TO LOC+LEN(PHRASE)
□X=PEEK(T)
□IF X>32 AND X<91 THEN
□X=X+32
□POKE T,X
□ENDIF
□NEXT T
□PRINT PHRASE
□END
```

POS Returns cursor's column position

Syntax: **POS**

Function: Returns the current column position of the cursor.

Parameters: None

Examples:

```
PRINT POS
```

Sample Program:

This procedure is a simple typing program that uses POS to make sure that words are not split when you type to the end of the screen. After you type 25 characters on a line, the procedure breaks the line at the next space character.

```
PROCEDURE wordwrap
□DIM CHARACTER:STRING[1]
□PRINT USING "S32^"; "Word Wrap Program"
□PRINT USING "S32^"; "Press [CTRL][C] to Exit"
□PRINT
□SHELL "TMODE -ECHO"
□WHILE CHARACTER<>" " DO
□GET #1,CHARACTER
□PRINT CHARACTER;
□IF POS>25 AND CHARACTER=" " THEN
□PRINT CHR$(13)
□ENDIF
□ENDWHILE
□SHELL "TMODE ECHO"
□END
```

PRINT Displays text

Syntax: **PRINT [*#path*] [TAB(*pos*);] *data*[;*data*...]**

Function: Prints numeric or string data on the video display unless another path is specified.

Parameters:

path	The number corresponding to an opened device or file. If you do not specify *path*, the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
pos	A column number that tells TAB where to begin printing. Specify any number from 0 to the width of your video display.
data	Any numeric or string constant or variable. Enclose string constants within quotation marks. All data items must be separated by a semicolon or comma.

Notes:

- If you specify more than one data item in the statement, separate them with commas or semicolons.
- If you use commas, PRINT automatically advances to the next tab *zone* before printing the next item. In BASIC09, tab zones are 16 characters apart.
- If you use semicolons or spaces to separate data items, BASIC09 prints the items without any spaces between them. BASIC09 begins the next print item immediately following the end of the last print item.
- If you end a print item without any trailing punctuation, PRINT begins printing at the beginning of the next line.

- If the data being printed is longer than the display screen width, PRINT moves to the next line and continues printing the data.
- TAB causes BASIC09 to begin displaying the specified data at the column position specified by TAB. If the output line is already past the specified TAB position, PRINT ignores TAB.
- You can concatenate items for printing using the plus (+) symbol, for example: `print "hello "+name$+" "+lastname$.`
- PRINT displays REAL numbers with nine or fewer digits in regular format. It displays REAL numbers with more than nine digits in exponential format. For example, 1073741824 is displayed as `1.07374182E+09`.
- You must enclose string constants within quotation marks.

Examples:

```
PRINT A$

PRINT "Menu Items"

PRINT COUNT

PRINT VALUE,TEMP+(n/2.5),LOCATION$

PRINT #PRINTER_PATH,"The result is ";NUMBER

PRINT #OUTPATH FMT$,COUNT,VALUE

PRINT "what is"+NAME$+"'s age? ";

PRINT "INDEX: ";I;TAB(25);"VALUE:  ";VALUE
```

Sample Program:

This procedure asks you to type a word or phrase, then displays it backwards by reading each character from end to beginning and using PRINT to display it on the screen.

```
PROCEDURE reverse
□DIM PHRASE,TITLE:STRING; T,BEGIN:INTEGER
□DIM INSTRUCTIONS:STRING[43]
□TITLE="Word Reversing Program"
□INSTRUCTIONS="Type a word or phrase you want to
reverse: "
□PRINT TITLE
□PRINT "_____"
□WHILE PHRASE<>"" DO
□PRINT
□PRINT INSTRUCTIONS
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□ENDWHILE
□END
```

PRINT USING Displays formatted text

Syntax: **PRINT [*#path*] USING [*format,*] *data*[;*data*...]**

Function: Prints data using a format you specify. This statement is especially useful for printing report headings, accounting reports, checks, or any document requiring a specific format. USING is actually an extension of the PRINT statement; therefore, the same rules that apply to the PRINT statement also apply to the PRINT USING statement (see PRINT).

Parameters:

path	The number corresponding to an opened device or file. If you do not specify *path*, the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
format	An expression specifying the arrangement of the displayed data.
data	Any numeric or string constant or variable. Always enclose string constants within quotation marks. Each data item must be separated by semicolons or commas.

Notes:

Each PRINT USING format specifier begins with a single *identifier* letter that specifies the type of format, as shown in the following table:

B	Boolean format
E	exponential format
H	hexadecimal format
I	integer format
R	real format
S	string format

Follow the identifier letter with a constant number that specifies the field width. This number indicates the exact number of print columns the output occupies. It must allow for both the data and any *overhead* characters, such as sign characters, decimal points, exponents, and so on.

Optionally, you can add a justification indicator to the format expression. The indicators are <, >, and ^. The meaning of these indicators varies, depending on the format type in which you use them. See the format type descriptions for specific information.

Note: Do not use any spaces within format expressions.

The following are the format type descriptions:

Real

Use this format for real, integer, or byte type numbers. The total field width specification must include two overhead positions for the sign and decimal point. The field width has two parts, separated by a period. The first part specifies the integer portion of the field. The second part specifies how many fractional digits to display to the right of the decimal point.

If a number has more significant digits than the field allows, BASIC09 uses the undisplayed digits to round the number within the correct field width.

The justification modes are:

- < Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces and sign.
- ^ Right justify with leading spaces and trailing sign (financial format).

Some examples and their results are:

```
PRINT USING "R8.2<",5678.123     5678.12
PRINT USING "R8.2>",5678.123     5678.12
PRINT USING "R8.2>",12.3           12.30
PRINT USING "R8.2>",-555.9       -555.90
PRINT USING "R10.2^",-6722.4599  6722.46-
```

Exponential

Use this format to display real, integer, or byte values in the scientific notation format—using a mantissa and decimal exponent. The field has two parts: the first part must allow for six overhead positions for the mantissa sign, decimal point, and exponent characters.

The justification modes are:

- `<` Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces and sign.

Some examples and their results are:

```
PRINT USING "E12.3",1234.567     1.235E+03
PRINT USING "E13.6>",-.001234  -1.234000E-03
PRINT USING "E18.5>",123456789        1.23457E+08
```

Integer

Use this format to display integer, byte, or real type numbers in an integer or byte format. The field width must allow for one position of overhead for the sign.

The justification modes are:

- `<` Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces and sign.
- `^` Right justify with leading sign and zeroes.

Some examples and their results are:

```
PRINT USING "I4<",10     10
PRINT USING "I4<",10      10
PRINT USING "I4^",-10   -010
```

Hexadecimal

Use this format to display any data type in hexadecimal notation. The field width specification determines the number of hexadecimal characters BASIC09 displays. If the data to display is string type, this function displays the ASCII value of each character in hexadecimal.

The justification modes are:

- `<` Left justify with trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces.
- `^` Center digits.

The number of bytes of memory used to represent data varies according to data type. The following chart suggests field widths for specific data types:

Type	Memory Bytes	Field Width To Specify
Boolean and Byte	1	2
Integer	2	4
Real	5	10
String	1 per character	2 times the string length

Some examples and their results are:

```
PRINT USING "H4",100      0064
PRINT USING "H4",-1       FFFF
PRINT USING "H8^","ABC"   414243
```

String

Use this format to display string data of any length. The field width specifies the total field size. If the string to display is shorter than the field size, PRINT USING pads it with spaces according to the justification mode. If the string to display is longer than the specified field width, PRINT USING truncates the right portion of the string.

The justification modes are:

- `<` Left justify with trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces.
- `^` Center characters.

Some examples and their results are:

```
PRINT USING "S9<","HELLO"    HELLO
PRINT USING "S9>","HELLO"        HELLO
PRINT USING "S9^","HELLO"      HELLO
```

Boolean

Use this format to display Boolean expression results. BASIC09 converts the result of the expression to the strings "True" or "False." The format and results are identical to STRING formats. The justification modes are:

- `<` Left justify with trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces.
- `^` Center characters.

If A=5 and B=6, some examples and their results are:

```
PRINT USING "B9<",A<B    True
PRINT USING "B9>",A>B        False
PRINT USING "B9^",A=B      False
```

Control Specifiers

You can also use *control specifiers* within PRINT USING formats. The three specifiers are:

T*n* Tab. *n* specifies a tab column at which to display the next data.

X*n* Spaces. *n* specifies a number of spaces to insert.

'text' Constant string. *text* is a string that is constant to the format.

An example and its result is:

```
PRINT USING "'Address',X1,H4,X4,'Data',X1,H2",
1000,100

Address 03E8     Data 64
```

Repeat

You can repeat identical sequences of specifications using parentheses within a format specification. Enclose the group of specifications you wish to repeat, preceded by a repetition count, such as:

"2(X2,r10.5)" in place of "X2,R10.5,X2,R10.5"

"2(I2,2(X1,S4))" in place of "I2,X1,S4,X1,S4,I2,X1,S4,X1,S4"

Sample Program:

This program looks at memory locations 32000 to 32010 and displays their contents in decimal, hexadecimal, and binary. PRINT USING formats the display in columns.

```
PROCEDURE memlook
□DIM NUMBER,T,MEM,VALUE:INTEGER
□DIM X,NUM:INTEGER; CHARACTER,BI:STRING
□PRINT "□Addr.□Dec.□Hex.□Bin□□□□□□ASCII"
□FOR Z=32000 TO 32010
□BI=""
□NUMBER=PEEK(Z)
□IF NUMBER>0 THEN
□GOSUB 100
□ENDIF
□IF PEEK(Z)<32 THEN
□CHARACTER=""
□ELSE
□CHARACTER=CHR$(PEEK(Z))
□ENDIF
□IF PEEK(Z)>0 THEN
□PRINT USING "I6<,T7,I4<,X2,H4<,X1,S8<,X2,S1",Z,
PEEK(Z),PEEK(Z),BI,CHARACTER
□ELSE PRINT USING "I6<,T7,I4<,X2,H4<,X1,S8>,X2,
S1",Z,0,0,"0000"," "
□ENDIF
□NEXT Z
□END
```

```
100 NUM=LOG10(NUMBER)/.3
 NUM=2^NUM
 REPEAT
 X=NUMBER/NUM
 IF X>0 THEN BI=BI+"1"
 NUMBER=MOD(NUMBER,NUM)
 ELSE BI=BI+"0"
 ENDIF
 NUM=NUM/2
 UNTIL NUM<=1
 IF NUMBER>0 THEN
 BI=BI+"1"
 ELSE BI=BI+"0"
 ENDIF
 RETURN
 END
```

PUT Writes to a direct access file

Syntax: **PUT *#path,data***

Function: Writes a fixed-size binary data record to a file or device. Use PUT to store data in random access files.

Although you usually use PUT with files, you can also use it to send data to a device.

For information about storing data in random access files, see Chapter 8, "Disk Files". Also, see GET, SEEK, and SIZE.

Parameters:

path	A variable name you chose to use in an OPEN or CREATE statement that stores the number of the path to the file or device to which you are directing data.
data	Either a variable containing the data you want to send or a string of data.

Examples:

```
PUT #PATH,DATA$

PUT INPUT,ARRAY$
```

Sample Program:

This procedure is a simple inventory data base. You type in the information for an item name, list cost, actual cost, and quantity. Using PUT, the procedure stores data in a file named Inventory.

```
PROCEDURE inventory
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL;
QTY:INTEGER
□DIM INV_ARRAY(100):INV_ITEM
□DIM WORK_REC:INV_ITEM
□DIM PATH:BYTE
□ON ERROR GOTO 10
```

```
□DELETE "inventory"
10□ON ERROR
□CREATE #PATH,"inventory"
□WORK_REC.NAME=""
□WORK_REC.LIST=0
□WORK_REC.COST=0
□WORK_REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK_REC
□NEXT N
□LOOP
□INPUT "Record number? ",recnum
□IF recnum<1 OR recnum>100 THEN
□PRINT
□PRINT "End of Session"
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK_REC.NAME
□INPUT "List price? ",WORK_REC.LIST
□INPUT "Cost price? ",WORK_REC.COST
□INPUT "Quantity? ",WORK_REC.QTY
□SEEK #PATH,(recnum-1)*SIZE(WORK_REC)
□PUT #PATH,WORK_REC
□ENDLOOP
□END
```

RAD Returns trigonometric calculations in radians

Syntax: **RAD**

Function: Set a procedure's *state flag* so that a procedure uses radians in SIN, COS, TAN, ACS, ASN, and ATN functions. Because this is the BASIC09 default, you do not need to use the RAD statement unless you previously used a DEG statement in the procedure.

Parameters: None

Examples:

```
RAD
```

Sample Program:

This program calculates sine, cosine, and tangent for a value you supply. It calculates one set of results in degrees, using DEG, and the second set of results in radians using RAD.

```
PROCEDURE trigcalc
□DIM ANGLE:REAL
□DEG
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "□□□□□□□□□Angle","SINE","COSINE","TAN"
□PRINT "□□□□□□□□□----------------------------------
-----------------"
□PRINT "Degrees = "; ANGLE,SIN(ANGLE),COS(ANGLE),
TAN(ANGLE)
□RAD
□PRINT "Radians = "; ANGLE,SIN(ANGLE),COS(ANGLE),
TAN(ANGLE)
□PRINT
□END
```

READ Reads data from a device or DATA statement

Syntax: **READ [*#path,*] *varname***

Function: Reads either an ASCII record from a sequential file or device, or an item from a DATA statement.

Parameters:

path	A variable containing the path number of the file you want to access. You can also specify one of the standard I/O paths (0, 1, or 2).
varname	The variable in which you want to store the data read from a file, device, or DATA line.

Notes:

The following information deals with reading sequential disk files:

- To read file records, you must first dimension a variable to contain the path number of the file, then use OPEN or CREATE to open a file in the READ or UPDATE access mode. The command begins reading records at the first record in the file. After it reads each item, it updates the pointer to the next item.
- Records can be of any length within a file. Make sure the variable you use to store the records is dimensioned large enough to store each item. If the variable storage is too small, BASIC09 truncates the record to the maximum size for which you dimensioned the variable. If you do not indicate a variable size with the DIM statement, the default is 32 characters.
- BASIC09 separates individual data items in the input record with ASCII null characters. You can also separate numeric items with comma or space character delimiters. Each input record is terminated by a carriage return character.

The following information deals with reading DATA items:

- READ accesses DATA line items sequentially. Each string type item in a DATA line must be surrounded by quotation marks. Items in a DATA line must be separated with commas.
- Each READ command copies an item into the specified variable storage and updates the data pointer to the next item, if any.
- You can independently move the pointer to a selected DATA statement. To do this, use line numbers with the DATA lines See the DATA and RESTORE commands for more information on using this function of READ.

Examples:

```
READ #PATH,DATA

READ #1,RESPONSE$

READ #INPUT,INDEX(X)

FOR T=1 TO 10
READ NAME$(T)
NEXT T
DATA "JIM","JOE","SUE","TINA","WENDY"
DATA "SALL","MICKIE","FRED","MARV","WINNIE"
```

Sample Program:

This procedure puts random values between 1 and 10 into a disk file, then READS the values and uses asterisks to indicate how many times RND selected each value.

```
PROCEDURE randlist
□DIM SHOW,BUCKET:STRING
□DIM T,PATH,SELECT(10),R:INTEGER
□BUCKET="************************"
□FOR T=1 TO 10
□SELECT(T)=0
□NEXT T
□ON ERROR GOTO 10
□SHELL "DEL RANDFILE"
10□ON ERROR
□CREATE #PATH,"randfile":UPDATE
□FOR T=1 TO 100
□R=RND(9)+1
□WRITE #PATH,R
□NEXT T
□PRINT "Random Distribution"
□SEEK #PATH,0
□FOR T=1 TO 100
□READ #PATH,NUM
□SELECT(NUM)=SELECT(NUM)+1
□NEXT T
□FOR T=1 TO 10
□SHOW=RIGHT$(BUCKET,SELECT(T))
□PRINT USING "S6<,I3<,S2<,S20<","Number",
T,":",SHOW
□NEXT T
□CLOSE #PATH
□END
```

REM Inserts remarks in a procedure

Syntax: **REM** [*text*]
(* [*text*][*)]

Function: Inserts remarks inside a procedure. BASIC09 ignores these remarks; they serve only to document a procedure and its functions. Use remarks to title a procedure, show its creation date, show the name of the programmer, or to explain particular features and operations of a procedure.

Parameters:

text	Comments you want to include within a procedure

Notes:

- You can insert remarks at any point in a procedure.
- The second form of REM, using parentheses and asterisks, is compatible with Pascal programming structure.
- When editing programs, you can use the exclamation character "!" in place of the keyword REM.
- BASIC09's initial compilation retains remarks, but the PACK compile command strips them from procedures.

Examples:

```
REM this is a comment

(* Insert text between parentheses and
asterisks*)

(* or use only one parenthesis and asterisk
```

Sample Program:

This procedure uses the various forms of REM to explain its operations.

```
PROCEDURE copydir
□REM Use this program with the
□(* GET sample program to *)
□(* create a file of directory*)
□(* filenames, then copy the*)
□(* files to another directory*)
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ (* open the file
□INPUT "Name of new directory...",DIRNAME (* get the directory
□SHELL "MAKDIR "+DIRNAME (* create a newdirectory
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE (* get a filename
□JOB=FILE+" "+DIRNAME+"/"+FILE (* create the COPY syntax
□ON ERROR GOTO 10
□PRINT "COPY "; JOB (* display the operation
□SHELL "COPY "+JOB (* copy the file
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

REPEAT/UNTIL

Establishes a loop/Terminates on specified condition

Syntax: **REPEAT**
procedure lines
UNTIL *expression*

Function: Establishes a loop that executes the encompassed procedure lines until the result of the expression following UNTIL is true. Because the loop is tested at the bottom, the lines within the loop are executed at least once.

Parameters:

expression	A Boolean expression (returns either True or False).
procedure lines	Statements you want to repeat until *expression* returns False.

Examples:

```
REPEAT
COUNT = COUNT+1
UNTIL COUNT > 100

INPUT X,Y
REPEAT
X = X-1
Y = Y-1
UNTIL X<1 OR Y<0
```

Sample Program:

The procedure sorts a disk file. In this case, it is written to sort the diskfile created by the GET sample program—a directory listing. It uses a REPEAT/UNTIL loop to compare a string in the file with the first string in the file. If the first string is greater than the comparison string, the procedure swaps them.

```
PROCEDURE dirsort
□DIM BTEMP:BOOLEAN; TEMP,FILES(125):STRING; TOP,
BOTTOM,M,N:INTEGER
□DIM T,X,PATH:INTEGER
□FOR T=1 TO 125
□FILES(T)=""
□NEXT T
□T=0
□OPEN #PATH,"dirfile":READ
□PRINT "LOADING:"
□WHILE NOT(EOF(#PATH)) DO
□T=T+1
□READ #PATH,FILES(T)
□ENDWHILE
□TOP=T
□BOTTOM=1
□PRINT "SORTING: ";
10□N=BOTTOM
□M=TOP
□PRINT ".";
□LOOP
□REPEAT
□BTEMP=FILES(N)<FILES(TOP)
□N=N+1
□UNTIL NOT(BTEMP)
□N=N-1
□EXITIF N=M THEN
□ENDEXIT

□TEMP=FILES(M)
□FILES(M)=FILES(N)
□FILES(N)=TEMP
□N=N+1
□EXITIF N=M THEN
□ENDEXIT
```

```
□ENDLOOP
□IF N<>TOP THEN
□IF FILES(N)<>FILES(TOP) THEN
□TEMP=FILES(N)
□FILES(N)=FILES(TOP)
□FILES(TOP)=TEMP
□ENDIF
□ENDIF

□IF BOTTOM<N-1 THEN
□TOP=N-1
□GOTO 10
□ENDIF
□IF N+1<TOP THEN
□BOTTOM=N+1
□GOTO 10
□ENDIF
□CLOSE #PATH
□DELETE "dirfile"
□CREATE #PATH,"dirfile":WRITE
□PRINT
□FOR Z=1 TO T
□WRITE #PATH,FILES(Z)
□PRINT FILES(Z),
□NEXT Z

□CLOSE #PATH
□END
```

RESTORE Resets READ pointer

Syntax: **RESTORE** ***linenumber***

Function: Sets the pointer for the READ command to the specified line number. RESTORE without a line number sets the data pointer to the first data statement in the procedure.

READ assigns the items in a DATA statement to variable storage. When you read an item, the pointer automatically advances to the next item. Using RESTORE you can skip backward or forward to data items at a specific line number.

Parameters:

linenumber	The line number of the DATA items you want READ to access next.

Examples:

```
RESTORE 100
```

Sample Program:

This procedure draws a box on the screen. It uses RESTORE to repeat the data in line 20 to create the sides of the box.

```
PROCEDURE box
□DIM LINE:STRING
□READ LINE
□PRINT LINE
□FOR T=1 TO 10
□RESTORE 20
□READ LINE
□PRINT LINE
□NEXT T
□RESTORE 10
□READ LINE
□PRINT LINE
10□DATA "--------------------------"
20□DATA "□□□□□□□□□□□□□□□□□□□□□□□□□"
□END
```

RETURN Returns from subroutine

Syntax: **RETURN**

Function: Returns procedure execution to the line immediately following the last GOSUB statement.

Every subroutine you access with GOSUB must contain a RETURN statement. You can call a subroutine in this manner as many times as you want.

Parameters: None

Sample Program:

This procedure draws a design of asterisks down the display screen. It uses MOD to send execution to a series of PRINT USING routines over and over. Each PRINT USING routine sends execution back to the main routine with a RETURN statement.

```
PROCEDURE stars
□DIM T:INTEGER
□SHELL "TMODE -PAUSE"
□FOR T=1 TO 100
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

RIGHT$ Returns specified rightmost portion of a string

Syntax: **RIGHT$(*string,length*)**

Function: Returns the specified number of characters from the right portion of the specified string. If *length* is the same as or greater than the number of characters in *string*, then RIGHT$ returns all of the characters in the string.

Parameters:

string	A sequence of string type characters or a variable containing a sequence of string type characters.
length	The number of characters you want to access.

Examples:

```
PRINT RIGHT$("HOTDOG",3)

PRINT RIGHT$(A$,6)
```

Sample Program:

```
PROCEDURE lastname
DIM NAMES:STRING; LASTNAME:STRING[10]
PRINT "Here are the last names:"
FOR T=1 TO 10
READ NAMES
POINTER=SUBSTR(" ",NAMES)
POINTER=LEN(NAMES)-POINTER
LASTNAME=RIGHT$(NAMES,POINTER)
PRINT LASTNAME
NEXT T
DATA "Joe Blonski","Mike Marvel","Hal Skeemish",
"Fred Langly"
DATA "Jane Misty","Wendy Paston","Martha
Upshong","Jacqueline Rivers"
DATA "Susy Reetmore","Wilson Creding"
END
```

RND Returns a random value

Syntax: **RND(*number*)**

Function: Returns a random real value in the following ranges:

If *number* = 0 then range = 0 to 1
If *number* > 0 then range = 0 to *number*

The values produced by RND are not truly random numbers, but occur in a predictable sequence. Specifying a number less than 0 begins the sequence over.

Parameters:

number	A numeric constant, variable, or expression.

Examples:

```
PRINT RND(5)

PRINT RND(A)

PRINT RND(A*5)
```

Sample Program:

This procedure presents addition problems for you to solve. It uses RND to select two numbers between 0 and 20.

```
PROCEDURE addition
□DIM A,B,ANSWER,C:INTEGER
□FOR T=1 TO 5
□A=RND(20)
□B=RND(20)
□C=A+B
□PRINT USING "'What is:□',I3>",A
□PRINT USING "'□□□□□□□+□',I3>",B
□PRINT "□□□□□□□□□------"
□INPUT "□□□□□□□□□",ANSWER
□IF ANSWER=C THEN
□PRINT "CORRECT"
□ELSE
□PRINT "WRONG"
□ENDIF
□PRINT
□NEXT T
□END
```

RUN Executes another procedure

Syntax: **RUN *procname*** [(***param***[,***param***,...])]

Function: Calls a procedure for execution, passing the specified parameters to the called procedure. When the called procedure ends, execution returns to the calling procedure, beginning at the statement following the RUN statement.

RUN can call a procedure existing within the workspace, a procedure previously compiled by the PACK command, or a machine language procedure outside the workspace.

Parameters:

procname	The name of the procedure to execute. The *procname* can be the literal name of the procedure to execute, or it can be a variable name containing the procedure name.
param	One or more parameters that the called program needs for execution. The parameters can be variables or constants, or the names of entire arrays or data structures.

Notes:

- You can pass all types of data to a called program except byte type. However, you can pass byte arrays.
- If a parameter is a constant or expression, BASIC09 passes it *by value*. That is, BASIC09 evaluates the constant or expression and places it in temporary storage. It passes the address of the temporary storage location to the called procedure. The called program can change the passed values, but the changes are not reflected in the calling procedure.
- If a parameter is the name of a variable, array, or data structure, BASIC09 passes it to the called program by *reference*. That is, it passes the address of the variable storage to the called procedure. Thus, the value can be changed by the receiving procedure, and these changes are reflected in the calling procedure.

- If the procedure named by RUN is not in the workspace, BASIC09 looks outside the workspace. If it cannot find it there, it looks in the current execution directory for a disk file with the proper name. If the file is on disk, BASIC09 loads and executes it, regardless of whether it is a packed BASIC09 program or a machine language program.

  If the program is a machine language module, BASIC09 executes a JSR (jump to subroutine) instruction to its entry point and executes it as 6809 native code. The machine language program returns to the original calling procedure by executing a RTS (return from subroutine) instruction.

- After you call an external procedure, and no longer need it, use KILL to remove it from memory to free space for other operations.

- Machine language modules return error status by setting the C bit of the MPU condition codes register, and by setting the B register to the appropriate error code.

Examples:

```
RUN CALCULATE(10,20,ADD)

RUN PRINT(TEXT$)
```

Sample Program:

Makelist creates and displays a list of fruit. Next, it asks you to type a word to insert. After you type and enter a new word, Makelist uses RUN to call a second procedure named Insert to look through the list and insert the new word in alphabetical order. After each insertion, the procedure asks for another word. Press only [ENTER] to terminate the program.

```
PROCEDURE makelist
□DIM LIST(25),NEWORD,TEMPWORD:STRING[15]
□DIM T,LAST:INTEGER
□LAST=10
□PRINT "This is your list..."
□FOR T=1 TO LAST
□READ LIST(T)
□PRINT LIST(T),
□NEXT T
□LOOP
```

```
□PRINT
□PRINT
□INPUT "Type a word to insert...",NEWORD
□EXITIF NEWORD="" THEN
□PRINT
□END "I've ended the session at your request..."
□ENDEXIT
□RUN Insert(LIST,NEWORD,LAST)
□PRINT
□PRINT "This is your new list..."
□FOR T=1 TO LAST
□PRINT LIST(T),
□NEXT T
□PRINT
□ENDLOOP
□DATA "APPLES","BANANAS","CANTALOUPE"
□DATA "DATES","GRAPES","LEMONS"
□DATA "MANGOS","PEACHES","PLUMS"
□DATA "PEARS"

PROCEDURE insert
□PARAM LIST(25),NEWORD:STRING[15]
□PARAM LAST:INTEGER
□DIM TEMPWORD:STRING[15]
□DIM T,X:INTEGER
□T=1
□WHILE NEWORD>LIST(T) DO
□T=T+1
□ENDWHILE
□FOR X=T TO LAST
□TEMPWORD=LIST(X)
□LIST(X)=NEWORD
□NEWORD=TEMPWORD
□NEXT X
□LAST=LAST+1
□LIST(LAST)=NEWORD
□END
```

SEEK Resets the direct-access file pointer

Syntax: **SEEK *#path,number***

Function: Changes the file pointer address in a disk file. The pointer indicates the location in a file for the next READ or WRITE operation.

You usually use SEEK with random access files to move the pointer from one record to another, in any order. You can also use SEEK with sequential access files to *rewind* the pointer to the beginning of the file (to the first item or record).

For information about storing data in random access files, see Chapter 8, "Disk Files." Also see PUT, GET, and SIZE.

Parameters:

path	A variable name you choose in which BASIC09 stores the number of the path it opens to the file you specify.
number	The item or record number you want to access. If you are rewinding a sequential access file, specify a *number* of 0.

Examples:

```
SEEK #PATH,0

SEEK #OUTFILE,A

SEEK #INDEX,LOCATION*SIZE(INVENTORY)
```

Sample Program:

This procedure creates a file named Test1, then writes 10 lines of data into it. Next, it reads the lines from the file and displays them. It uses SEEK to both store and extract the lines in blocks of 25 characters.

```
PROCEDURE makelines
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM PATH:BYTE
□LENGTH=25
□BASE 0
□ON ERROR GOTO 10
□DELETE "test1"
10□ON ERROR

□CREATE #PATH,"test1":WRITE

□FOR T=0 TO 9
□READ LINE$
□SEEK #PATH,LENGTH*T
□PUT #PATH,LINE$
□NEXT T
□CLOSE #PATH

□OPEN #PATH,"test1":READ
□FOR T=9 TO 0 STEP -1
□SEEK #PATH,LENGTH*T
□GET #PATH,LINE
□PRINT LINE
□NEXT T
□CLOSE #PATH
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #10"
```

SGN Returns a value's sign

Syntax: SGN(*number*)

Function: Determines whether a number's sign is positive or negative.

If *number* is less than 0, then SGN returns -1. If *number* equals 0, then SGN returns 0. If *number* if greater than 0, then SGN returns 1.

Parameters:

number	The value for which you want to determine the sign.

Examples:

```
PRINT SGN(-22)

PRINT SGN(A)

PRINT SGN(44-A)
```

Sample Program:

This procedure uses SGN to create half sine waves down the screen. SGN tests when the SIN calculation results are positive.

```
PROCEDURE halfsine
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 100
□CALCULATE=CALCULATE+FORMULA
□POSITION=INT(SIN(CALCULATE)*10+16)
□IF SGN(SIN(CALCULATE))>0 THEN
□PRINT TAB(POSITION); "*"
□ENDIF
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

SHELL Forks another shell

Syntax: **SHELL** [**"*string*"**][+ **"*string*"...**][+ ***variable***] [+ ***variable...***]

Function: Executes OS-9 commands or programs from within a BASIC09 procedure. Using SHELL, you can access OS-9 functions, including multiprogramming, utilities, commands, terminal and input/output control, and so on.

When you use the SHELL command, OS-9 creates a new process to handle the commands you provide. If you specify an operation, BASIC09 evaluates the expression and passes it to the shell for execution. If you do not specify an operation, BASIC09 temporarily halts, and the shell process displays prompts and accepts commands in the normal manner. In this case, press CTRL BREAK to return to BASIC09.

When the shell process terminates, BASIC09 becomes active and resumes execution at the statement following the SHELL statement.

Parameters:

string	Any OS-9 command or function. String constants must be enclosed in quotation marks. Concatenate string constants and string variables using a plus symbol (+).
variable	Any string variable containing an OS-9 command or function.

Examples:

```
SHELL "COPY FILE1 FILE2"

SHELL "COPY FILE1 FILE2&"

SHELL "COPY "+FILE$+" "+DIRNAME+"/"FILE$

SHELL "LIST DOCUMENT"

SHELL "KILL "+STR$(N)
```

Sample Program:

You must use this procedure with the GET sample program. Using the two programs together enables you to copy all the files from one directory to another directory. The GET sample program reads the files in a directory and stores them in a file named Dirfile. This procedure reads the filenames from Dirfile and uses SHELL to copy them to the directory you specify.

```
PROCEDURE copyutil
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ
□INPUT "Name of new directory...",DIRNAME
□SHELL "MAKDIR "+DIRNAME
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE
□JOB=FILE+" "+DIRNAME+"/"+FILE
□ON ERROR GOTO 10
□PRINT "COPY "; JOB
□SHELL "COPY "+JOB
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

SIN Returns the sine of a number

Syntax: **SIN(*number*)**

Function: Calculates the trigonometric sine of *number*. You can use the DEG or RAD commands to cause *number* to represent a value in either degrees or radians. Unless you specify DEG, the default is radians. SIN returns a real number.

Parameters:

number	The angle of two sides of a triangle for which you want to find the ratio.

Examples:

```
PRINT SIN(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent values for a number you type.

```
PROCEDURE ratiocalc
□DEG
□DIM ANGLE:REAL
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "Angle","SINE","COSINE","TAN"
□PRINT "----------------------------------------
--------------------"
□PRINT ANGLE,SIN(ANGLE),COS(ANGLE),TAN(ANGLE)
□PRINT
□END
```

SIZE Returns the size of a data structure

Syntax: **SIZE(*variable*)**

Function: Returns the size in bytes of a variable, array, or data structure. SIZE is especially useful with random access files to determine the size of records to store in a file. You can also use SIZE to determine the size of variable storage for other purposes.

SIZE returns the size of assigned storage, not necessarily the size of a string. For example, if you dimension a variable for 15 characters, and assign a 10-character string to it, SIZE returns 15, not 10. SIZE returns the total size of arrays. That is, it returns the number of elements multiplied by the size of the elements.

Parameters:

variable	The variable, array, or data structure for which you want to find the size.

Examples:

```
RECORDLENGTH = SIZE(A$)

PRINT "YOUR NAME IS STORED IN A "; SIZE(NAME$);
" CHARACTER STRING."
```

Sample Program:

This procedure creates a simple inventory, stored in a file named Inventory. It uses SIZE to calculate the size of each element to be stored in the file, and to move the pointer to the beginning of each record's storage space.

```
PROCEDURE inventory
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL;
QTY:INTEGER
□DIM INV_ARRAY(100):INV_ITEM
□DIM WORK_REC:INV_ITEM
□DIM PATH:BYTE
□ON ERROR GOTO 10
□DELETE "inventory"
10□ON ERROR
□CREATE #PATH,"inventory"
□WORK_REC.NAME=""
□WORK_REC.LIST=0
□WORK_REC.COST=0
□WORK_REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK_REC
□NEXT N
□LOOP
□INPUT "Record number? ",recnum
□IF recnum<1 OR recnum>100 THEN
□PRINT
□PRINT "End of Session"
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK_REC.NAME
□INPUT "List price? ",WORK_REC.LIST
□INPUT "Cost price? ",WORK_REC.COST
□INPUT "Quantity? ",WORK_REC.QTY
□SEEK #PATH,(recnum-1)*SIZE(WORK_REC)
□PUT #PATH,WORK_REC
□ENDLOOP
□END
```

SQ Returns the value of a number raised to the power of 2

Syntax: **SQ(*number*)**

Function: Calculates the value of a number raised to the power of 2.

Parameters:

number The number you want raised to the power of 2.

Examples:

```
PRINT SQ(188)

PRINT PI*SQ(R)
```

Sample Program:

This procedure uses SQ in a formula that positions asterisks on the screen in a sine wave pattern.

```
PROCEDURE sinedown
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 200
□CALCULATE=CALCULATE+SQ(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

SQR/SQRT Returns the square root of a number

Syntax: **SQR(*number*)**
SQRT(*number*)

Function: Calculates the square root of a number. SQR and SQRT serve the same function.

Parameters:

number The number for which you want the square root.

Examples:

```
PRINT SQR(188)

PRINT PI*SQRT(R)
```

Sample Program:

This procedure uses SQRT in a formula to position asterisks on the screen in a sine wave pattern.

```
PROCEDURE sqrdown
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=PI/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 200
□CALCULATE=CALCULATE+SQRT(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

STEP Establishes the size of increments in a FOR loop

Syntax:
FOR *variable* = *init val* TO *end val* [STEP *value*]
[*procedure statements*]
NEXT *variable*

Function: STEP provides an increment value in a FOR/NEXT loop. If you do not specify a STEP value, the loop steps in increments of 1.

BASIC09 executes the lines following the FOR statement until it encounters a NEXT statement. Then it either increases or decreases the initial value by 1 (the default) or by the value given by STEP. If you give the loop an initial value that is greater than the final value, and specify a negative value for STEP, the loop decreases from the initial value to the end value.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to be executed within the loop.

Examples:

```
FOR COUNTER = 1 to 100 STEP .5
PRINT COUNTER
NEXT COUNTER

FOR X = 10 TO 1 STEP -1
PRINT X
NEXT X

FOR TEST = A TO B STEP RATE
PRINT TEST
NEXT TEST
```

Sample Program:

This procedure reverses the order of characters in a word or phrase you type. It uses STEP to decrement a counter that points to each character in the string in reverse order.

```
PROCEDURE reverse
□DIM PHRASE:STRING; T,BEGIN:INTEGER
□PRINT "Type a word or phrase you want to
reverse:";
□PRINT
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□END
```

STOP Terminates a procedure

Syntax: **STOP [*"string"*]**

Function: Causes a procedure to cease execution, print the message "STOP Encountered", and return control to BASIC09's command mode. You can also specify additional text to display when BASIC09 encounters STOP.

Use stop when you want a procedure to terminate without entering the DEBUG mode.

Parameters:

string	Text to display when STOP executes.

Examples:

```
STOP "Program terminated before completion."

IF RESPONSE = "Y" THEN STOP "Program terminated
at your request."
ENDIF
```

STR$ Converts numeric data to string data

Syntax: **STR$(*number*)**

Function: Converts a numeric type to a string type. This lets you display the number as a string or use string operators on a number. The conversion replaces the numeric values with the ASCII characters of the number. STR$ is the inverse of the VAL function.

Parameters:

number Any numeric-type data.

Examples:

```
PRINT STR$(1010)

DIM I:INTEGER
I=44
PRINT STR$(I)

DIM B:BYTE
B=001
PRINT STR$(B)

DIM R:REAL
R=1234.56
PRINT STR$(R)
```

Sample Program:

This procedure calculates an exponential value, then adds the necessary number of zeroes to convert it to standard notation. It uses STR$ to convert the number you type to a string type value so that it can use string functions to add the zeroes.

```
PROCEDURE bignum
□DIM C,PLACES,B,SIGN:STRING; EX,COUNT,NEWCOUNT,
DECIMAL:INTEGER
□DIM NEW,ZERO,NEWEST:STRING[100]
□COUNT=-1
□ZERO="00000000000000000000000000000000000000"
□NEW="" \NEWEST=""
□INPUT "What number do you want to raise to the
power of 14?...", NUM
□A=NUM^14
□B=STR$(A)
□EX=SUBSTR("E",B)
□SIGN=MID$(B,EX+1,1)
□PLACES=RIGHT$(B,LEN(B)-EX-1)
□FOR T=1 TO LEN(B)
□C=MID$(B,T,1)
□IF C="." THEN
□DECIMAL=0
□GOTO 10
□ENDIF
□DECIMAL=DECIMAL+1
□IF C="E" THEN 100
□NEW=NEW+C
10□NEXT T
100□NEWCOUNT=VAL(PLACES)-DECIMAL
□NEW=NEW+LEFT$(ZERO,NEWCOUNT+1)
□FOR T=LEN(NEW) TO 1 STEP -1
□COUNT=COUNT+1
□NEWEST=MID$(NEW,T,1)+NEWEST
□IF MOD(COUNT,3)=2 AND T>1 THEN
□NEWEST=","+NEWEST
□ENDIF
□NEXT T
□PRINT NUM; " to the power of 14 = "; a
□PRINT "= "; NEWEST
□END
```

SUBSTR Searches for specified characters in a string

Syntax: **SUBSTR(*targetstring,searchstring*)**

Function: Searches for the first occurrence of *targetstring* within *searchstring* and returns the numeric value of its location. SUBSTR counts the first character in *searchstring* as character Number 1. Therefore, if you searched for the string "worth" in the string "Fortworth", SUBSTR returns a value of 5.

If SUBSTR cannot find *targetstring*, it returns a value of 0.

Parameters:

targetstring	The group of characters you want to locate.
searchstring	The string in which you want to find *targetstring*.

Examples:

```
PRINT SUBSTR("THREE","ONETWOTHREEFOURFIVESIX")

X=SUBSTR(" ",FULLNAME$)
```

Sample Program:

This procedure selects the last name from a string containing both a first name and a last name. It uses SUBSTR to find the space between the two names in order to determine where the last name begins.

```
PROCEDURE lastname
□DIM NAMES:STRING; LASTNAME:STRING[10]
□PRINT "Here are the last names:"
□FOR T=1 TO 10
□READ NAMES
□POINTER=SUBSTR(" ",NAMES)
□POINTER=LEN(NAMES)-POINTER
□LASTNAME=RIGHT$(NAMES,POINTER)
□PRINT LASTNAME
□NEXT T
□DATA "Joe Blonski","Mike Marvel","Hal
Skeemish","Fred Laungly"
□DATA "Jane Misty","Wendy Paston","Martha
Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
□END
```

SYSCALL Executes an OS-9 System Call

Syntax: **SYSCALL *callcode registers***

Function: Lets you execute any OS-9 system call from BASIC09. Use this command to directly manipulate your system or data or to directly access devices.

Be careful! Used improperly, SYSCALL can cause undesirable results—you might unintentionally format a disk or destroy disk or memory data. Before using SYSCALL, you should be familiar with assembly language programming and should understand the system call information in the *OS-9 Technical Reference* manual. The *OS-9 Technical Reference* manual provides information about all OS-9 system calls.

To pass required register values to the SYSCALL command, create a complex data structure that contains values for all registers. For example:

```
TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
DIM REGS:REGISTERS
DIM CALLCODE:BYTE
```

The complex data type REGISTERS contains values for all registers. Unless you specifically assign values to variables (for instance, REGS.CC, REGS.A, and REGS.B in the previous example), they contain random values. See the TYPE command for further information.

Parameters:

callcode	is the request code of the system call you wish to use. All system call codes are referenced in the *OS-9 Technical Reference* manual.
registers	is a list of the register entry values required for the system call you are using.

Examples: see "Sample Programs."

Sample Programs:

The following programs set up a data type (REGISTERS) for the register variables. Before executing SYSCALL, the procedures store the required register entry values in the complex data structure REGS. The procedures also establish CALLCODE as a variable to hold the request code of the system call you want to use.

The Writecall procedure uses the string variable TEST to store text that it writes to the screen through Path 0 (the standard output path) using System Call $8A, I$Write. Before the procedure calls I$Write, it stores the appropriate path number (0) in Register A. The ADDR command calculates the address of the variable TEST, and the LEN command determines the length of the variable. The procedure stores these two values in Registers X and Y.

The Readcall uses System Call $8B, I$ReadLn to perform a function that is the opposite of Writecall. Readcall establishes TEST as a string variable into which it writes the characters you type. Because the length of TEST is restricted to ten characters (`DIM TEST:STRING[10]`), the terminal bell sounds if you attempt to type more than 10 characters. Pressing **[ENTER]** terminates the call and the procedure prints the contents of TEST—the characters you typed.

```
PROCEDURE WriteCall
□TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
□DIM REGS:REGISTERS
□DIM PATH,CALLCODE:BYTE
□DIM TEST:STRING[50]
□TEST="This is a test of I$Write, System call
$8A..."
□REGS.A=0
□REGS.X=ADDR(TEST)
□REGS.Y=LEN(TEST)
□CALLCODE=$8A
□RUN SYSCALL(CALLCODE,REGS)
□PRINT
□END

PROCEDURE Readcall
□TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
□DIM REGS:REGISTERS
□DIM PATH,CALLCODE:BYTE
□DIM TEST:STRING[10]
□REGS.A=0
□REGS.X=ADDR(TEST)
□REGS.Y=10
□CALLCODE=$8B
□RUN SYSCALL(CALLCODE,REGS)
□PRINT
□PRINT TEST
□END
```

TAB Causes PRINT to jump to the specified column

Syntax: **TAB(*number*)**

Function: Causes PRINT to display the next PRINT item to display in the column specified by *number*. If the cursor is already past the desired tab position, BASIC09 ignores TAB.

Use POS to determine the current cursor position when displaying characters on the screen.

Screen display columns are numbered from 1, the leftmost column, to a maximum of 255. The size of BASIC09 output buffer varies according to the stack size.

You can also use TAB with PRINT USING statements.

Parameters:

number	The column at which you want PRINT to begin.

Examples:

```
PRINT TAB(20);TITLE$

PRINT TAB(X);ITEMNUMBER;ITEM$
```

Sample Program:

This procedure uses asterisks to simulate a sine wave on the screen. It uses TAB to position each asterisk in the proper location.

```
PROCEDURE sinewave
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 200
□CALCULATE=CALCULATE+SQ(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

TAN Returns the tangent of a value

Syntax: **TAN(*number*)**

Function: Calculates the trigonometric tangent of *number*. Using DEG or RAD, you can specify the measure of the angle (*number*) in either degrees or radians. Radians are the default units.

Parameters:

number	The angle for which you want to find the tangent.

Examples:

```
PRINT TAN(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent values for a number you type.

```
PROCEDURE ratiocalc
□DEG
□DIM ANGLE:REAL
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "Angle","SINE","COSINE","TAN"
□PRINT "----------------------------------------
--------------------"
□PRINT ANGLE,SIN(ANGLE),COS(ANGLE),TAN(ANGLE)
□PRINT
□END
```

TRIM$ Removes spaces from the end of a string

Syntax: **TRIM$(*string*)**

Function: Removes any trailing spaces from the end of the specified string. This function is particularly useful for trimming records you recover from a random access file.

Parameters:

string	The string or string variable from which you wish to remove trailing spaces.

Examples:

```
PRINT TRIM$(A$)

GET A$,B$,C$
PRINT TRIM$(A$),TRIM$(B$),TRIM$(C$)
```

Sample Program:

This program takes names you type and puts them in a random access disk file. Because random access files use the same amount of storage for each item, short names are *padded* with extra spaces. When reading the names back from the file, the procedure uses TRIM$ to remove these extra spaces.

```
PROCEDURE namestor
□DIM NAMES,TEMP1,NAME(100):STRING[26]; FIRST,LAST:
STRING[15]; INITIAL:STRING[1]
□DIM PATH,T:INTEGER
□NAMES=""
□ON ERROR GOTO 10
□DELETE "namelist"
10□ON ERROR
□CREATE #PATH,"namelist":UPDATE
□FOR T=1 TO 100
□NAME(T)=""
□NEXT T
□T=0
```

```
□LOOP
□INPUT "First Name: ",FIRST
□EXITIF FIRST="" THEN
□CLOSE #PATH
□GOTO 100
□ENDEXIT
□INPUT "Initial: ",INITIAL
□INPUT "Last: ",LAST
□T=T+1
□NAME(T)=FIRST+" "+INITIAL+" "+LAST
□PUT #PATH,NAME(T)
□SEEK #PATH,T*26
□ENDLOOP
100□OPEN #PATH,"namelist":READ
□PRINT  \ PRINT
□PRINT "Lastname","Firstname","Initial"
□REM Print underline (40 characters)
□PRINT "________________________________________"
□PRINT
□SEEK #PATH,0
□T=0
□WHILE NOT(EOF(#PATH)) DO
□GET #PATH,NAMES
□T=T+1
□NAMES=TRIM$(NAMES)
□X=SUBSTR(" ",NAMES)
□FIRST=LEFT$(NAMES,X-1)
□TEMP1=RIGHT$(NAMES,LEN(NAMES)-X+1)
□INITIAL=MID$(TEMP1,2,1)
□LAST=RIGHT$(TEMP1,LEN(TEMP1)-3)
□PRINT LAST,FIRST,INITIAL
□SEEK #PATH,T*26
□ENDWHILE
□CLOSE #PATH
□END
```

TRON/TROFF Turns on/off trace function

Syntax: **TRON**
TROFF

Function: Turns on or off the BASIC09 trace mode. When trace is turned on (TRON), BASIC09 decompiles and displays each statement in a procedure before execution. BASIC09 also displays the result of each expression after evaluation. This function lets you follow program flow and is helpful in debugging and analyzing the execution of a procedure. After the procedure is debugged, remove the TRON statement.

If you want to view only a portion of a procedure's execution, surround that portion with TRON and TROFF. Tracing begins immediately after the TRON statement and ends at the TROFF statement. The rest of the program executes normally.

Parameters: None

Examples:

```
B$="00000000000000000000"+B$
N=1+LEN(B$)
FOR I=4 TO 1 STEP -1
TRON
N=N-4
A(I)=VAL(MID$(B$,N,4))
TROFF
NEXT I
```

TRUE Returns a Boolean TRUE value

Syntax: *variable*=TRUE

Function: TRUE is a Boolean function that always returns True. You can use TRUE and FALSE to assign values to Boolean variables.

Parameters:

variable The Boolean storage unit you want to set to True.

Examples:

```
DIM TEST:BOOLEAN
TEST=TRUE
```

Sample Program:

This procedure asks five questions. If your answer is correct, it stores the Boolean value TRUE in a Boolean type variable. If your answer is incorrect, it stores the Boolean value FALSE in the variable.

```
PROCEDURE quiz
DIM REPLY,VALUE:BOOLEAN; ANSWER:STRING[1];
QUESTION:STRING[80]
FOR T=1 TO 5
READ QUESTION,VALUE
PRINT QUESTION
PRINT "(T) = TRUE       (F) = FALSE"
PRINT "Select T or F:  ";
GET #1,ANSWER
IF ANSWER="T" THEN
REPLY=TRUE
ELSE
REPLY=FALSE
ENDIF
IF REPLY=VALUE THEN
PRINT  \ PRINT "That's Correct...Good Show!"
ELSE
```

```
□PRINT "Sorry, you're wrong...Better Luck next
time."
□ENDIF
□PRINT  \ PRINT  \ PRINT
□NEXT T
□DATA "In computer talk, CPU stands for Central
Packaging Unit.", FALSE
□DATA "The actual value of 64K is 65536
bytes.",TRUE
□DATA "The bits in a byte are normally numbered Ø
through 7.",TRUE
□DATA "BASICØ9 has four data types.",FALSE
□DATA "The LAND function is a Boolean type
operator.",FALSE
□END
```

TYPE Defines a data type

Syntax: **TYPE *name* = *typedeclar* [;*typedeclar*[;...]]**

Function: Defines new data types (complex data structures). New data types are *vectors* (one-dimensional arrays) of previously defined types. Structures created by TYPE differ from arrays in that they can consist of elements of different types, and BASIC09 accesses elements by field names rather than by an indexed position.

Parameters:

name	The name you select for the new data type.
typedeclar	One or more type declarations, which can consist of field names, type declarations, and subscripts. Separate different types or different lengths of string declarations with semicolons.

Notes:

- Complex data structures allow you to create data types that are appropriate for a specific task. You can organize, read, and write associated data naturally. Also, BASIC09 establishes and defines element positions at compilation time. This saves time and overhead at run time because BASIC09 can access the elements of a data structure faster than it can access the elements of an array.
- When you define new data structures using TYPE, you can include any of the five existing data types (string, real, integer, byte, and Boolean), or you can include data structure types that you previously defined with TYPE. This means that your structures can be simple or very complex, such as non-rectangular data lists or trees.
- TYPE does not create storage. You create storage using the DIM statement, after using TYPE.
- To access elements of a data structure, use the field name as well as any appropriate element index.

- For more information on creating and using complex data types, see "Complex Data Types" in Chapter 6.

Examples:

```
TYPE LIBRARY=TITLE,AUTHOR,PUBLISHER:STRING[25];
REFERENCE:INTEGER
DIM BOOK(500):LIBRARY

TYPE PARTS=ITEM,LOCATION:STRING[20]; CAT:REAL;
QUANTITY;INTEGER
DIM INVENTORY(1000):PARTS
```

Sample Program:

This procedure builds an array to contain a book reference list, including the book title, the author's name, the publisher, and a reference number. It does so by using TYPE to create a special data structure to store all the information for each book.

```
PROCEDURE books
□TYPE LIBRARY=TITLE,AUTHOR,PUBLISHER:STRING[30];
REFERENCE:INTEGER
□DIM BOOKS(100):LIBRARY
□T=0
□LOOP
□T=T+1
□INPUT "BOOK TITLE...",BT$
□BOOKS(T).TITLE=BT$
□EXITIF BOOKS(T).TITLE="" THEN
□GOTO 100
□ENDEXIT
□INPUT "Book Author...",BA$
□BOOKS(T).AUTHOR=BA$
□INPUT "Book Publisher...",BP$
□BOOKS(T).PUBLISHER=BP$
□INPUT "Reference Number...",BOOKS(T).REFERENCE
□ENDLOOP
100□FOR X=1 TO T-1
□PRINT BOOKS(X).TITLE; " , "; BOOKS(X).AUTHOR; " ,
";
□BOOKS(X).PUBLISHER; " , "; BOOKS(X).REFERENCE
□NEXT X
□END
```

UNTIL Terminates a REPEAT loop on specified condition

Syntax: **REPEAT**
procedure lines
UNTIL *expression*

Function: Ends a REPEAT loop. REPEAT establishes a loop that executes the encompassed procedure lines until the result of the expression following UNTIL is true. Because the loop is tested at the bottom, the lines within the loop are executed at least once.

Parameters:

procedures lines	Statements you want to execute in the loop.
expression	A Boolean expression (the result must be either True or False).

Examples:

```
REPEAT
COUNT = COUNT+1
UNTIL COUNT > 100

INPUT X,Y
REPEAT
X = X-1
Y = Y-1
UNTIL X<1 OR Y<0
```

See REPEAT for more information.

USING Formats PRINT output

Syntax: **PRINT [*#path*] USING [*format,*] *data*[;*data*...]**

Function: Prints data using a format you specify. This statement is especially useful for printing report headings, accounting reports, checks, or any document requiring a specific format.

USING is actually an extension of the PRINT statement. The same rules that apply to the PRINT statement also apply to the PRINT USING statement (see PRINT).

Parameters:

path	The number to an opened device or file. If you do not specify *path* the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
format	An expression specifying the arrangement of the displayed data.
data	Any numeric or string constant or variable. Always enclose string constants within quotation marks. Separate all data items with semicolons or commas.

See PRINT USING for more information.

VAL Converts string data to numeric data

Syntax: VAL(*string*)

Function: Converts string-type data to numeric-type. VAL is the inverse of the STR$ function. It returns the real value represented by the characters in a string. If any character in the specified string is not numeric, BASIC09 returns an error.

Parameters:

string	An ASCII string containing one or more of the following characters: 0123456789.+$-.

Examples:

```
PRINT VAL(123)

A$="44.66"
PRINT VAL(A$)
```

Sample Program:

This procedure calculates an exponential value, then adds the necessary number of zeroes to convert it to standard notation. It uses STR$ to convert the original number to a string, then uses VAL to convert the exponent into a value to determine the correct decimal place.

```
PROCEDURE bignum
□DIM  C,PLACES,B,SIGN:STRING;  EX,COUNT,NEWCOUNT,
DECIMAL:INTEGER
□DIM NEW,ZERO,NEWEST:STRING[100]
□COUNT=-1
□ZERO="0000000000000000000000000000000000000000"
□NEW="" \NEWEST=""
□INPUT "What number do you want to raise to the
power of 14?...",NUM
□A=NUM^14
□B=STR$(A)
□EX=SUBSTR("E",B)
□SIGN=MID$(B,EX+1,1)
```

```
□PLACES=RIGHT$(B,LEN(B)-EX-1)
□FOR T=1 TO LEN(B)
□C=MID$(B,T,1)
□IF C="." THEN
□DECIMAL=0
□GOTO 10
□ENDIF
□DECIMAL=DECIMAL+1
□IF C="E" THEN 100
□NEW=NEW+C
10□NEXT T
100□NEWCOUNT=VAL(PLACES)-DECIMAL
□NEW=NEW+LEFT$(ZERO,NEWCOUNT+1)
□FOR T=LEN(NEW) TO 1 STEP -1
□COUNT=COUNT+1
□NEWEST=MID$(NEW,T,1)+NEWEST
□IF MOD(COUNT,3)=2 AND T>1 THEN
□NEWEST=","+NEWEST
□ENDIF
□NEXT T
□PRINT NUM; " to the power of 14 = "; A
□PRINT "= "; NEWEST
□END
```

WHILE/DO/ENDWHILE

Establishes a loop

Syntax: **WHILE *expression* DO**
procedure lines
ENDWHILE

Function: Establishes a loop that executes the encompassed procedure lines while the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless the expression is true.

Parameters:

expression	A Boolean expression (has a result of True or False).
procedure lines	Program lines to execute if the expression is true.

Examples:

```
WHILE COUNT < 12 DO
COUNT = COUNT+1
ENDWHILE
```

Sample Program:

You must create a file of directory names using the GET sample program before you can use the following procedure. Copyutil uses the filenames created by the GET sample program to copy a directory's files to another directory you specify. You must specify a directory name that does not exist. Copyutil uses a WHILE/DO/ENDWHILE loop to continue copying until BASIC09 reaches the end of the file.

```
PROCEDURE copyutil
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ
□INPUT "Name of new directory...",DIRNAME
□SHELL "MAKDIR "+DIRNAME
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE
□JOB=FILE+" "+DIRNAME+"/"+FILE
□ON ERROR GOTO 10
□PRINT "COPY "; JOB
□SHELL "COPY "+JOB
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

WRITE Writes data to a sequential file or device

Syntax: **WRITE [*#path,*] *data***

Function: Writes an ASCII record to a sequential file or to a device.

Parameters:

path	A variable containing the path number of the file or device to which you want to send data. *Path* can be one of the the standard I/O paths (0, 1, 2).
data	The data you want to send over the specified path.

Notes:

The following information deals with writing sequential disk files:

- To write file records, you must first dimension a variable to contain the path number of the file, then use OPEN or CREATE to open a file in the WRITE or UPDATE access mode.
- Records can be of any length within a file.
- Individual data items in the input record are separated by ASCII null characters. You can also separate numeric items with comma or space character delimiters. Each input record is terminated by a carriage return character.

Examples:

```
WRITE #PATH,DATA$

WRITE #1,RESPONSE$

WRITE #OUTPUT,INDEX(X)
```

```
OPEN #PATH,"namefile":WRITE
FOR T=1 TO 10
READ NAME$
WRITE #PATH, NAME$
NEXT T
CLOSE #PATH
DATA "JIM","JOE","SUE","TINA","WENDY"
DATA "SALL","MICKIE","FRED","MARV","WINNIE"
```

Sample Program:

This procedure selects 100 random values between 1 and 10. It uses WRITE to place the values into a disk file. Next, it reads the values from the file and uses asterisks to indicate how many times RND selected each value.

```
PROCEDURE randlist
□DIM SHOW,BUCKET:STRING
□DIM T,PATH,SELECT(10),R:INTEGER
□BUCKET="**********"
□FOR T=1 TO 10
□SELECT(T)=0
□NEXT T
□ON ERROR GOTO 10
□SHELL "DEL RANDFILE"
10□ ON ERROR
□CREATE #PATH,"randfile":UPDATE
□FOR T=1 TO 100
□R=RND(9)+1
□WRITE #PATH,R
□NEXT T
□PRINT "Random Distribution"
□SEEK #PATH,0
□FOR T=1 TO 100
□READ #PATH,NUM
□SELECT(NUM)=SELECT(NUM)+1
□NEXT T
□FOR T=1 TO 10
□SHOW=RIGHT$(BUCKET,SELECT(T))
□PRINT USING "S6<,I3<,S2<,S20<","Number",T,":",
SHOW
□NEXT T
□CLOSE #PATH
□END
```

XOR Returns the exclusive OR of two values

Syntax: *operand1* **XOR** *operand2*

Function: Performs the logical exclusive OR operation on two or more values, returning a value of either TRUE or FALSE.

Parameters:

operand1 *operand2*	Boolean values or expressions (that result in values of True or False).

Examples:

```
PRINT A>2 XOR B>3

PRINT A$="YES" XOR B$="YES"
```

Sample Program:

This procedure lets two people type numbers until one of them guesses the number that the computer picks. It uses XOR to determine that one of the numbers typed is the correct number, but not both.

```
PROCEDURE drawstraw
□DIM NUM1,NUM2,R:INTEGER; A:BOOLEAN
□PRINT "This program will help you pick"
□PRINT "between two people. Choose who will be"
□PRINT "Person 1 and who will be Person 2."
□PRINT "Then, enter numbers between 1 and 10"
□PRINT "when requested."
□PRINT
□R=RND(10)
10□INPUT "Person 1, type a number and press
ENTER...",NUM1
□INPUT "Person 2, type a number and press
ENTER...",NUM2
□A=NUM1=R XOR NUM2=R
□IF A=FALSE THEN
□PRINT "You'll have to try again..."
□PRINT
```

```
GOTO 10
ENDIF
IF NUM1=R THEN
PRINT "You win, Person 1"
ENDIF
IF NUM2=R THEN
PRINT "You win, Person 2"
ENDIF
PRINT "The Number was..."; R
END
```

Chapter 12

Program Optimization

BASIC09's multipass compiler produces a compressed and optimized low-level I-code for execution. Compared to other BASIC languages, BASIC09 greatly decreases both the storage space required for program code and the execution speed of programs.

Because BASIC09 produces I-code at a powerful level, it can handle numerous MPU (micro processor unit) instructions with a single interpretation. Therefore, for complex programs, there is little performance difference between the execution of I-code and pure machine-language instructions.

Most BASIC languages have to compile from text as they run, or search tables of *tokens* in order to execute code. Instead, BASIC09 I-code instructions contain direct references to variables, statements, and labels. BASIC09 fully utilizes the power of the 6809 instruction set, as well, which is optimized for efficient execution of compiler-produced code.

Because BASIC09 interprets I-code, you have a variety of entry-time and run-time tests and development aids. The editor reports syntax errors immediately when they are entered. The debugger lets you debug using original program source statements and names. The I-code interpreter performs run-time error checking of array structures and BASIC09 functions.

Optimum Use of Numeric Data Types

The following notes apply to the use of BASIC09 numeric data types:

- BASIC09 includes several different numeric representations (real, integer, byte), and performs automatic type conversions between them. This means that without care, your code might contain expressions or loops that take more than ten times longer to execute than is necessary.

- Some BASIC09 numeric operators, such as +,-,*, and /, and some BASIC09 control structures include versions for both real and integer values. Integer versions execute much faster and can have slightly different properties. For instance, integer division discards any remainder.

  Integer operations are faster because they use corresponding 6809 instructions. Using integers increases speed and decreases storage requirements. Integer operations use the same symbols as real operations, but BASIC09 automatically selects the integer operations when when all operands of an expression are of byte or integer type.

- Type conversion takes time. Using expressions with operands and operators of the same kind is most efficient.

- BASIC09's real (floating point) math provides excellent performance. It includes a 40-bit binary floating point representation and uses the CORDIC technique to derive all transcendental functions. This integer shift-and-add technique is faster and more consistent than the common series-expansion approximations.

- At times, you can obtain similar or identical results in a number of different ways and at different execution speeds. For example, if the variable Value is an integer, then `Value*2` is a fast integer operation. However, if the expression is `Value*2.0`, 2.0 is represented as a real number and the operation requires real multiplication. BASIC09 must transform the integer Value into a real value. If the result of the expression is assigned to an integer type variable, BASIC09 must transform the result back to an integer type. The decimal point can slow the operation by about ten times.

Arithmetic Functions Ranked by Speed

Operation	Typical Speed in MPU Cycles
Integer add or subtract	150
Integer multiply	240
Real add	440
Real subtract	540
Integer divide	960
Real multiply	990
Real divide	3870
Real square root	7360
Real logarithm or exponential	20400
Real sine or cosine	32500
Real power	39200

Referring to the previous table can help you in your programming. For instance, notice that it is quicker to add a value to itself rather than multiplying it by 2. Similarly, multiplying a value by itself or using SQ on a value is much faster than raising a value to the power of 2.

Notice that a real divide takes 3870 cycles, while a real multiplication takes only 990 cycles. Multiplying a value by 0.5 is four times quicker than dividing the value by 2.

Quicker Loops

BASIC09 has two versions of FOR/NEXT loops, one for integer loop counter variables and one for real loop counter variables. It automatically uses the appropriate version. Integer FOR/NEXT loops are much faster than real FOR/NEXT loops.

Other kinds of loops also run faster if you use integer type variables for the loop counters. When writing program loops, remember that statements inside the loop can execute many times for each execution outside the loop. Whenever possible, compute values before entering loops.

Arrays and Data Structures

The internal workings of BASIC09 use integer numbers to index arrays and complex data structures. This means that BASIC09 must convert real type variable or expression subscripts before it can handle them. Using integer expressions for subscripts increases execution speed.

Using the assignment statement LET to copy identically sized data structures is much faster than copying arrays or structures element-by-element inside a loop.

The PACK Command

PACK causes a second compilation of a specified procedure. Depending on such variables as the number of procedure comments and the inclusion of line numbers, packed procedures execute from 10 to 30 percent faster. Line numbers cause unpacked procedures to run slower.

Minimizing Constant Expressions and Subexpressions

For maximum execution speed, precalculate constant expressions. For instance, `x = x+5` produces the same result as `x = x+sqrt(100)/2`. However, the first expression requires approximately 150 MPU cycles while the second expression requires 11,650 MPU cycles. If you use such an expression inside a loop, the additional execution time is enormous.

Input and Output

Accessing data one line or record at a time is much faster than accessing it one character at a time. Also, the GET and PUT statements are much faster than READ and WRITE statements when accessing disk files. This is because GET and PUT use the same binary format as BASIC09's internal operations. READ, WRITE, PRINT, and INPUT must perform binary-to-ASCII or ASCII-to-binary conversions, which take more time.

Error Codes

Signal Errors

Code	Meaning
1	Unconditional termination
2	Keyboard termination
3	Keyboard interrupt

BASIC09 Error Codes

Code	Meaning
10	Unrecognized symbol
11	Excessive verbiage
12	Illegal statement construction
13	I-code overflow, need more workspace memory
14	Illegal channel reference, bad path number given
15	Illegal mode (read/write/update) - directory only
16	Illegal number
17	Illegal prefix
18	Illegal operand
19	Illegal operator
20	Illegal record field name
21	Illegal dimension
22	Illegal literal
23	Illegal relational
24	Illegal type suffix
25	Too-large dimension
26	Too-large line number
27	Missing assignment statement
28	Missing path number
29	Missing comma
30	Missing dimension
31	Missing DO statement
32	Memory full, need more workspace memory
33	Missing GOTO
34	Missing left parenthesis
35	Missing line reference
36	Missing operand
37	Missing right parenthesis
38	Missing THEN statement
39	Missing TO

Code	Meaning
40	Missing variable reference
41	No ending quote
42	Too many subscripts
43	Unknown procedure
44	Multiply-defined procedure
45	Divide by zero
46	Operand type mismatch
47	String stack overflow
48	Unimplemented routine
49	Undefined variable
50	Floating overflow
51	Line with compiler error
52	Value out of range for destination
53	Subroutine stack overflow
54	Subroutine stack underflow
55	Subscript out of range
56	Parameter error
57	System stack overflow
58	I/O type mismatch
59	I/O numeric input format bad
60	I/O conversion: number out of range
61	Illegal input format
62	I/O format repeat error
63	I/O format syntax error
64	Illegal path number
65	Wrong number of subscripts
66	Non-record-type operand
67	Illegal argument
68	Illegal control structure
69	Unmatched control structure
70	Illegal FOR variable
71	Illegal expression type
72	Illegal declarative statement
73	Array size overflow
74	Undefined line number
75	Multiply-defined line number
76	Multiply-defined variable
77	Illegal input variable
78	Seek out of range
79	Missing data statement

Windowing and System Errors

Code	Meaning
183	Illegal window type
184	Window already defined
185	Font not found
186	Stack overflow
187	Illegal argument
188	(unused)
189	Illegal coordinates
190	Internal integrity check
191	Buffer size is too small
192	Illegal command
193	Screen or window table is full
194	Bad/undefined buffer number
195	Illegal window definition
196	Window undefined
197	(unused)
198	(unused)
199	(unused)
200	Path table full
201	Illegal path number
202	Interrupt polling table full
203	Illegal mode
204	Device table full
205	Illegal module header
206	Module directory full
207	Memory full
208	Illegal service request
209	Module busy
210	Boundary error
211	End of file
212	Returning non-allocated memory
213	Non-existing segment
214	No permission
215	Bad path name
216	Path name not found
217	Segment list full
218	File already exists
219	Illegal block address
220	Phone hangup data carrier detect lost
221	Module not found
223	Suicide attempt

Code	Meaning
224	Illegal process number
226	No children, can't wait for nonexistent child process
227	Illegal SWI code
228	Process aborted, signal 2
229	Process table full, can't fork a process
230	Illegal parameter area
231	Known module
232	Incorrect module CRC
233	Signal error
234	Non-existent module
235	Bad name
237	System RAM full
238	Unknown process ID
239	No task number available
240	Illegal unit error
241	Bad sector number
242	Write protected disk
243	CRC error
244	Read error
245	Write error
246	Not ready, device not ready
247	Seek error
248	Media full
249	Wrong type, incompatible media type
250	Device busy
251	Disk ID change, disk changed with open files
252	Record is locked out
253	Non-sharable file busy

The Inkey Program

Assembly Language Listing of Inkey

An assembled version of Inkey is included on the CONFIG/BASIC09 diskette. Use Inkey from BASIC09 with the RUN statement.

```
**************************
* INKEY - a subroutine for BASIC09 by Robert Doggett
*
* Called by: RUN INKEY(StrVar)
*          RUN INKEY(Path,StrVar)
* INKEY determines if a key has been typed on the given path
* (Standard Input if not specified), and if so, returns the next
* character in the String Variable. If no key has been typed, the
* null string is returned. If a path is specified, it must be
* either type BYTE or INTEGER.

                          NAM    INKEY
                          IFP1
                          USE    /D0/DEFS/OS9DEFS
                          ENDC

  0021             TYPE    set    SBRTN+OBJCT
  0081             REVS    set    REENT+1

  0000  87CD005E           mod    InKeyEnd,InKeyNam,TYPE,REVS
                                  ,InKeyEnt,SIZE
  000D  496E6B65   InKeyNam fcs   "Inkey"
D 0000                     org    0          Parameters
D 0000             Return  rmb    2          Return addr of caller
D 0002             PCount  rmb    2          Num of params following
D 0004             Param1  rmb    2          1st param addr
D 0006             Length1 rmb    2          size
D 0008             Param2  rmb    2          2nd param addr
D 000A             Length2 rmb    2          size
  000C             E$Param equ    $38
  000E             SIZE    equ    *
  0012  3064       InKeyEnt leax  Param1,S
```

```
0014  EC62              ldd    Pcount,S    Get parameter count
0016  10830001          cmpd   #1          just one parameter?
001A  2727              beq    InKey20     ..Yes; default path A=0
001C  10830002          cmpd   #2          Are there two params?
0020  2635              bne    ParamErr    No, abort
0022  ECF804            ldd    [Param1,S]  Get path number
0025  AE66              ldx    Length1,S
0027  301F              leax   -1,X        byte variable?
0029  2706              beq    InKey10     ..Yes; (A)=Path number
002B  301F              leax   -1,X        Integer?
002D  2628              bne    ParamErr    ..No; abort
002F  1F98              tfr    B,A
0031  3068        InKey10 leax   Param2,S
0033  EE02        InKey20 ldu    2,X         length of string
0035  AE84              ldx    0,X         addr of string
0037  C6FF              ldb    #$FF
0039  E784              stb    0,X         Initialize to null str
003B  11830002          cmpu   #2          at least two-byte str?
003F  2502              blo    InKey30     ..No
0041  E701              stb    1,X         put str terminator
0043  C601        InKey30 ldb    #SS.Ready
0045  103F8D            OS9    I$GetStt    is there any data ready?
0048  2508              bcs    InKey90     ..No; exit
004A  108E0001          ldy    #1
004E  103F89            OS9    I$Read      Read one byte
0051  39                rts                returns error status
0052  C1F6        InKey90 cmpb   #E$NotRdy
0054  2603              bne    InKeyErr
0056  39                rts                (carry clear)
0057  C638        ParamErr ldb   #E$Param    Parameter Error
0059  43          InKeyErr coma
005A  39                rts
005B  1A6916            emod
005E            InKeyEnd equ    *
```

Index

ABS command 11-4
absolute value 11-4
accessing
    files 8-1, 10-8
    lines (editor) 4-4 - 4-5
    OS-9 commands from BASIC 3-7
ACS command 11-5
adding lines 4-10 - 4-12
addition 7-3 - 7-4
ADDR command 11-6
address
    of variable 6-8, 11-6
    space 11-6
advantages of BASIC09 1-1 - 1-2
ALPHA (medium-res) 9-9, 9-13
alphanumeric
    mode 9-10
    screen 9-9, 9-13, 9-30
ALT key 1-6, 9-4
AND
    command 11-8
    logical AND command 11-84
    operator 7-3, 7-4, 7-7
appending
    data to files 8-3
    strings 7-6
ARC command (high-res) 9-50
arccosine 11-5
arcsine 11-10
arctangent 11-11
arithmetic
    function speed 12-2
    operators 7-3
array 6-9 - 6-13
    address 11-6
    element 6-9
    index 11-12
    with random access files 8-9
ASC command 11-9
ASCII
    character value 11-18
    codes 9-1 - 9-6, 11-9
ASN command 11-10
assign
    variable storage 11-31
    variable values 11-78
    variables (debug) 5-3
ATN command 11-11
auto execution 3-8
automatic error checking 1-4

background color
    high-resolution 9-34
    medium-resolution 9-11
backslash 1-6
BAR command (high-res) 9-52 - 9-53
base 10 logarithm 11-83
BASE command 11-12 - 11-13
BASIC09
    advantages 1-1 - 1-2
    graphics with 128K 9-37 - 9-39
    quitting 1-5, 3-1
    starting 1-2 - 1-4
    starting windows from 9-39 - 9-41
beep 9-54
beginning debug 5-1
BELL command (high-res) 9-54
binary data record 11-58
BLNKOFF command (high-res) 9-55
BLNKON command (high-res) 9-55
BOLDSW command (high-res) 9-56

Boolean
data 6-1 - 6-2, 6-5
functions 7-10
OR 11-106
TRUE 11-175 - 11-176
value 11-51
border color (high-res) 9-58, 9-65
BORDER command (high-res) 9-58
BOX command 9-60 - 9-61
brace characters 1-6
BREAK
command (debug) 5-2
key 1-6, 5-2
breakpoint (debug) 5-2
buffer
defining 9-78
font (high-res) 9-94
get/put (high-res) 9-117
group (high-res) 9-101
pattern (high-res) 9-111
button, joystick (medium-res) 9-9, 9-22
BYE command 1-5, 3-1, 10-9, 11-14
byte
data type 6-1 - 6-2
numeric range 6-2
retrieval from a file 8-5
type functions 7-9

calculate
low-res characters 9-5
sine 11-154
square root 11-158
call a shell command 10-9
carriage return 1-7
high-resolution 9-67
CHAIN command 11-15 - 11-16
changing
a procedure name 10-9
color (high-res) 9-65 - 9-66
changing (*cont'd*)
color (medium-res) 9-9
directory 3-1, 3-7, 10-9, 11-17, 11-19
file pointer 11-148
procedures 1-4
scale (high-res) 9-121 - 9-122
text 4-7 - 4-9
text (editor) 4-1 - 4-2
working area (high-res) 9-76
character
backslash 1-6
blink (high-res) 9-55
braces 1-6
brackets 1-6
fonts 9-43 - 9-44
graphic 1-6
high-resolution 9-8, 9-94
reverse video (high-res) 9-120
tilde 1-6
underline (high-res) 9-126
underscore 1-6
up arrow 1-6
value 11-18
vertical bar 1-6
CHD command 3-1, 3-7, 10-9, 11-17, 11-19
CHX command 3-1, 3-7, 10-9, 11-17 - 11-19
CIRCLE
high-resolution 9-62
medium-resolution 9-9, 9-15 - 9-16
CLEAR
high-resolution 9-64
key 1-6
medium resolution 9-9, 9-17
close a window (high-res) 9-83 - 9-84

CLOSE command 11-20 - 11-21
code
    ASCII 9-1 - 9-6, 11-9
    error 11-43, A1 - A4
COLOR
    high-resolution 9-65
    medium resolution 9-9, 9-18, 9-19
color
    codes (medium-res) 9-10 - 9-11
    default 9-79
    high-resolution 9-31, 9-109 - 9-110
    medium-resolution 9-11
    of border (high-res) 9-58 - 9-59
    of pixel (medium-res) 9-28 - 9-29
    of screen (medium-res) 9-26
    palette default 9-79
    set (medium-res) 9-18 - 9-19
command
    interpreter 3-1
    line storage area 3-3
    line symbols 11-2
    lines using spaces 2-2
    mode 1-3
    mode reference 10-9
commands
    by type 10-7
    configuring (high-res) 9-47
    debug 10-11
    drawing (high-res) 9-46
    editing 10-10
    executing OS-9 3-7 - 3-8
    font (high-res) 9-49
    quick reference 10-1 - 10-6
    system 3-1
commands (*cont'd*)
    text/cursor (high-res) 9-48
    using wildcards 3-5
    window (high-res) 9-45
comments in a procedure 11-135 - 11-136
compile procedure 3-1, 3-8 - 3-9, 10-9
compiler, multipass 12-1
compiling
    procedures 1-5
    saving space 1-2
complement, logical 11-96
complex
    data structure 1-2, 8-11 - 8-12, 11-177 - 11-178
    data types 6-1, 6-13 - 6-16
compressed procedures 12-1
concatenation 7-3
condensed procedures 3-1
configuring commands (high-res) 9-47
constant expressions 12-4
constants, string 6-7
control key 1-6
converting
    data types 6-6, 7-2
    numeric types 11-54, 11-71, 11-162 - 11-163
    string data 11-181 - 11-183
copying structure elements 6-16
COS command 11-22
cosine 11-22
create
    data types 11-177
    overlay windows (high-res) 9-107
    procedures 2-1
    random access files 8-6 - 8-9

create (*cont'd*)
sentences procedure 4-3
sequential files 8-2 - 8-3
windows 9-35 - 9-36
CREATE command 8-2 - 8-3, 8-6 - 8-7, 11-23 - 11-24
CRRTN command (high-res) 9-67
CTRL key 1-6 - 1-7
CTRL-BREAK key sequence 1-6, 3-1
CURDWN command (high-res) 9-68
CURHOME command 9-69
CURLFT command (high-res) 9-70
CUROFF command (high-res) 9-71
CURON command (high-res) 9-72
current command line 1-7
CURRGT command (high-res) 9-73
cursor
graphics (high-res) 9-95, 9-119
graphics (medium-res) 9-27
invisible (high-res) 9-71
movement 1-6, 9-67 - 9-68, 9-74 - 9-75
position 11-116
CURUP command (high-res) 9-74
CURXY command (high-res) 9-75
CWAREA command (high-res) 9-76 - 9-77

data
changing in sequential file 8-4
complex types 6-1, 6-13 - 6-16
constants 6-6 - 6-7
data (*cont'd*)
directory 3-7
items 6-1
manipulation 7-1 - 7-2
meaning 6-1
pointer 11-140
reading 11-132 - 11-133
structure 1-2, 11-177 - 11-178, 12-2
structure address 11-6
to files 8-1
type, Boolean 6-5
type, byte 6-2
type, conversion 7-2
type, integer 6-3
type, real 6-3 - 6-4
types 6-1, 10-8, 11-177 - 11-178, 12-1
types, creating 11-177 - 11-178
DATA command 11-25 - 11-26
DATE$ command 11-27 - 11-28
day 11-27
deallocate
buffer (high-res) 9-101 - 9-102
graphics memory 9-30
windows (high-res) 9-83 - 9-84
debug
beginning 5-1
breakpoint 5-2
commands 5-2 - 5-4, 10-11
display procedure 5-3
quitting 5-3
starting 5-1, 5-4 - 5-5, 11-112
tracing 5-4
debug command
$ 5-2
BREAK 5-2
CONT 5-2

debug command (*cont'd*)
DEG 5-2
DIR 5-3
LET 5-3
LIST 5-3
PRINT 5-3
Q 5-3
RAD 5-2
STATE 5-3
STEP 5-4
TROFF 5-4
TRON 5-4
default colors 9-79
DEFBUFF command (high-res) 9-78
DEFCOL command (high-res) 9-79
define a window (high-res) 9-86 - 9-87
defining string variables 6-4
DEG command 11-29
degrees, selecting in debug 5-2, 11-29
DELETE command 11-30
delete line 1-6, 2-2
editor 4-2
high-resolution 9-80, 9-92
deleting
procedure lines 4-6 - 4-7
procedures 3-6
delimiter 4-8
in sequential files 8-2
symbols (editor) 4-8
DELLIN command (high-res) 9-80
device path 11-104
DIM command 11-31 - 11-32
DIM statement 6-2, 11-31
DIR
command 3-1 -3-2, 10-9
debug 5-3
file access 8-1
directory
change 3-1, 3-7, 11-17, 11-19
data 3-7
execution 3-7
ROOT 3-7
disassembled procedure 3-3
disk file 8-1
creation 11-23
deletion 11-30
display
a formatted listing 10-9
a window 1-6, (high-res) 9-123 - 9-124
clearing (medium-res) 9-17
current command line 1-7
last line 1-7
previous window 1-6
procedure 3-1
procedure from debug 5-3
procedure information 3-1, 10-9
text 11-119 - 11-120
workspace size 3-1, 10-9
division 7-3
remainder 11-93
DO command 11-34
dot, graphics (medium-res) 9-28 - 9-29
draw
a circle (high-res) 9-62 - 9-63
a circle (medium-res) 9-9, 9-15 - 9-16
a line (high-res) 9-103 - 9-104
an ellipse 9-88 - 9-89
arcs (high-res) 9-50 - 9-51
command (high-res) 9-46, 9-81 - 9-82
pointer (high-res) 9-125

draw (*cont'd*)
pointer (medium-res) 9-12
lines (medium-res) 9-24 - 9-25, 9-103
rectangles (high-res) 9-52 - 9-53, 9-60 - 9-61
DWEND command (high-res) 9-83 - 9-84
DWPROTSW command (high-res) 9-85
DWSET command (high-res) 9-86 - 9-87

edit
compiler 3-1
mode, entering 1-4
pointer 4-1
terminating 2-3
EDIT command 3-1, 10-9 - 10-10
editor 4-1 - 4-9
element 6-9
elements
of a structure, copying 6-16
of an array 6-9
ELLIPSE command (high-res) 9-88 - 9-89
ELSE command 11-35
END command 11-36 - 11-37
end execution 11-14
end-of-file
message 1-6
test 11-42
ENDEXIT command 11-38
ENDIF command 11-39
ENDLOOP command 11-40
ENDWHILE 11-41
ENTER
command (editor) 4-1
in the editor 4-4
key 1-7
entering
debug 5-4 - 5-5
the edit mode 1-4
EOF command 11-42
equal operator 7-5
erase
a disk file 11-30
procedures 3-1, 11-72
to end of line 9-90
to end of window 9-91
EREOLINE command (high-res) 9-90
EREOWNDW command (high-res) 9-91
ERLINE command (high-res) 9-92
ERR command 11-43 - 11-44
error
checking, automatic 1-4
code 11-43 - 11-44, A-1 - A-4
in a program line 2-2
simulation 11-45 - 11-46
trapping 11-97 - 11-99
ERROR command 11-45
escape function 1-6
establishing a window 9-32, 9-41, 9-86 - 9-87
evaluating expressions 7-1 - 7-2
evaluation, order of operators 7-4 - 7-5
examine
a procedure 4-4
memory 11-113
exclusive OR 11-187 - 11-188
EXEC file access 8-1
executable procedures 3-8
execute
a procedure 2-3, 3-1, 3-8, 10-9, 11-145 - 11-147
an OS-9 command 3-1, 3-7 - 3-8

execute (*cont'd*)
modules 11-15 - 11-16
procedure lines 11-34
execution
automatic 3-8 - 3-9
directory change 3-1, 3-7
speed 1-1
stepping 5-5 - 5-6
stopping 11-161
termination 11-14
EXITIF/THEN/ENDEXIT commands 11-47
exiting
BASIC09 1-5
debug 5-3
EXP command 11-50
exponent, natural 11-50
exponentiation 7-3
expression 7-1

FALSE
command 11-51 - 11-52
value 7-7
faster loops 12-2
file
listing procedures to 3-4
path 11-104
pointer 8-3, 8-5, 11-148 - 11-149
pointer, rewinding 8-11
retrieving bytes 8-5
writing 11-129 - 11- 130, 11-185- 11-186
files 8-1
accessing 8-1, 10-8
appending data 8-3
closing 11-20 - 11-21
creating random access 8-6 - 8-9
creating sequential 8-2 - 8-4
creation 11-23 - 11-24
opening 11-104 - 11-105
files (*cont'd*)
random access 8-5 - 8-11
writing to 8-3
FILL command (high-res) 9-93
filled rectangles (high-res) 9-52 - 9-53
finding
graphics screen (medium-res) 9-20 - 9-21
lines 4-5
fire button (medium-res) 9-22
FIX command 11-53
FLOAT command 11-54
FONT command (high-res) 9-94
font-handling commands (high-res) 9-49
fonts 9-43 - 9-44
FOR/NEXT loops 11-159 - 11-160
FOR/NEXT/STEP commands 11-55 - 11-57
foreground color
high resolution 9-65 - 9-66
medium resolution 9-11, 9-18 - 9-19
fork a shell 11-152 - 11-153
to a window 9-32
format
medium resolution 9-10
of screen (medium-res) 9-26
of windows 9-34
formatted procedure 3-1
formatting
display screen 11-180
screen display 11-122 - 11-127
functions 7-7 - 7-10
Boolean type 7-10
byte type 7-9
integer type 7-9

functions (*cont'd*)
logical 7-10
numeric type 7-9, 10-7
real type 7-8
string 7-10, 10-7
trace 5-5 - 5-6
transcendental 10-7

GCOLR (medium-res) 9-9
GCSET command (high-res) 9-95
GET command 8-5, 11-58
high-resolution 9-96
GET/PUT buffer 9-78
high-resolution 9-101
GET/PUT commands (high-res) 9-47
global symbol (editor) 4-5
GLOC (medium-res) 9-9, 9-20
GOSUB/RETURN commands 11-61
GPLOAD command (high-res) 9-98
graphics
characters 1-6
cursor (high-res) 9-95, 9-119
cursor (medium-res) 9-9, 9-27
high-resolution 9-31 - 9-126
levels 9-1
logic functions 9-105
low resolution 9-4 - 9-8
medium-resolution 9-8 - 9-30
memory deallocate 9-30
number of levels 1-2
pattern (high-res) 9-111 - 9-112
pointer (high-res) 9-42
screen (medium-res) 9-26

graphics (*cont'd*)
screen location (medium-res) 9-20 - 9-21
window 9-35 - 9-36
with 128K 9-37 - 9-40
greater than 7-3, 7-5
grid format (medium-res) 9-10
group
buffer (high-res) 9-101 - 9-102
number 9-78

hardware window 9-32 - 9-35
high-resolution 9-31 - 9-126
adapter 9-22
characters 9-8
colors 9-109 - 9-110
quick reference 9-44 - 9-49
text 9-42
hour 11-27

I-Code 3-3, 12-1
IF/THEN/ELSE loop 11-35
IF/THEN/ELSE/ENDIF commands 11-63 - 11-65
image, get (high-res) 9-98
immortal shell 9-32
initialize a disk file 11-23 - 11-24
INIZ command 9-32 - 9-33
Inkey program B-1 - B-2
INPUT command 8-5, 11-68 - 11-70
input/output 12-4
insert
a line (high-res) 9-99 - 9-100
text (editor) 4-1
INSLIN command (high-res) 9-99 - 9-100
INT command 11-71
integer
constants 6-7

integer (*cont'd*)
data type 6-1, 6-2, 6-3
functions 7-9
numeric range 6-2
interfacing with OS-9 1-1
invisible cursor (high-res) 9-71

JOYSTK 9-9, 9-22
jump
to line number 11-102 - 11-103
to subroutine 11-100 - 11-101

key
ALT 1-6, 9-4
BREAK 1-6, 5-2
CLEAR 1-6
CTRL 1-6 - 1-7
ENTER 1-7
key sequence
CTRL with other keys 1-6 - 1-7
SHIFT with other keys 1-6
keyword 11-1
KILL command 3-1, 3-6, 10-9, 11-72 - 11-73
KILLBUFF command (high-res) 9-101
killing a procedure 3-6

LAND command 11-74 - 11-75
language modules 1-5
last line, displaying 1-7
left brace 1-6
left bracket 1-6
LEFT$ command 11-76
LEN command 11-77
length of string variables 6-4
less than 7-3 - 7-4, 7-5
LET command 6-8, 11-78 - 11-79
debug 5-3
LINE (medium-res) 9-9
line
accessing (editor) 4-5
adding 4-10 - 4-12
adding (editor) 4-10
erasing 9-90
see also line, deleting
inserting (high-res) 9-99 - 9-100
jumping to 11-102 - 11-103
numbers 4-5
renumbering 4-2, 4-10
LINE command
high-resolution 9-103
medium-resolution 9-24
line deleting 1-6, 2-2, 9-92
editor 4-2
high-resolution 9-80
in procedures 4-6 - 4-7
LIST command 3-1, 3-2 - 3-5, 4-6, 10-9
listing
procedures 3-2 - 3-5, 6-6, 10-9
procedure lines (editor) 4-2
to a file 3-4
to a printer 3-4
LNOT command 11-80 - 11-81
LOAD command 3-1, 3-6, 10-9
loading
a buffer (high-res) 9-98
BASIC09 1-2 - 1-4
procedures 3-1, 3-6, 10-9
window image (high-res) 9-101 - 9-102
local variable 6-7
LOG command 11-82

LOG10 command 11-83
logarithm 11-82, 11-83
logic comparison 6-5
LOGIC command (high-res) 9-105 - 9-106
logical
    AND 11-8, 11-74 - 11-75
    block (file) 8-1
    complement 11-96
    functions 7-10
    NOT 11-80 - 11-81, 11-96
    operators 7-7
    OR 11-87 - 11-88
    XOR 11-89 - 11-91
loop
    EXITIF/ENDEXIT/ENDEXIT 11-38, 11-47 - 11-49
    FOR/NEXT 11-55, 11-57, 11-95, 11-159 - 11-160
    IF/THEN/ELSE/ENDIF 11-35, 11-39, 11-63 - 11-65
    LOOP/ENDLOOP 11-40, 11-84 - 11-86
    REPEAT/UNTIL 11-137 - 11-139, 11-179
    WHILE/DO/ENDWHILE 11-34, 11-41, 11-183 - 11-184
loop repetition 11-95
LOR command 11-87 - 11-88
low-resolution 9-1 - 9-7
LXOR command 11-89 - 11-91

math 1-2
medium-resolution 9-8 - 9-30
    format 9-10 - 9-11
MEM command 1-3 - 1-4, 3-1, 10-9
memory
    changing 11-116 - 11-117
    examining 11-113 - 11-114
    in the workspace 3-1
    requesting 1-3 - 1-4
    saving 1-2
    size 1-3, 1-4
message, end-of-file 1-6
MID$ command 11-92
minimizing storage 12-1
minutes 11-27
mistakes in program lines 2-2
mixing data types 7-2
MOD command 11-93
MODE (medium-res) 9-9, 9-26
modes
    command 1-3
    edit 1-4
module
    execution 11-15
    high-resolution 9-31
    medium-resolution 9-8 - 9-9
modulus 11-93 11-94
month 11-27
mouse (medium-res) 9-22
MOVE (medium-res) 9-9, 9-27
move cursor 1-6
    high-resolution 9-68, 9-70, 9-73 - 9-75
move
    backward (editor) 4-5
    draw pointer (high-res) 9-125
    graphics cursor (high-res) 9-95
    the edit pointer 4-1
multipass compiler 12-1
multiplication 7-3 - 7-4

natural exponent 11-50
negation 7-3
nesting order (debug) 5-3
NEXT command 11-95
NOT command 11-96
not equal to 7-3, 7-4, 7-5
NOT, logical 11-80 - 11-81
    operator 7-4, 7-7
null constants 6-9
numbers for lines 4-5
numeric
    constants 6-6
    data conversion 11-162 - 11-163
    data types 12-1 - 12-2
    functions 10-7
    type conversion 11-54, 11-71
    type functions 7-9

ON ERROR/GOTO command 11-97 - 11-99
ON/GOSUB command 11-100 - 11-101
ON/GOTO command 11-102 - 11-103
OPEN command 8-3, 11-104 - 11-105
operands 7-2
operators 7-1
    arithmetic 7-3 - 7-4
    equal 7-5
    greater than 7-5
    hierarchy of 7-4
    less than 7-5
    logical 7-7
    relational 7-5 - 7-6
    string 7-6
    types 7-3
    unequal 7-5
OR
    command 11-106
    logical 11-87 - 11-88
    operator 7-7
order
    of nesting (debug) 5-3
    of operators 7-4 - 7-5
OS-9 commands 11-152
    accessing 3-7 - 3-8
overlay windows 9-41, 9-107 - 9-108
OWSET command (high-res) 9-107 - 9-108

PACK command 3-1, 3-8, 3-9, 10-9, 12-4
paint (high-res) 9-93
PALETTE command (high-res) 9-109 - 9-110
palette
    default colors 9-79
    high-resolution 9-34 - 9-35
    registers 9-35
PARAM command 6-8, 11-108 - 11-111
passing variables 6-8, 11-108 - 11-111
path
    input 11-68
    opening 11-104 - 11-105
PATTERN command (high-res) 9-111 - 9-112
PAUSE command 5-5, 11-112
PEEK command 9-20, 11-113 - 11-114
PI command 11-115
pixel 9-34
    color (medium-res) 9-28 - 9-29
    set (high-res) 9-113 - 9-114
plus sign 7-6
POINT
    high-resolution 9-113 - 9-114
    medium-resolution 9-10, 9-28 - 9-29

pointer
draw (hi-res) 9-42, 9-125
draw (medium-res) 9-12
edit 4-1
file 8-5
graphics 9-42
READ 11-140
POKE command 9-20, 11-116
POS command 11-118
position
graphics cursor (medium-res) 9-9
of a record in a file 8-5
of cursor 11-118
power of 2 11-157
predefined windows 9-32 - 9-33
PRINT command 11-119 - 11-120
debug 5-3
PRINT USING command 11-122 - 11-128, 11-180 - 11-182
printer, listing files 3-4
printing (tabs) 11-166 - 11-167
procedure
changing 1-4
comments 11-135 - 11-136
compilation 10-9
compiling 1-5
compressing 12-1
condensing 3-1
data size 3-2
deleting 3-6
disassembling 3-3
display 3-1
displaying information about 3-1
erasing 3-1, 11-72 - 11-73
examining 4-4
executing 1-5
procedure (*cont'd*)
execution 2-3, 3-1
grouping 1-4
listing 3-2 - 3-3, 4-6
loading 3-6
renaming 3-2
returning from 11-141
saving 3-1, 3-5 - 3-6, 10-9
size 3-2
suspending 11-112
terminating 11-36 - 11-37, 11-161
tracing 11-174
writing 2-1 - 2-2
procedures
executable 3-8
executing 11-145 - 11-147
loading 3-1
program
execution termination 1-6
mistakes 2-2
modular 1-1
proportional text (high-res) 9-115 - 9-116
PROPSW command (high-res) 9-115 - 9-116
protect window switch (high-res) 9-85
PUT command 8-5, 8-6, 9-117 - 9-118, 11-129 - 11-130
PUTGC command 9-119

QUIT (medium-res) 9-10, 9-30
quit
BASIC09 1-5, 3-1
debug 5-3
the editor 2-3, 4-2

RAD command 5-2, 11-131
radians 5-2, 11-131

random access files 8-5 - 8-11
and arrays 8-9 - 8-11
creating 8-6 - 8-9
random value 11-143 - 11-144
range of numbers 6-2
READ 8-4, 11-25, 11-132 - 11-133
file access 8-1
read
input 11-66 - 11-70
pixel color (medium-res) 9-9
read a record 11-58 - 11-60
real
constants 6-7
data type 6-1 - 6-4
functions 7-8
number conversion 11-71
number range 6-2
number rounding 11-53
record 8-2
binary data 11-58
position 8-5
rectangle, drawing (high-res) 9-52 - 9-53, 9-60 - 9-61
reduce memory size 1-4
registers palette 9-35, 9-109 - 9-110
relational operators 7-5 - 7-6
relative storage area 3-3
REM command 11-135 - 11-136
remainder (division) 11-93
removing
disk files 11-30
procedures 3-6, 10-9, 11-72
spaces 11-172 - 11-173
RENAME command 3-1
renaming procedures 3-2
renumbering lines (editor) 4-2, 4-10
REPEAT/UNTIL
commands 11-137 - 11-139, 11-179
requesting memory 1-3 - 1-4
reset file pointer 11-148 - 11-149
RESTORE command 11-140
retrieving bytes from a file 8-5
RETURN command 11-141
returning
from subroutine 11-61 - 11-62
to OS-9 10-9
reverse video (high-res) 9-120
REVON command (high-res) 9-120
rewind a file 8-11
right brace 1-6
right bracket 1-6
RIGHT$ command 11-142
ring bell 9-54
RND command 11-143 - 11-144
ROOT directory 3-7
rounding a real number 11-53
RUN command 3-1, 6-8, 10-9, 11-145 - 11-147

SAVE command 3-1, 3-5, 10-9
saving
a window area 9-96 - 9-97
graphic images (high-res) 9-117 - 9-118
memory 1-2
procedures 3-1, 3-5
space by compiling 1-2
SCALESW command (high-res) 9-121 - 9-122
screen
alphanumeric 9-30
blink (high-res) 9-55
clearing (high-res) 9-64

screen (*cont'd*)
clearing (medium-res) 9-9, 9-17
color (medium-res) 9-26
display 11-122
format (medium-res) 9-26
formatting 11-180
location (medium-res) 9-20 - 9-21
resolution 9-31
selecting (medium-res) 9-13 - 9-14
switching (medium-res) 9-9
searching
for text (editor) 4-2, 4-9
in strings 11-164 - 11-165
seconds 11-27
SEEK command 11-148 - 11-149
select a window 9-32 - 9-33
SELECT command (high-res) 9-123 - 9-124
selecting memory 1-3
sending a carriage return 9-67
sentence-creating procedure 4-3
sequential file writes 11-185 - 11-186
SETDPTR command (high-res) 9-125
setting
a point (medium-res) 9-10, 9-28 - 9-29
border color (high-res) 9-58 - 9-59
color (medium-res) 9-18
pixel (high-res) 9-113 - 9-114
READ pointer 11-140
screen (medium-res) 9-9
SGN command 11-150 - 11-151
SHELL command 11-152 - 11-153
shell commands 10-9
SHIFT-← key sequence 1-6
SHIFT-BREAK key sequence 1-6
SHIFT-CLEAR key sequence 1-6
show text 11-119 - 11-121
sign of a value 11-150 - 11-151
simulating an error 11-45 - 11-46
SIN command 11-154
sine 11-154
single-dimènsioned array 6-9 - 6-10
SIZE command 11-155 - 11-156
size
data 3-2
memory 1-3
procedure 3-2
space
removing 11-172 - 11-173
saving by compiling 1-2
spaces in command lines 2-2
special keys 1-5 - 1-7
speed
of arithmetic functions 12-2
of execution 1-1
SQ command 11-157
SQR/SQRT commands 11-158
square root 11-158
starting
a shell in a window 9-36
BASIC09 1-2 - 1-4
STATE command (debug) 5-3
statements 10-7
status of joystick (medium-res) 9-22 - 9-23
STEP command 5-4, 11-159 - 11-160

step rate (debug) 5-4
stepping through
  procedures 5-5 - 5-6
STOP command 11-161
stop program execution 1-6
storage
    area of command
      lines 3-3
    minimization 12-1
    of variables 11-31 -
      11-33
storing
    data 11-25 - 11-26
    in memory 11-116 -
      11-117
STR$ command 11-162 -
  11-163
string
    constants 6-7
    data conversion 11-181 -
      11-182
    data type 6-1 - 6-2
    functions 10-7
    length 6-4, 11-77
    operators 7-6
    storage 6-5
    variables 6-4 - 6-5
strings
    appending 7-6
    portioning 11-76, 11-92,
      11-142
    searching 11-164
structured programming 1-1
structures, complex data
  8-11 - 8-15
subroutine
    commands 11-61 - 11-62
    jumps 11-100 - 11-101
SUBSTR command 11-164 -
  11-165
substrings 11-92
subtraction 7-3 - 7-4
suspending execution 11-112
switching screens (medium-
  res) 9-9, 9-13 - 9-14
symbolic debugging 5-1
syntax 11-1
system
    commands 3-1
    interfacing 1-1

TAB command 11-166 -
  11-167
TAN command 11-168
tangent 11-168
terminating
    a procedure 11-36 -
      11-37, 11-161
    the editor 4-2
test for end-of-file 11-42
text
    changing 4-2, 4-7 - 4-9
    characters (high-res)
      9-94
    display 11-119 - 11-121
    fonts 9-43 - 9-44
    formatting 11-122 -
      11-128
    high-resolution 9-42 -
      9-44
    proportional 9-115 -
      9-116
    searching 4-2, 4-9
    cursor commands (high-
      res) 9-48
three-dimension arrays 6-13
tilde 1-6
time 11-27 - 11-28
tracing
    execution 5-4 - 5-6,
      11-174
transcendental functions 10-7
trapping errors 11-97 - 11-99
TRIM$ command 11-172 -
  11-173
TROFF command (debug)
  5-4, 11-174
TRON command 5-4 - 5-6,
  11-174

TRUE command 11-175 - 11-176
turning off the cursor 9-71
two-dimension array 6-9
type
    conversion 6-6, 7-2
    mismatch 6-6
    of data 6-1 - 6-16, 10-8
    of file access 8-1
    of operators 7-3
TYPE command 8-12, 11-177 - 11-178

underscore 1-6
UNDLNOFF command (high-res) 9-126
UNDLNON command (high-res) 9-126
unequal 7-5
UNTIL 11-137 - 11-139, 11-180
up arrow 1-6
UPDATE 8-1, 8-4
USING command 11-180 - 11-182
using debug 5-4 - 5-5

VAL command 11-181 - 11-182
value
    absolute 11-4
    Boolean 11-51 - 11-52
    random 11-143 - 11-144
variable
    address 6-8, 11-6
    initialization 6-8
    local 6-7
    passing 6-8 - 6-9, 11-108 - 11-111
    size 11-155 - 11-156
    storage 11-31 - 11-33
    value of 11-78 - 11-79
variables 11-2
    assigning (debug) 5-3
    local 6-7
    string 6-4 - 6-5
vector 6-13
vertical bar 1-6
video
    address (medium-res) 9-9
    reverse (high-res) 9-120
visible cursor (high-res) 9-72

WCREATE command 9-33 - 9-34
WHILE/DO/ENDWHILE loop 11-34, 11-41, 11-180 - 11-181
whole number, range 6-2
wildcard
    editor 4-1
    using with commands 3-5
window
    area, saving 9-96 - 9-97
    commands (high-res) 9-45
    deallocating (high-res) 9-83 - 9-84
    defining (high-res) 9-86 - 9-87
    display 1-6, 9-123 - 9-124
    erasing 9-91
    establishing 9-32 - 9-41
    formats 9-34
    graphics 9-35 - 9-36
    hardware 9-32 - 9-35
    image (high-res) 9-101 - 9-102
    overlay (high-res) 9-107 - 9-108
    protect switch (high-res) 9-85
    shell 9-36
    working area (high-res) 9-76 - 9-77

windows
    defining 9-33 - 9-34
    from BASIC09 9-39 - 9-41
    overlay 9-41
    predefined 9-32 - 9-33
    with high-resolution 9-31
working area (high-res) 9-76
workspace 1-3, 3-1
WRITE command 11-185 - 11-186
writing
    a procedure 2-1 - 2-2
    to files 8-3, 11-129

XOR command 11-187 - 11-188
XOR operator 7-7

year 11-27

OS-9
Technical
Reference

Contents

Chapter 1 System Organization 1-1
I/O System Modules 1-1
Color Computer OS-9 Modules 1-2
Kernel, Clock Module, and INIT 1-2
Input/Output Modules 1-3
I/O Manager 1-3
File Managers 1-3
Device Drivers 1-3
Device Descriptors 1-4
Shell 1-4

Chapter 2 The Kernel 2-1
System Initialization 2-1
System Call Processing 2-4
OS9Defs and Symbolic Names 2-4
Types of System Calls 2-4
Memory Management 2-5
Memory Use 2-5
Color Computer OS-9 Typical Memory Map 2-7
Memory Management Hardware 2-7
Multiprogramming 2-12
Process Creation 2-12
Process States 2-13
Execution Scheduling 2-14
Signals 2-15
Interrupt Processing 2-16
Logical Interrupt Polling System 2-17
Virtual Interrupt Processing 2-19

Chapter 3 Memory Modules 3-1
Module Types 3-1
Module Format 3-1
Module Header 3-2
Module Body 3-2
CRC Value 3-2
Module Headers: Standard Information 3-3
Sync Bytes 3-3
Module Size 3-3
Offset to Module Name 3-3
Type/Language Byte 3-4
Attributes/Revision Level Byte 3-4
Header Check 3-5

Module Headers: Type-Dependent Information 3-5
Executable Memory Module Format 3-6

Chapter 4 OS-9's Unified Input/Output System 4-1
I/O System Modules 4-1
The I/O Manager 4-2
File Managers 4-3
File Manager Structure 4-3
Create, Open 4-4
Makdir 4-4
ChgDir 4-4
Delete 4-5
Seek 4-5
Read 4-5
Write 4-6
ReadLn 4-6
WriteLn 4-6
GetStat, PutStat 4-6
Close 4-7
Interfacing with Device Drivers 4-7
Device Driver Modules 4-8
Device Driver Module Format 4-10
OS-9 Interaction With Devices 4-11
Suspend State (Level Two Only) 4-13
Device Descriptor Modules 4-15
Path Descriptors 4-18

Chapter 5 Random Block File Manager 5-1
Logical and Physical Disk Organization 5-1
Identification Sector (LSN 0) 5-2
Disk Allocation Map Sector (LSN 1) 5-3
ROOT Directory 5-3
File Descriptor Sector 5-3
Directories 5-5
The RBF Manager Definitions of the Path Descriptor ..5-5
RBF-Type Device Descriptor Modules 5-8
RBF Record Locking 5-10
Record Locking and Unlocking 5-11
Non-Shareable Files 5-12
End-of-File Lock 5-12
Deadlock Detection 5-13
RBF-Type Device Driver Modules 5-13
The RBF Device Memory Area Definitions 5-13
RBF Device Driver Subroutines 5-16

Chapter 6 Sequential Character File Manager 6-1
SCF Line Editing Functions 6-1
Read and Write 6-1
Read Line and Write Line 6-2
SCF Definitions of the Path Descriptor 6-2
SCF-Type Device Descriptor Modules 6-6
SCF-Type Device Driver Modules 6-9
SCF Device Driver Subroutines 6-10

Chapter 7 The Pipe File Manager (PIPEMAN) 7-1

Chapter 8 System Calls 8-1
Calling Procedure 8-1
I/O System Calls 8-2
System Call Descriptions 8-2
User Mode System Calls Quick Reference 8-3
System Mode Calls Quick Reference 8-5
User System Calls 8-7
I/O User System Calls 8-44
Privileged System Mode Calls 8-66
Get Status System Calls 8-112
Set Status System Calls 8-130

Appendices A-1
A Memory Module Diagrams A-1
B Standard Floppy Disk Format B-1
C System Error Codes C-1

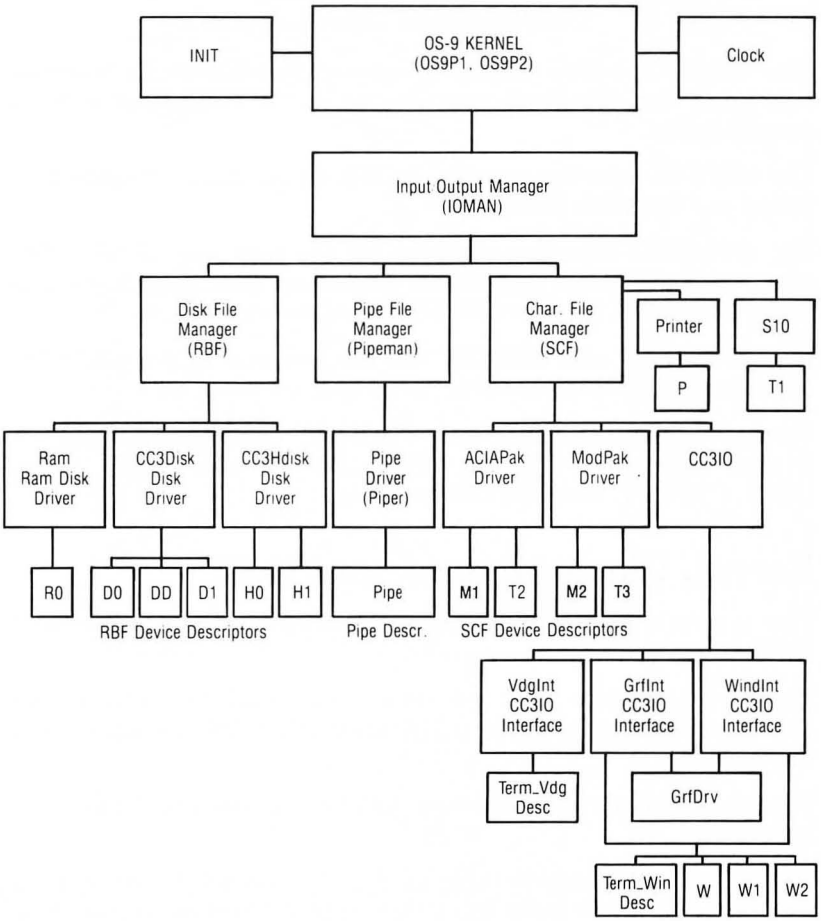
Index

Chapter 1

System Organization

OS-9 is composed of a group of modules, each of which has a specific function. The following illustration shows the major modules. Actual module names are capitalized.

I/O System Modules



OS-9 COMPONENT MODULE ORGANIZATION

Color Computer OS-9 Modules

IOMAN	Input/output management
INIT	System initialization table
CLOCK	Software routine time management
RBF	Random block file management
SCF	Sequential character file management
PIPEMAN	Pipe file management
CC3DISK	Color Computer disk driver
CC3IO	Color Computer input/output driver

The VDGINT (video display generator) provides both interface functions and low-level routines for Color Computer 2 VDG compatibility.

The GRFINT interface provides high-level graphics code interpretation and interface functions.

The WINDINT interface contains all the functions of GRFINT, along with additional support for Multiview functions. If you are using Multiview, exclude GRFINT from the system.

Both WINDINT and GRFINT use the low-level driver GRFDRV to perform the actual drawing on bitmap screens.

Term_VDG uses CC3IO and VDGINT. TERM_WIN and all window descriptors (W, W1, W2, and so on) use CC3IO, WINDINT, GRFINT, and GRFDRV modules.

Kernel, Clock Module, and INIT

The system's 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 OS-9 modules into the system.

The clock module is a software handler for the real-time clock hardware.

INIT is an initialization table used by the kernel during system startup. This table loads initial tasks and specifies initial table sizes and initial system device names. It is loaded into RAM (random access memory) by the OS-9 bootstrap module Boot. Boot also loads the OS9P2 and INIT modules during system startup.

There are two ways to run boot:

- Using the DOS command with Color Disk BASIC, Version 1.1, or later.
- Pressing the reset button after OS-9 is running.

Input/Output Modules

The remaining modules make up the OS-9 I/O system. They are defined briefly here and are discussed in detail in Chapter 4.

I/O Manager

The system's second level (the level below the kernel) contains the input/output manager, IOMAN. The I/O manager provides common processing for all input/output operations. It is required for performing any input/output supported by OS-9.

File Managers

The system's third level contains the *file managers*. File managers perform I/O request processing for similar classes of I/O devices. There are three file managers:

RBF manager — The random block file manager processes all disk I/O operations.

SCF manager — The sequential character file manager handles all non-disk I/O operations that operate one character at a time. These operations include terminal and printer I/O.

PIPEMAN — The pipe file manager handles *pipes*. Pipes are memory buffers that act as files. Pipes are used for data transfers between processes.

Device Drivers

The system's fourth level contains the *device drivers*. Device drivers handle basic I/O functions for specific I/O controller hardware. You can use pre-written drivers, or you can write your own.

Device Descriptors

The system's fifth level contains the *device descriptors*. Device descriptors are small tables that define the logical name, device driver, and file manager for each I/O port. They also contain port initialization and port address information. Device descriptors require only one copy of each I/O controller driver used.

Shell

The shell is the command interpreter. It is a program and not a part of the operating system. The shell is fully described in the *OS-9 Commands* manual.

Chapter 2

The Kernel

The kernel is the core of OS-9. The kernel supervises the system and manages system resources. Half of the kernel (called OS9P1) resides in the boot module. The other half of the kernel (called OS9P2) is loaded into RAM with the other OS-9 modules.

The kernel's main functions are:

- System initialization after reset
- Service request processing
- Memory management
- Multiprogramming management
- Interrupt processing

I/O functions are not included in the list because the kernel does not directly process them. Instead, it passes I/O system calls to the I/O Manager for processing.

System Initialization

After a hardware reset, the kernel initializes the system. This involves:

1. Locating modules loaded in memory from the OS-9 Boot file.
2. Determining the amount of available RAM.
3. Loading any required modules that were not loaded from the OS-9 Boot file.

OS-9 Level Two cannot install new system calls using the OS-9 Level One system call F$SSvc. F$SSvc does not work with a Level Two user program because of the separation of system and user address space.

OS9P3 can be used to tailor the system to fit specific needs. The following listing is an example of how to use the OS9P3 module.

```
Microware OS-9 Assembler 2.1  11/18/83    16:06:01   Page 001

OS-9 Level TWO V1.2, part 2 - OS-9 System Symbol Definitions

00001
00002
00003

00011  ************************************************************************
00012  *
00013  *                Module Header
00014  *
00015  00C1             Type      set   Systm+Objct
00016  0081             Revs      set   ReEnt+1
00017  0000  87CD005E             mod   OS9End,OS9Name,Type,Revs,Cold,256
00018  000D  4F533970   OS9Name   fcs   "OS9p3"
00019
00029  0012  01                   fcb   1  edition number
00030                              use   defsfile
00031  0002             level     equ   2
00032                              opt   -c
0033                               opt   f
00041
00042  ************************************************************
00043  *
00044  *                Routine Cold
00045  *
00046  *
00047
00048  0013  318D0004   Cold      leay  SvcTbl,pcr   get service routine
00049  0017  103F32               OS9   F$SSvc   install new service
00050  001A  39                   rts
00051
00052
00053  ************************************************************
00054  *
00055  *                Service Routines Initialization Table
00056  *
00057
00058  0025             F$SAYHI   equ   $25 set up new call
00059  *      Add this to the user os9defs file.
00060
```

```
00061   001B             SvcTbl    equ   *
00062   001B   25                 fcb   F$SAYHI
00063   001C   0001               fdb   SayHi-*-2
00064   001E   80                 fcb   $80


Microware OS-9 Assembler 2.1  11/18/83   16:06:01   Page 002
OS-9 Level TWO V1.2, part 2 - OS-9 System Symbol Definitions

00068           *
00069           *Service call Say Hello to user
00070           *
00071           *Input:   U = Registers ptr
00072           *             R$X,u = Message ptr (if 0 send default)
00073           *             Max message length = 40 bytes.
00074           *
00075           *Output:  Message sent to standard error path of user.
00076           *
00077           *Data:    D.Proc
00078           *
00079
00080   001F   AE44      SayHi     ldx   R$X,u       get mess. address
00081   0021   2619                bne   SayHi6      bra if not default
00082   0023   109E50              ldy   D.Proc      get proc descr ptr
00083   0026   EE24                ldu   P$SP,y      get caller's stack
00084   0028   33C8D8              leau  -40,u       room for message
00085   002B   96D0                lda   D.SysTsk    system's task num
00086   002D   E626                ldb   P$TASK,y    caller's task num
00087   002F   108E0028            ldy   #40         set byte count
00088   0033   308D0012            leax  Hello,pcr   destination ptr
00089   0037   103F38              OS9   F$Move      mess into user mem
00090   003A   30C4                leax  0,u
00091   003C   108E0028  SayHi6    ldy   #40         get max byte count
00092   0040   DE50                ldu   D.Proc      get proc desc ptr
00093   0042   A6C832              lda   P$PATH+2,u  path num of stderr
00094   0045   103F8C              OS9   I$WritLn    write mess line
00095   0048   39                  rts
00096
00097   0049   48656C6C  Hello    fcc   "Hello there user."
00098   005A   0D                 fcb   $D
00099
00100   005B   5104B6              emod        module CRC
00101
```

```
00102  005E            OS9End    equ    *
00103
00104                                    end

00000 error(s)
00000 warning(s)
$005E 00094 program bytes generated
$0000 00000 data bytes allocated
$2884 10372 bytes used for symbols
```

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. You usually pass parameters for system calls in the 6809 registers.

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 equivalent to:

```
SWI2
FCB I$Read
```

The OS-9 assembly macro OS9 generates an SWI2 function. The label I$Read is the label for the code $89.

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. The kernel passes calls of this type to the I/O manager 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$. For example, the Link function call is called F$Link.

The function calls include *user calls* and privileged *system mode* calls. (See Chapter 8, "System Calls", 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 of all the programs in memory. The directory contains information about each module, including its name and address and the number of processes using it. The number of processes using a module is called the module's *link count.*

When a module's link count is zero, OS-9 deallocates its part of memory and removes its name from the module directory.

Memory modules are the foundation of OS-9's modular software environment. Advantages of memory management are:

- Automatic runtime linking of programs to libraries of utility modules
- Automatic sharing of re-entrant programs
- Replacement of small sections of large programs into memory for update or correction

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.

OS-9 pools all other RAM into a free memory space. As the system allocates or deallocates memory, it dynamically takes it from or returns it to this pool. RAM does not need to be contiguous because the memory management unit can dynamically rearrange memory addresses.

The basic unit of memory allocation is the 256-byte *page*. OS-9 always allocates memory in whole numbers of pages.

The data structure that OS-9 Level Two uses to keep track of memory allocation is a 256-byte *bit map*. Each bit in this table is associated with a specific page of memory. A cleared bit indicates that the page is free and available for assignment. A set bit indicates that the page is in use (that no RAM is free at that address). OS-9 Level Two always allocates memory in 8192-byte increments. This is the smallest memory block that the memory management hardware supports.

OS-9 automatically allocates memory when any of the following occurs:

- Program modules are loaded into RAM
- Processes are created
- Processes execute system calls to request additional RAM
- OS-9 needs I/O buffers or larger tables

OS-9 also has inverse functions to deallocate memory allocated to program modules, new processes, buffers, and tables.

In general, memory for program modules and buffers is allocated from high addresses downward. Memory for process data areas is allocated from low addresses upward.

Following, is a memory map of a typical system. Actual memory sizes and addresses can vary depending on the exact system configuration.

Color Computer OS-9 Typical Memory Map

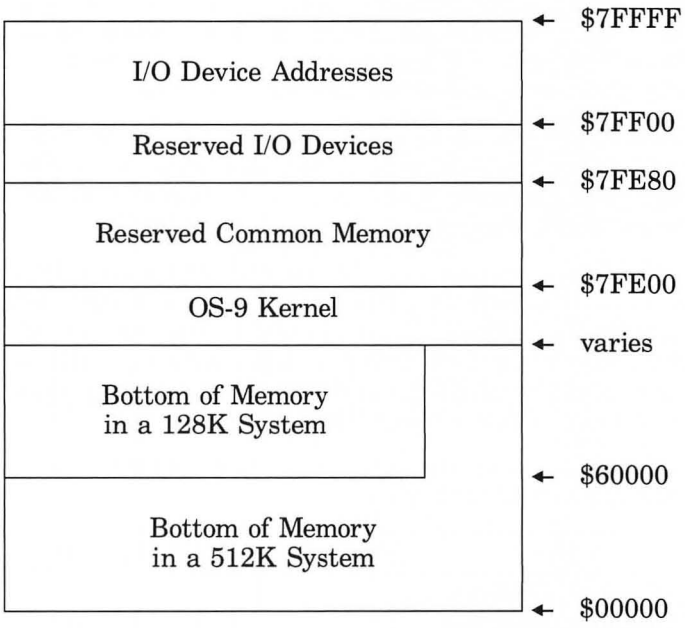


Figure 2.1

Note: The high two pages of every logical address space contain the defined areas I/O Device Addresses, Reserved I/O Devices, and Reserved Common Memory.

Memory Management Hardware

The 8-bit CPU in the Color Computer 3 can directly address only 64 kilobytes of memory using its 16 address lines (A0-A15). The Color Computer 3's Memory Management Unit (MMU) extends the addressing capability of the computer by increasing the address lines to 19 (A0-A18). This lets the computer address up to 512 kilobytes of memory ($0-$7FFFF).

The 512K address space is called the *physical address space*. The physical address space is subdivided into 8K *blocks*. The six high order address bits (A13-A18) define a *block number*.

OS-9 creates a *logical address space* of up to 64K for each task by using the FORK system call. Even though the memory within a logical address space appears to be contiguous, it might not be—the MMU translates the physical addresses to access available memory. Address spaces can also contain blocks of memory that are common to more than one map.

The MMU consists of a multiplexer and a 16 by 6-bit RAM array. Each of the 6-bit elements in this array is an MMU task register. The computer uses these task registers to determine the proper 8-kilobyte memory segment to address.

The MMU task registers are loaded with addressing data by the CPU. This data indicates the actual location of each 8-kilobyte segment of the current system memory. The task registers are divided into two sets consisting of eight registers each. Whether the task register select bit (TR bit) is set or reset, determines which of the two sets is to be used.

The relation between the data in the task register and the generated addresses is as follows:

Bit	D5	D4	D3	D2	D1	D0
Corresponding Memory Address	A18	A17	A16	A15	A14	A13

Figure 2.2

When the CPU accesses any memory outside the I/O and control range (XFF00 = XFFFF), the CPU address lines (A13-A15) and the TR bit determine what segment of memory to address. This is done through the multiplexer when SELECT is low. (See the following table.)

When the CPU writes data to the MMU, A0-A3 determine the location of the MMU register to receive the incoming data when SELECT is high. The following diagram illustrates the operation of the Color Computer 3's memory management:

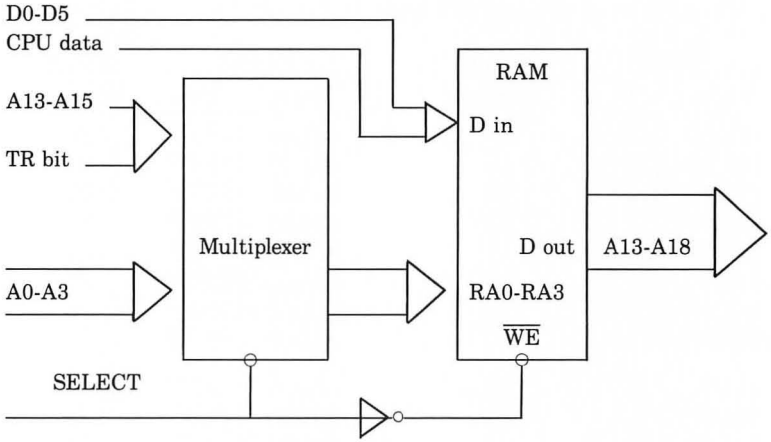


Figure 2.3

The system uses the data from the MMU registers to determine the block of memory to be accessed, according to the following table:

TR Bit	A15	A14	A13	AddressRange	MMU Address
0	0	0	0	X0000-X1FFF	FFA0
0	0	0	1	X2000-X3FFF	FFA1
0	0	1	0	X4000-X5FFF	FFA2
0	0	1	1	X6000-X7FFF	FFA3
0	1	0	0	X8000-X9FFF	FFA4
0	1	0	1	XA000-XBFFF	FFA5
0	1	1	0	XC000-XDFFF	FFA6
0	1	1	1	XE000-XFFFF	FFA7
1	0	0	0	X0000-X1FFF	FFA8
1	0	0	1	X2000-X3FFF	FFA9
1	0	1	0	X4000-X5FFF	FFAA
1	0	1	1	X6000-X7FFF	FFAB
1	1	0	0	X8000-X9FFF	FFAC
1	1	0	1	XA000-XBFFF	FFAD
1	1	1	0	XC000-XDFFF	FFAE
1	1	1	1	XE000-XFFFF	FFAF

Figure 2.4

The translation of physical address to 8K-blocks is as follows:

Range		Block	Range		Block
From	To	Number	From	To	Number
00000	01FFF	00	40000	41FFF	20
02000	03FFF	01	42000	43FFF	21
04000	05FFF	02	44000	45FFF	22
06000	07FFF	03	46000	47FFF	23
08000	09FFF	04	48000	49FFF	24
0A000	0BFFF	05	4A000	4BFFF	25
0C000	0DFFF	06	4C000	4DFFF	26
0E000	0FFFF	07	4E000	4FFFF	27
10000	11FFF	08	50000	51FFF	28
12000	13FFF	09	52000	53FFF	29
14000	15FFF	0A	54000	55FFF	2A
16000	17FFF	0B	56000	57FFF	2B
18000	19FFF	0C	58000	59FFF	2C
1A000	1BFFF	0D	5A000	5BFFF	2D
1C000	1DFFF	0E	5C000	5DFFF	2E
1E000	1FFFF	0F	5E000	5FFFF	2F
20000	21FFF	10	60000	61FFF	30
22000	23FFF	11	62000	63FFF	31
24000	25FFF	12	64000	65FFF	32
26000	27FFF	13	66000	67FFF	33
28000	29FFF	14	68000	69FFF	34
2A000	2BFFF	15	6A000	6BFFF	35
2C000	2DFFF	16	6C000	6DFFF	36
2E000	2FFFF	17	6E000	6FFFF	37
30000	31FFF	18	70000	71FFF	38
32000	33FFF	19	72000	73FFF	39
34000	35FFF	1A	74000	75FFF	3A
36000	37FFF	1B	76000	77FFF	3B
38000	39FFF	1C	78000	79FFF	3C
3A000	3BFFF	1D	7A000	7BFFF	3D
3C000	3DFFF	1E	7C000	7DFFF	3E
3E000	3FFFF	1F	7E000	7FFFF	3F

Figure 2.5

In order for the MMU to function, the TR bit at $FF90 must be cleared and the MMU must be enabled. However, before doing this, the address data for each memory segment must be loaded into the designated set of task registers. For example, to select a standard 64K map in the top range of the Color Computer 3's 512K RAM, with the TR bit set to 0, the following values must be preloaded into the MMU's registers:

MMU Location Address	Data (Hex)	Data (Bin)	Address Range
FFA0	38	111000	70000-71FFF
FFA1	39	111001	72000-73FFF
FFA2	3A	111010	74000-75FFF
FFA3	3B	111011	76000-77FFF
FFA4	3C	111100	78000-79FFF
FFA5	3D	111101	7A000-7BFFF
FFA6	3E	111110	7C000-7DFFF
FFA7	3F	111111	7E000-7FFFF

Figure 2.6

Although this table shows MMU data in the range $38 to 3F, any data between $0 and $3F can be loaded into the MMU registers to select memory addresses in the range 0 to $7FFFF, as illustrated by Figure 2.5.

Normally, the blocks containing I/O devices are kept in the system map, but not in the user maps. This is appropriate for time-sharing applications, but not for process control. To directly access I/O devices, use the F$MspBlk system call. This call takes a starting block number and block count, and maps them into *unallocated* spaces of the process's address space. The system call returns the logical address at which the blocks were inserted.

For example, suppose a display screen in your system is allocated at extended addresses $7A000-$7DFFF (blocks 3D and 3E). The following system call maps them into your address space:

```
ldb     #2          number of blocks
ldx     #3D         starting block number
os9     F$MapBlk    call MapBlk
stu     IOPorts     save address where mapped
```

On return, the U register contains the starting address at which the blocks were switched. For example, suppose that the call returned $4000. To access extended address $7A020, write to $4020.

Other system calls that copy data to or from one task's map to another are available, such as F$STABX and F$Move. Some of these calls are system mode privileged. You can unprotect them by changing the appropriate bit in the corresponding entry of the system service request table and then making a new system boot with the patched table.

Multiprogramming

OS-9 is a multiprogramming operating system. This means that several independent programs called *processes* can be executed at the same time. By issuing the appropriate system call to OS-9, each process can have access to any system resource.

Multiprogramming functions use a hardware real-time clock. The clock generates interrupts 60 times per second, or one every 16.67 milliseconds. These interrupts are called ticks.

Processes that are not waiting for some event are called *active processes*. OS-9 runs active processes for a specific system-assigned period called a time slice. The number of time slices per minute during which a process is allowed to execute depends on a process's 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*).

Finding 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 in the execution directory, with the requested module name as a filename.

Assigning 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 (see the following section "Process States"), 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 to allocate RAM for the process. The primary module's header contains a storage size. OS-9 uses this size unless the Fork system call requests a larger area. OS-9 then attempts to allocate a memory area of the specified size from the free memory space. The memory space does not need to be contiguous.

Proceed or Terminate. 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 terminates 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 its 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 until it receives a signal. A child process is a process that is started (execution is begun by) another process—a *parent process*.

- Sleeping—The process is suspended for a specific period of time or until it receives a signal.

Each state has its own queue, a linked list of *descriptors* of processes in that state. To change a process's state, move its descriptor to another queue.

The Active State. Each active process is given a time slice for execution, according to its priority. The scheduler in the kernel ensures 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 terminates or until it receives a *signal.* (See the "Signals" section later in this chapter.)

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 wakeup 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 the number of process switches that have occurred since the process's 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 increase by one time slice count. When the executing process's 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 increase by one. Ages never go beyond 255.

A new active process that was terminated while in the system state is an exception. This process is given high priority because it is usually executing critical routines that affect shared system resources.

When there are no active processes, the kernel handles the next interrupt and then executes a CWA1 instruction. This procedure decreases interrupt latency time (the time it takes the system to process an interrupt).

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 then 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.

A signal can convey status information in the form of a 1-byte numeric value. Some *signal codes* (values) are predefined, but you can define most. The signal codes are:

0	=	Kill (terminates the process, is non-interceptable)
1	=	Wakeup (wakes up a sleeping process)
2	=	Keyboard terminate
3	=	Keyboard interrupt
4	=	Window change
128-255	=	User defined

When a signal is sent to a process, the signal is saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. When the process 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 the process has set up a signal intercept trap, the process resumes execution at the address given in the Intercept call. The signal code passes to this routine. Terminate the routine with an RTI instruction to resume normal execution of the process.

Note: A wakeup signal activates a sleeping process. It sets a flag but ignores the call to branch to the intercept routine.

If it has not set up a signal intercept trap, the process is terminated immediately. It is also terminated if the signal code is zero. If the process is in the system mode, OS-9 defers the termination. The process dies upon return to the user state.

A process can have a signal pending (usually because the process has not been assigned a time slice since receiving the signal). If it does, and another process tries to send it another signal, the new signal is terminated, and the Send system call returns an error. To give the destination process time to process the pending signal, the sender needs to execute a Sleep system call for a few ticks before trying to send the signal again.

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.

The SWI, SWI2, and SWI3 vectors point to routines that read the corresponding pseudo vector from the process's 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.

Hardware Vector Table

Vector	Address
SWI3	$FFF2
SWI2	$FFF4
FIRQ	$FFF6
IRQ	$FFF8
SWI	$FFFA
NMI	$FFFC
RESTART	$FFFE

FIRQ Interrupt. The system uses the FIRQ interrupt. The FIRQ vector is not available to you. The FIRQ vector is reserved for future use. Only one FIRQ generating device can be in the system at a time.

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.

IRQ Interrupt. Most OS-9 I/O devices generate IRQ interrupts. The IRQ vector points to the real-time clock and the keyboard scanner routines. These routines, in turn, jump to a special IRQ polling system that 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.

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 six variables:

Polling Address — Points to the status register of the device. The register must have a bit or bits that indicate if 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.

Mask Byte	Selects one or more interrupt request flag bits within the device status register. The bits identify the active task or device.
Service Routine Address	Points to the interrupt service routine for the device. 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 polled (a number from 0 to 255).

Polling the Entries. When an IRQ interrupt occurs, OS-9 enters the polling system via the corresponding RAM interrupt vector. It starts polling the devices in order of priority. OS-9 loads the status register address of each entry into Accumulator A, using the device address from the table.

OS-9 performs an exclusive-OR operation using the flip byte, followed by a logical-AND operation using the mask byte. If the result is non-zero, OS-9 assumes that the device is the source of the interrupt.

OS-9 reads the device memory address and service routine address from the table, and performs the interrupt service routine.

> **Note:** If you are writing your own device driver, terminate the interrupt service routine with an RTS instruction, **not** an RTI instruction.

Adding Entries to the Table. You can make entries to the IRQ (interrupt request) polling table by using the Set IRQ system call (F$IRQ). Set IRQ is a *privileged system call*, OS-9 can execute it only in the system mode. OS-9 is in system mode whenever it is running a device driver.

> **Note:** The code for the interrupt polling system is located in the I/O Manager module. The OS9P1 and OS9P2 modules contain the physical interrupt processing routines.

Virtual Interrupt Processing

A virtual IRQ, or VIRQ, is useful with devices in Multi-Pak expansion slots. Because of the absence of an IRQ line from the Multi-Pak interface, these devices cannot initiate physical interrupts. VIRQ enables these devices to act as if they were interrupt driven. Use VIRQ only with device driver and pseudo device driver modules. VIRQ is handled in the Clock module, which handles the VIRQ polling table and installs the F$VIRQ system call. Since the F$VIRQ system call is dependent on clock initialization, the CC3GO module forces the clock to start.

The virtual interrupt is set up so that a device can be interrupted at a given number of clock ticks. The interrupt can occur one time, or can be repeated as long as the device is used.

The F$VIRQ system call installs VIRQ in a table. This call requires specification of a 5-byte packet for use in the VIRQ 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 re-installed in the table after being issued

F$VIRQ also specifies an initial tick count for the interrupt. The actual call is summarized here and is described in detail in Chapter 8.

Call:	OS9 F$VIRQ
Input:	(Y) = *address of 5-byte packet*
	(X) = 0 to delete entry, 1 to install entry
	(D) = *initial count value*
Output:	none
	(CC) carry set on error
	(IS) *appropriate error code*

The 5-byte packet is defined as follows:

Name	Offset	Function
Vi.Cnt	$0	Actual counter
Vi.Rst	$2	Reset value for counter
Vi.Stat	$4	Status byte

Two of the bits in the status byte are used. These are:

Bit 0 - set if VIRQ occurs
Bit 7 - set if a count reset is required

When making an F$VIRQ call, the packet might require initialization with a reset value. Bit 7 of the status byte must be either set or cleared 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 OS-9 processes the call, it writes the packet address into the VIRQ table.

At each clock tick, OS-9 scans the VIRQ table and subtracts one from each timer value. When a timer count reaches zero, OS-9 performs the following actions:

1. Sets Bit 0 in the status byte. This specifies a Virtual IRQ.
2. Checks Bit 7 of the status byte for a count reset request.
3. If bit 7 is set, resets the count using the reset value. If bit 7 is reset, deletes the packet address from the VIRQ table.

When a counter reaches zero and makes a virtual interrupt request, OS-9 runs the standard interrupt polling routine and services the interrupt. Because of this, you must install entries on both the VIRQ and DIRQ polling tables whenever you are using a VIRQ.

Unless the device has an actual physical interrupt, install the device on the IRQ polling table via the F$IRQ system call before placing it on the VIRQ table.

If the device has a physical interrupt, use the interrupt's hardware register address as the polling address for the F$IRQ call. After setting the polling address, set the flip and mask bytes for the device, and make the F$IRQ call.

If the device is totally VIRQ-driven, and has no interrupts, use the status byte from the VIRQ packet as the status byte. Use a mask byte of %00000001, defined as Vi.IFlag in the defs file. Use a flip byte value of 0. The following examples show how to set up both types of VIRQ calls. The first example is taken from an ACIA-type driver that has a physical interrupt found in a status register, but that cannot be accessed by the processor if used in the Multi-Pak. The second example is for a device with no physical interrupt handling, all interrupts are handled through the VIRQ.

```
* VIRQ Example #1 - Device Driver possessing real IRQ's

     *    Copyright 1985,1986 by Microware Systems
     *    Corporation. Reproduced Under License

     use defsfile

     * actual mask byte for hardware interrupt
     IRQReq set %10000000 Interrupt Request
       .
       .

     * offset to the actual hardware status register
     Status equ 1

     * VIRQ countdown value
     VIRQCNT equ 1      do the VIRQ on every tick
       .
       .

     ********************
     * Static storage offsets
     *
      org V.SCF room for scf variables
       .
       .

     VIRQBUF rmb 5 buffer for fake interrupt from clock
       .
       .
     MEM equ . Total static storage requirement

     ***************
     * Module Header
      mod MEND,NAM,DRIVR+OBJCT,REENT+1,ENT,MEM
      fcb UPDAT.

      fcb Edition Current Revision
     *************************
     * Driver entry jump table
     ENT lbra INIT
      lbra READ
      lbra WRITE
      lbra GETSTA
```

```
 lbra PUTSTA
  bra TRMNAT

* Actual mask information for F$IRQ call for the
* hardware interrupt MASK fcb 0 no flip bits
* fcb IRQReg Irq polling mask
* fcb 10 (higher) priority

****************
* Init
*   Initialize the device
*   Includes setting up the IRQ and VIRQ entries
*
INIT
   .
   .

* Install IRQ polling Table Entry first
*   Use the hardware status register and the hardware
*   mask
 ldd V.PORT,U get port address in D
 add #Status point to hardware status byte
 leax MASK,PCR get the hardware interrupt mask
 leay MIRQ,PCR address of interrupt service routine
 OS9 F$IRQ Add to IRQ polling table
 bcs INIT9 error - return it

* Install VIRQ in Clock Module second
*
 leay VIRQBUF,U get the 5 byte VIRQ buffer pointer
 lda #$80 get reset flag for repeated VIRQ's
 sta Vi.Stat,y put it into buffer
 ldd #VIRQCNT get count for number of ticks for the VIRQ
 std Vi.Rst,y put in initial reset value
 ldx #1 put onto table
 os9 F$VIRQ make the service request
 bcs INIT9 Error - return it
   .
   .

INIT9 rts

READ
   .
```

```
    .
WRITE
GETSTA
PUTSTA
    .
    .

***************
* Subroutine TRMNAT
*    Terminate device, including removal from tables
*
TRMNAT
    .
    .
* remove from VIRQ table first
 ldx #0 remove from VIRQ table
 leay VIRQBUF,U get address
 os9 F$VIRQ remove modem from VIRQ table
* next remove from IRQ table
 ldx #0
 OS9 F$IRQ remove modem from polling tbl
 rts

********************
* MIRQ
*   process Interrupt
*
MDIRQ
    .
    .
<  actual interrupt service routine  >
    .
 rts

 emod Module Crc
MEND equ *


* VIRQ Example #2 - Device Driver without hardware interrupts

***************
* STATIC STORAGE DEFINITION
*
```

```
*
  .
  .

VIRQBF rmb 5 buffer for VIRQ
DMEM equ .

******************
* Module Header
*
 mod DEND,DNAM,DRIVR+OBJCT,REENT+REV,DENT,DMEM
 fcb UPDAT. mode byte

 fcb 3 EDITION BYTE


* Driver entry table
DENT lbra INIT initialize
 lbra READ
 lbra WRITE
 lbra GETSTAT get status
 lbra SETSTAT set status
 lbra TERM terminate

* Mask information packet for F$IRQ call
* NOTE: uses the virtual interrupt flag, Vi.IFlag, for
* the maskbyte
*
DMSK fcb 0 no flip bits
 fcb Vi.IFlag polling mask for VIRQ
 fcb 10 priority

*****************
* INITIALIZE STORAGE AND CONTROLLER
* Includes setting up the IRQ and VIRQ table entries
*
INIT

* set up IRQ table entry first
* NOTE: uses the status register of the VIRQ buffer for
* the interrupt status register since no hardware status
* register is available
*
 leay VIRQBF+Vi.Stat,U get address of status byte
```

```
 tfr y,d put it into D reg
 leay DIRQ,PCR get address of interrupt routine
 leax DMSK,PCR get VIRQ mask info
 os9 F$IRQ install onto table
 bcs INIT9 exit on error

* now set up the VIRQ table entry
 leay VIRQBF,U point to the 5-byte packet
 lda #$80 get the reset flag to repeat VIRQ's
 sta Vi.Stat,y save it in the buffer
 ldd #VIRQCNT get the VIRQ counter value
 std Vi.Rst,y save it in the reset area of buffer
 ldx #1 code to install the VIRQ
 os9 F$VIRQ install on the table
 bcs INIT9 exit on error
   .
   .

INIT9 rts
  .
  .

 READ
 WRITE
 GETSTAT
 PUTSTAT
  .
************************
* TERM - terminate the device and remove entries from
* tables
TERM
  .
* remove from VIRQ table first
 ldx #0 get zero to remove from table
 leay VIRQBF,U get address of packet
 os9 F$VIRQ
* then remove from IRQ table
 ldx #0 get zero to remove from table
 os9 F$IRQ
  .
  .
  rts

************************************************************
```

```
* DIRQ - interrupt service routine
*
* NOTE: The service routine must be sure to reset the
* status byte of the VIRQ packet so that the interrupt
* looks as if it is cleared.
*
DIRQ
  .
  .
 lda VIRQBF+Vi.Stat,U get status byte
 anda #$FF-Vi.IFlag mask off interrupt bit
 sta VIRQBF+Vi.Stat,U put it back
   .
   .
  rts

 EMOD
DEND equ   *

END
```

Chapter 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 individually named *object.*

You also learned that each program 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 has a different use and function. These are the main requirements of a module:

- It cannot modify itself.
- It must be position-independent so that 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 can 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).

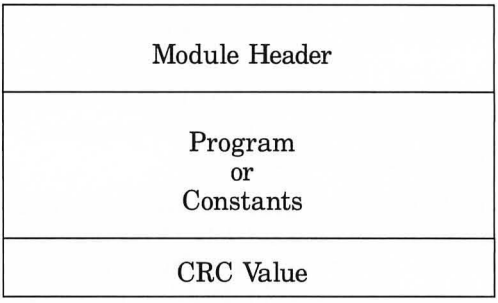


Figure 3.1

Module Header

At the beginning of the module (the 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 information includes the following:

- Size
- Type (machine code, BASIC09 compiled code, and so on)
- Attributes (executable, re-entrant, and so on)
- Data storage memory requirements
- Execution starting address

Usually, you do not need to write routines to generate the modules and headers. All OS-9 programming languages automatically create modules and headers.

Module Body

The module body contains the program or constants. It usually is pure code. The module name string is included in this area. Figure 3.2 provides the offset values for calculating the location of a module's name. (See "Offset to Module Name".)

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.

When the system first loads the module into memory, it performs a 25-bit CRC over the entire module, from the first byte of the module header to the byte immediately before the CRC. 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
$07	Attributes/Revision level
$08	Header check

Figure 3.2

Sync Bytes

The 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

The module size specifies the size of the module in bytes (includes CRC).

Offset to Module Name

The 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. It consists of a string of ASCII characters with the most significant bit set on the last character.

Type/Language Byte

The 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 (0 is not a legal type code):

Code	Module Type	Name
$1x	Program module	Prgrm
$2x	Subroutine module	Sbrtn
$3x	Multi-module (for future use)	Multi
$4x	Data module	Data
$5x-$Bx	User-definable module	
$Cx	OS-9 system module	Systm
$Dx	OS-9 file manager module	FlMgr
$Ex	OS-9 device driver module	Drivr
$Fx	OS-9 device descriptor module	Devic

Figure 3.3

The four least significant bits of Byte 6 indicate the language (denoted by x in the previous Figure). The language codes are given here:

Code	Language
$x0	Data (non-executable)
$x1	6809 object code
$x2	BASIC09 I-code
$x3	PASCAL P-code
$x4-$xF	Reserved for future use

Figure 3.4

By checking the language type, high-level language runtime systems can verify that a module is the correct type before attempting execution. BASIC09, for example, can run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

Attributes/Revision Level Byte

The attributes/revision level byte defines 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 re-entrant and, therefore, *shareable.*

The four least significant bits of this byte are a revision level in the range 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 its link count goes to zero.

Header Check

The header check byte contains 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 a module header. The layout and meaning vary, depending on the module type.

Module types $Cx-$Fx (system module, file manager module, device driver module, and device descriptor module) are used only by OS-9. Their formats are given later in the manual.

Module types $1x through $Bx have a general-purpose executable format. This format is often used in programs called by F$Fork or F$Chain. Here is the format used by these module types:

Executable Memory Module Format

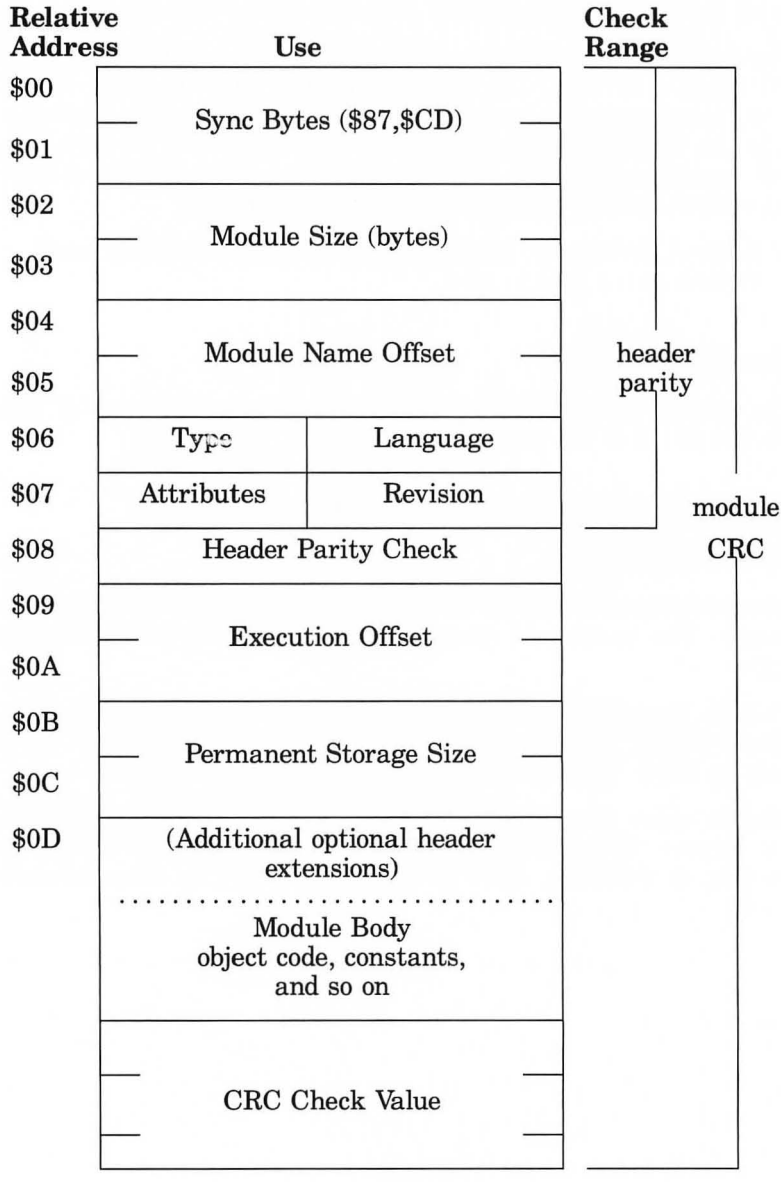


Figure 3.5

As you can see from the preceding chart, the executable memory has four 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) might 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's data area.

If the module is not 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 re-entrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

When OS-9 starts after a single 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 OS-9 detects the header sync code, it checks to see if the header 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.

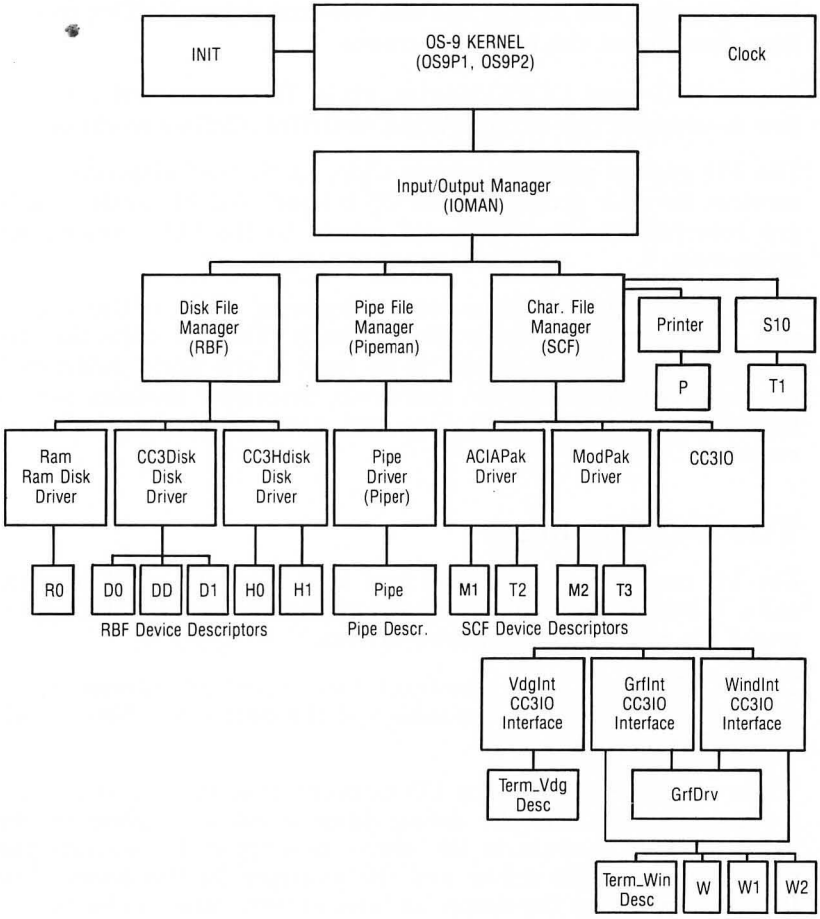
After the module search, OS-9 links to the component modules it found. This is the secret to OS-9's ability to adapt to almost any 6809 computer. It automatically locates its required and optional component modules and rebuilds the system each time it is started.

Chapter 4

OS-9's Unified Input/Output System

Chapter 1 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



OS-9 COMPONENT MODULE ORGANIZATION

The VDG Interface performs both interface and low level routines for VDG Color Computer 2 compatible modes and has limited support for high res screen allocation.

The GrfInt Interface provides the standard code interpretations and interface functions.

The WindInt Interface, available in the Multi-view package, contains all the functionality of GrfInt, along with additional support features. If you use WindInt, do not include GrfInt.

Both WindInt and GrfInt use the low-level driver GrfDrv to perform drawing on the bit-map screens.

Term_VDG uses CC3IO/VdgInt while Term_win and all window descriptors use CC3IO/(WindInt/GrfInt)/GrfDrv modules.

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 the I/O manager for processing.

The I/O manager 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.

The I/O Manager

The I/O manager provides the first level of service of I/O system calls. It routes data on I/O process paths to and from the appropriate file managers and device drivers.

The I/O Manager also maintains two important internal OS-9 data structures—the *device table* and the *path table*. Never modify the I/O manager.

When a path is opened, the I/O manager 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. The I/O manager 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 serves.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They can also monitor and process the data stream, for example, adding line-feed characters after carriage-return characters.

The file managers are re-entrant. The three standard OS-9 file managers are:

- Random block file manager: The RBF manager supports random-access, block-structured devices such as disk systems and bubble memories. (Chapter 5 discusses the RBF manager in detail.)
- Sequential Character File Manager: The SCF manager supports single-character-oriented devices, such as CRTs or hardcopy terminals, printers, and modems. (Chapter 6 discusses SCF in detail.)
- Pipe File Manager (PIPEMAN): The pipe manager supports interprocess communication via *pipes*.

File Manager Structure

Every file manager must have a branch table in exactly the following format. Routines that are not used by the file manager must branch to an error routine, that sets the carry and loads Register B with an appropriate error code before returning. Routines returning without error must ensure that the carry bit is clear.

```
* All routines are entered with:
* (Y) = Path Descriptor pointer
* (U) = Caller's register stack pointer

EntryPt equ *
 lbra Create
 lbra Open
 lbra MakDir
 lbra ChgDir
 lbra Delete
 lbra Seek
 lbra Read
 lbra Write
 lbra ReadLn
 lbra WriteLn
 lbra GetStat
 lbra PutStat
 lbra Close
```

Create, Open

Create and Open handle file creating and opening for devices. Typically, the process involves allocating any required buffers, initializing path descriptor variables, and establishing the path name. If the file manager controls multi-file devices (RBF), directory searching is performed to find or create the specified file.

Makdir

Makdir creates a directory file on multi-file devices. Makdir is neither preceded by a Create nor followed by a Close. File managers that are incapable of supporting directories need to return carry set with an appropriate error code in Register B.

ChgDir

On multi-file devices, ChgDir searches for a directory file. If ChgDir finds the directory, it saves the address of the directory (up to four bytes) in the caller's process descriptor. The descriptor is located at P$DIO+2 (for a data directory) or P$DIO+8 (for an execution directory).

In the case of the RBF manager, the address of the directory's file descriptor is saved. Open/Create begins searching in the current directory when the caller's pathlist does not begin with a slash (/). File managers that do not support directories should return the carry set and an appropriate error code in Register B.

Delete

Multi-file device managers handle file delete requests by initiating a directory search that is similar to Open. Once a device manager finds the file, it removes the file from the directory. Any media in use by the file are returned to unused status. In the case of the RBF manager, space is returned for system use and is marked as available in the free cluster bit map on the disk. File managers that do not support multi- file devices return an error.

Seek

File managers that support random access devices use Seek to position file pointers of an already open path to the byte specified. Typically, the positioning is a logical movement. No error is produced at the time of the seek if the position is beyond the current "end of file".

Normally, file managers that do not support random access ignore Seek. However, an SCF-type manager can use Seek to perform cursor positioning.

Read

Read returns the number of bytes requested to the user's data buffer. Make sure Read returns an EOF error if there is no data available. Read must be capable of copying pure binary data, and generally performs no editing on the data. Generally, the file manager calls the device driver to actually read the data into the buffer. Then, the file manager copies the data from the buffer into the user's data area to keep file managers device-independent.

Write

The Write request, like Read, must be capable of recording pure binary data without alteration. The routines for Read and Write are almost identical with the exception that Write uses the device driver's output routine instead of the input routine. The RBF manager and similar random access devices that use fixed-length records (sectors) must often preread a sector before writing it, unless they are writing the entire sector. In OS-9, writing past the end of file on a device expands the file with new data.

ReadLn

ReadLn differs from Read in two respects. First, ReadLn terminates when the first end-of-line (carriage return) is encountered. ReadLn performs any input editing that is appropriate for the device. In the case of SCF, editing involves handling functions such as backspace, line deletion, and the removal of the high-order bit from characters.

WriteLn

WriteLn is the counterpart of ReadLn. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing can also be performed. For example, SCF outputs a line feed, a carriage return character, and nulls (if appropriate for the device). It also pauses at the end of a screen page.

GetStat, PutStat

The GetStat (get status) and PutStat (put status) system calls are wildcard calls designed to provide a method of accessing features of a device (or file manager) that are not generally device independent. The file manager can perform specific functions such as setting the size of a file to a given value. Pass *unknown* status calls to the driver to provide further means of device independence. For example, a PutStat call to format a disk track might behave differently on different types of disk controllers.

Close

Close is responsible for ensuring that any output to a device is completed. (If necessary, Close writes out the last buffer.) It releases any buffer space allocated in an Open or Create. Close does not execute the device driver's terminate routine, but can do specific end-of-file processing if you want it to, such as writing end-of-file records on disks, or form feeds on printers.

Interfacing with Device Drivers

Strictly speaking, device drivers must conform to the general format presented in this manual. The I/O Manager is slightly different because it only uses the Init and Terminate entry points. Other entry points need only be compatible with the file manager for which the driver is written. For example, the Read entry point of an SCF driver is expected to return one byte from the device. The Read entry point of an RBF driver, on the other hand, expects Read to return an entire sector.

The following code is part of an SCF file manager. The code shows how a file manager might call a driver.

```
****************
*  IOEXEC
*    Execute Device's Read/Write Routine
*
*  Passed:    (A) = Output character (write)
*             (X) = Device Table entry ptr
*             (Y) = Path Descriptor pointer
*             (U) = Offset of routine (D$Read,
                    D$Write)
*  Returns:   (A) = Input char (read)
*             (B) = Error code, CC set if error
*  Destroys B,CC

IOEXEC pshs a,x,y,u save registers
 ldu V$STAT,x get static storage for driver
 ldx V$DRIV,x get driver module address
 ldd M$EXEC,x and offset of execution entries
 addd 5,s offset by read/write
 leax d,x absolute entry address
 lda ,s+ restore char (for write)
 jsr 0,x execute driver read/write
 puls x,y,u,pc return (A)=char, (B)=error

 emod Module CRC
Size equ * size of sequential file manager
```

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 re-entrant. So, one copy of the module can concurrently run several devices that use identical I/O controllers.

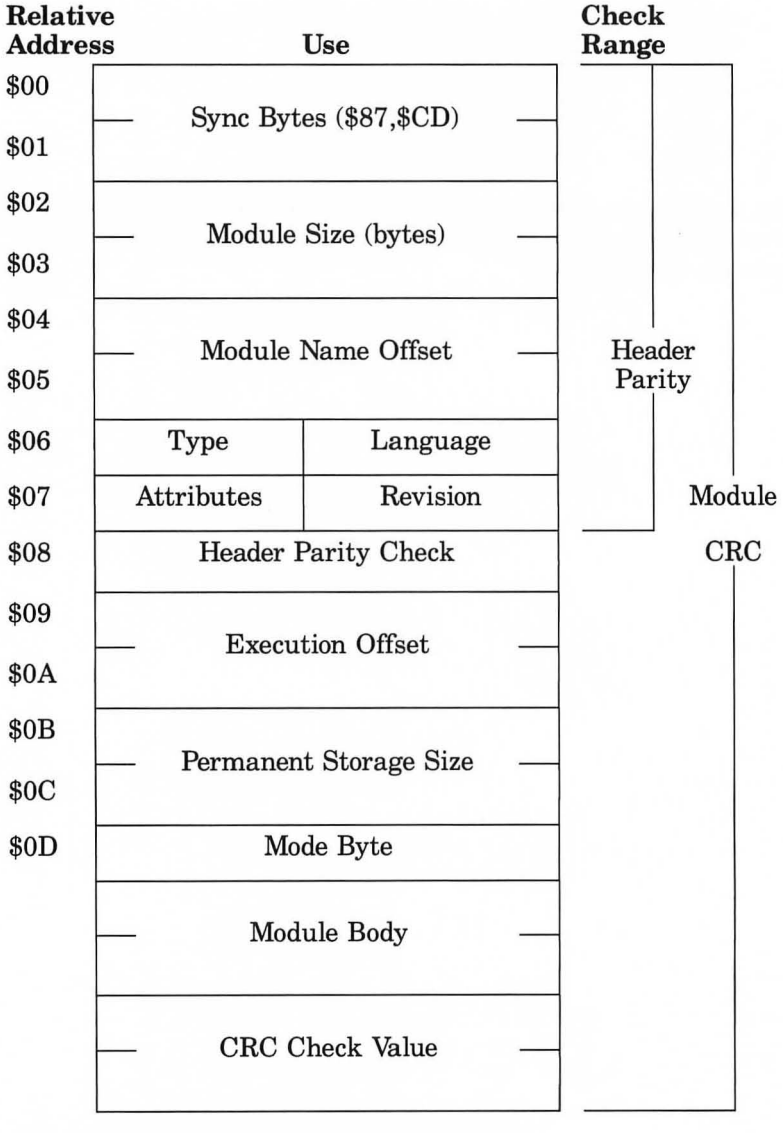
Device driver modules use a standard module header, in which the module type is specified as code $Ex (device driver). The execution offset address in the module header points to a branch table that has a minimum of six 3-byte entries.

Each entry is typically an 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 passing the hardware control register address in the 6809 registers. 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 Driver Module Format



$0D Mode Byte - (D S PE PW PR E W R)

OS-9 Interaction With Devices

Device drivers often must wait for hardware to complete a task or for a user to enter data. Such a wait situation occurs if an SCF device driver receives a Read but there is no data is available, or if it receives a Write and no buffer space is available. OS-9 drivers that encounter this situation should suspend the current process (via F$Sleep). In this way the driver allows other processes to continue using CPU time.

The most efficient way for a driver to awaken itself and resume processing data is by using interrupt requests (IRQs). It is possible for the driver to sleep for a number of system clock ticks and then check the device or buffer for a ready signal. The drawbacks to this technique are:

- It requires the system clock to always remain active.
- It might require a large number of ticks (perhaps 20) for the device to become ready. Such a case leaves you with a dilemma. If you make the program sleep for two ticks, the system wastes CPU time while checking for device ready. If the driver sleeps 20 ticks, it does not have a good response time.

An interrupt system allows the hardware to report to the CPU and the device drivers when the device is finished with an operation. Using interrupts to its advantage, a device driver can set up interrupt handling to occur when a character is sent or received or when a disk operation is complete. There is a built-in polling facility for pausing and awakening processes. Here is a technique for handling interrupts in a device driver:

1. Use the Init routine to place the driver interrupt service call (IRQSVC) routine in the IRQ polling sequence via an F$IRQ system call:

```
ldd V.Port,u get address to poll
leax IRQPOLL,pcr point to IRQ packet
leay IRQSVC,pcr point to IRQ routine
OS9 F$IRQ add dev to poll sequence
bcs Error abnormal exit if error
```

2. Ensure that driver programs waiting for their hardware, call the sleep routine. The sleep routine copies V.Busy to V.Wake. Then, it goes to sleep for some period of time.

3. When the driver program wakes up, have it check to see whether it was awakened by an interrupt or by a signal sent from some other process.

   Usually, the driver performs this check by reading the V.Wake storage byte. The V.Busy byte is maintained by the file manager to be used as the process ID of the process using the driver. When V.Busy is copied into V.Wake, then V.Wake becomes a flag byte and an information byte. A non-zero Wake byte indicates that there is a process awaiting an interrupt. The value in the Wake byte indicates the process to be awakened by sending a wakeup signal as shown in the following code:

```
        lda V.Busy,u        get proc ID
        sta V.Wake,u        arrange for wakeup
        andcc #^IntMasks    prep for interrupts
Sleep50 ldx #0              or any other tick time
                            (if signal test)
        OS9 F$Sleep         await an IRQ
        ldx D.Proc          get proc desc ptr if
                            signal test
        ldb P$Signal,x      is signal present?
                            (if signal test)
        bne SigTest         bra if so if signal
                            test
        tst V.Wake,u        IRQ occur?
        bne Sleep50         bra if not
```

   Note that the code labeled "if signal test" is only necessary if the driver wishes to return to the caller if a signal is sent without waiting for the device to finish. Also note that IRQs and FIRQs must be masked between the time a command is given to the device and the moving of V.Busy and V.Wake. If they are not masked, it is possible for the device IRQ to occur and the IRQSVC routine to become confused as to whether it is sending a wakeup signal or not.

4. When the device issues an interrupt, OS-9 calls the routine at the address given in F$IRQ with the interrupts masked. Make the routine as short as possible, and have it return with an RTS instruction. IRQSVC can verify that an interrupt has occurred for the device. It needs to clear the interrupt to retrieve any data in the device. Then the V.Wake byte communicates with the main driver module. If V.Wake is non-zero, clear it to indicate a true device interrupt and use its contents as the process ID for an F$Send system call. The F$Send call sends a wakeup signal to the process. Here is an example:

```
        ldx V.Port,u get device address
        tst ?? is it real interrupt from device?
        bne IRQSVC90 bra to error if not
        lda Data,x get data from device
        sta 0,y
        lda V.Wake,u
        beq IRQSVC80 bra if none
        clr V.Wake,u clear it as flag to main
        routine
        ldb #S$Wake,u get wakeup signal
        OS9 F$Send send signal to driver
IRQSVC80 clrb clear carry bit (all is well)
        rts
IRQSVC90 comb set carry bit (is an IRQ call)
        rts
```

Suspend State (Level Two only)

The Suspend State allows the elimination of the F$Send system call during interrupt handling. Because the process is already in the active queue, it need not be moved from one queue to another. The device driver IRQSVC routine can now wake up the suspended main driver by clearing the process status byte suspend bit in the process state. Following are sample routines for the Sleep and IRQSVC calls:

```
        lda D.Proc get process ptr
        sta V.Wake,u prep for re-awakening

        enable device to IRQ, give command, etc.

        bra Cmd50 enter suspend loop

Cmd30 ldx D.Proc get ptr to process desc
```

```
        lda P$State,x get state flag
        ora #Suspend put proc in suspend state
        sta P$State,x save it in proc desc
        andcc #^IntMasks unmask interrupts
        ldx #1 give up time slice
        OS9 F$Sleep suspend (in active queue)
Cmd50   orcc #IntMasks mask interrupts while
changing state
  ldx D.Proc get proc desc addr (if signal
test)
  lda P$Signal,x get signal (if signal test)
  beq SigProc bra if signal to be handled
  lda V.Wake,u true interrupt?
  bne Cmd30 bra if not
  andcc #^IntMasks assure interrupts unmasked
```

Note that D.Proc is a pointer to the process descriptor of the current process. Process descriptors are always allocated on 256-byte page boundaries. Thus, having the high order byte of the address is adequate to locate the descriptor. D.Proc is put in V.Wake as a dual value. In one instance, it is a flag byte indicating that a process is indeed suspended. In the other instance, it is a pointer to the process descriptor which enables the IRQSVC routine to clear the suspend bit. It is necessary to have the interrupts masked from the time the device is enabled until the suspend bit has been set. Making the interrupts ensure that the IRQSVC routine does not think it has cleared the suspend bit before it is even set. If this happens, when the bit is set the process might go into permanent suspension. The IRQSVC routine sample follows:

```
ldy V.Port,u get dev addr
tst V.Wake,u is process awaiting
IRQ?
beq IRQSVCER no exit

clear device interrupt
exit if IRQ not from this device

lda V.Wake,u get process ptr
clrb
stb V.Wake,u clear proc waiting flag
tfr d,x get process descriptor ptr
lda P$State,x get state flag
anda # Suspend clear suspend state
sta P$State,x save it
```

```
            clrb clear carry bit
            rts
IRQSVCER comb set carry bit
            rts
```

Device Descriptor Modules

Device descriptor modules are small, non-executable modules. Each one provides 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 $Fx (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	The relative address of the file manager name string address
$0B,$0C	The relative address of the device driver name string
$0D	Mode/Capabilities: D S PE PW PR E W R (directory, single user, public execute, public write, public read, execute, write, read)
$0E,$0F,$10	The absolute physical (24-bit) address of the device controller
$11	The number of bytes (*n* bytes) in the initialization table
$12,$12+*n*	Initialization table

When OS-9 opens a path to the device, the system copies the initialization table into the option section (PD.OPT) of the path descriptor. (See "Path Descriptors" in this chapter.)

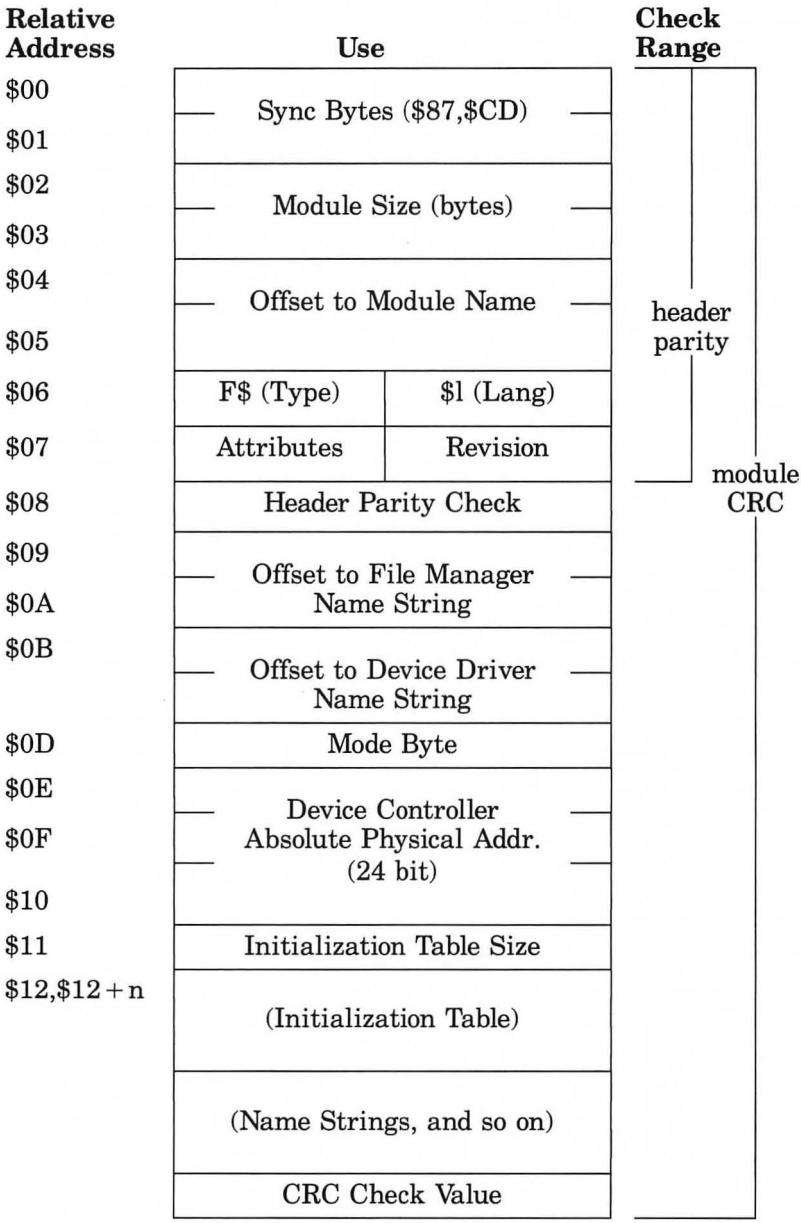
The values in this table can be used to define the operating parameters that are alterable by the Get Status and Set Status system calls (I$GetStt and I$SetStt). For example, parameters that are used when initializing terminals define which control characters are to be used for functions such as backspace and delete.

The initialization table can be a maximum of 32 bytes long. If the table is fewer than 32 bytes long, OS-9 sets the remaining values in the path descriptor to 0.

You might wish to add devices to your system. If a similar device driver already exists, all you need to do is add the new hardware and load another device descriptor. Device descriptors can be in the boot module or they can be loaded into RAM from mass-storage files while the system is running.

The following diagram illustrates the device descriptor format:

Device Descriptor Format



Path Descriptors

Every open path is represented by a data structure called a *path descriptor* (PD). The PD 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 the I/O manager as paths are opened and closed.

They are internal data structures, that 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 section, which is ten bytes long, is the same for all file managers and device drivers. The information in the first section is shown in the following chart.

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 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 22-byte storage reserved for and defined by each type of file manager for file pointers, permanent variables, and so on.

PD.OPT is a 32-byte option area used for file or device operating parameters that are dynamically alterable. When the path is opened, the I/O manager initializes these variables by copying the initialization table that is in the device descriptor module. User programs can change the values 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 the SCF manager or RBFDefs for the RBF manager).

Chapter 5

Random Block File Manager

The random block file manager (RBF manager) supports *disk storage*. It is a re-entrant subroutine package called by the I/O manager for I/O system calls to random-access devices. It maintains the logical and physical file structures.

During normal operation, the RBF manager requests allocation and deallocation of 256-byte data buffers. Usually, one buffer is required for each open file. When physical I/O functions are necessary, the RBF manager directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks (pages).

The RBF manager 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 the RBF manager 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, the parameters are written on the first few sectors of Track 0. The device drivers also use the information, particularly the physical parameters stored on Sector 0. These parameters are written by the FORMAT program that initializes and tests the disk.

Logical and Physical Disk Organization

All disks used by OS-9 store basic identification, file structure, and storage allocation information on these first few sectors.

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.

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 following table 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 in which the last character has the most significant bit set
DD.OPT	$3F		Path descriptor options

Disk Allocation Map Sector (LSN 1)

LSN 1 contains the *disk allocation map*, which is created by FORMAT. This map shows which sectors are allocated to the files and which are free for future use.

Each bit in the allocation map represents a sector or cluster of sectors on the disk. If the bit is set, the sector is considered to be in use, defective, or non-existent. If the bit is cleared, the corresponding cluster is available. The allocation map usually starts at LSN1. The number of sectors it requires varies according to how many bits are needed for the map. DD.MAP specifies the actual number of bytes used in the map.

Multiple sector allocation maps allow the number of sectors/cluster to be as small as possible for high volume media.

The FORMAT utility bases the size of the allocation map on the size and number of sectors per cluster.

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 disk allocation 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).

If a cluster is available, the corresponding bit is cleared. If it is allocated, non-existent, or physically defective, the corresponding bit is set.

ROOT Directory

This file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as /D1). Usually, it immediately follows the Allocation Map. The location of the ROOT directory file descriptor is specified in DD.DIR. The ROOT directory contains an entry for each file that resides in the directory, including other directories.

File Descriptor Sector

The first sector of every file is the *file descriptor*. It contains the logical and physical description of the file.

The following table 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 next chart)
FD.OWN	$01	2	Owner's user ID
FD.DAT	$03	5	Date last modified: (Y M D H M)
FD.LNK	$08	1	Link count
FD.SIZ	$09	4	File size (number of bytes)
FD.CREAT	$0D	3	Date created (Y M D)
FD.SEG	$10	240	Segment list (see next chart)

FD.ATT. (The attribute byte) contains the file permission bits. When set the bits indicate the following:

Bit 7 Directory
Bit 6 Single user
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 a maximum of 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 is 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 also made in minimum allocation increments. Whenever possible, OS-9 expands the last segment, instead of adding a segment. When the file is closed, OS-9 truncates unused sectors in the last segment.

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 in the file.

If a file is repeatedly closed, opened, and expanded, it can become fragmented so that it has many segments. You can avoid this fragmentation by writing a byte at the highest address you want 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 three bytes for the LSN of the file's descriptor sector. The filename is left-justified in the field, with the most significant 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.

The RBF Manager 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 following table describes the use of this section by the RBF manager. 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
The RBF manager Path Descriptor Definitions (PD.FST Section)			
PD.SMF	$0A	1	State flag: Bit 0 = current buffer is altered Bit 1 = current sector is in the buffer Bit 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)

Name	Relative Address	Size (Bytes)	Use
PD.SSZ	$19	3	Segment size
PD.DSK	$1C	2	Disk ID (for internal use only)
PD.DTB	$1E	2	Address of drive table

Name	Relative Address	Size (Bytes)	Use
The RBF manager Option Section Definitions (PD.OPT Section)			
(Copied from the device descriptor)			
PD.DTP	$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
PD.TFM	$2F	1	DMA transfer mode
PD.EXTEN	$30	2	Path extension for record locking
PD.STOFF	$32	1	Sector/track offsets

Name	Relative Address	Size (Bytes)	Use
(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 following values are those the I/O manager 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
IT.STP	$14	1	Step rate
IT.TYP	$15	1	Device type (see RBF path descriptor)
IT.DNS	$16	1	Media density: Always 1 (double) (see following information)
IT.CYL	$17	2	Number of cylinders (tracks)
IT.SID	$19	1	Number of sides
IT.VFY	$1A	1	0 = Verify disk writes 1 = no verify
IT.SCT	$1B	2	Default number of sectors per track
IT.T0S	$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. Number the drives for each controller as 0 to *n-1*, where *n* is the maximum number of drives the controller can handle.

IT.TYP specifies the device type (all types).

Bit 0 — 0 = 5-inch floppy diskette

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 diskette
1 = Hard disk

IT.DNS specifies the density capabilities (floppy diskette 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)

IT.SAS specifies the minimum number of sectors allowed at one time.

RBF Record Locking

Record locking is a general term that refers to methods designed to preserve the integrity of files that can be accessed by more than one user or process. The OS-9 implementation of record locking is designed to be as invisible as possible. This means that existing programs do not have to be rewritten to take advantage of record locking facilities. You can usually write new programs without special concern for multi-user activity.

Record locking involves detecting and preventing conflicts during record access. Whenever a process modifies a record, the system locks out other procedures from accessing the file. It defers access to other procedures until it is safe for them to write to the record. The system does not lock records during reads; so, multiple processes can read the record at the same time.

Record Locking and Unlocking

To detect conflicts, OS-9 must recognize when a record is being updated. The RBF manager provides true record locking on a byte basis. A typical record update sequence is:

```
OS9 I$Read      program reads record
                RECORD IS LOCKED
   .
   .            program updates record
   .
OS9 I$Seek      reposition to record
OS9 I$Write     record is rewritten
                RECORD IS RELEASED
```

When a file is opened in update mode, any read causes locking of the record being accessed. This happens because the RBF manager cannot determine in advance if the record is to be updated. The record stays locked out until the next read, write, or close.

However, when a file is opened in the read or execute modes, the system does not lock accessed records because the records cannot be updated in these two modes.

A subtle but important problem exists for programs that interrogate a data base and occasionally update its data. If you neglect to release a record after accessing it, the record might be locked up indefinitely. This problem is characteristic of record locking systems and you can avoid it with careful programming.

Only one portion of a file can be locked out at a time. If an application requires more than one record to be locked out, open multiple paths to the same file and lock the record accessed by each path. RBF notices that the same process owns both paths and keeps them from locking each other out.

Non-Shareable Files

Sometimes (although rarely), you must create a file that can never be accessed by more than one user at a time. To lock the file, you set the single-user (s) bit in the file's attribute byte. You can do this by using the proper option when the file is created, or later using the OS-9 ATTR command. Once the single-user bit is set, only one user can open the file at a time. If other users attempt to open the file, Error 253 is returned. Note however, that non-shareable means only one path can be opened to a file at one time. Do not allow two processes to concurrently access a non-shareable file through the same path.

More commonly, you need to declare a file as single-user only during the execution of a specific program. You can do this by opening the file with the single-user bit set. For example, suppose a process is sorting a file. With the file's single-user bit set, OS-9 treats the file exactly as though it had a single-user attribute. If another process attempts to open the file, OS-9 returns Error 253.

You can duplicate non-shareable paths by using the I$Dup system call. This means that it can be inherited, and therefore accessible to more than one process at a time. Single-user means that the file can be opened only once.

End-of-File Lock

A special case of record locking occurs when a user reads or writes data at the end of a file, creating an *EOF Lock*. An EOF Lock keeps the end of the file locked out until a process performs a READ or WRITE that is not at the end of the file. It prevents problems that might otherwise occur when two users want to simultaneously extend a file. The EOF Lock is the only case in which a WRITE call automatically causes portions of a file to be locked out. An interesting and useful side effect of the EOF Lock function occurs if a program creates a file for sequential output. As soon as the program creates the file, EOF Lock is set and no other process can *pass* the writer in processing the file. For example, if an assembler redirects a listing to a disk file, and a spooler utility tries to print a line from the file before it is written, record locking makes the spooler wait and stay at least one step behind the assembler.

Deadlock Detection

A deadly embrace, or deadlock, typically occurs when two processes attempt to gain control of two or more disk areas at the same time. If each process gets one area (locking out the other process), both processes become permanently stuck. Each waits for a segment that can never become free. This situation is not restricted to any particular record locking scheme or operating system.

When a deadly embrace occurs, RBF returns a deadlock error (Error 254) to the process that caused OS-9 to detect the deadlock. To avoid deadlocks, make sure that processes always access records of shared files in the same sequence.

When a deadlock error occurs, it is not sufficient for a program to retry the operation that caused the error. If all processes use this strategy, none can ever succeed. For any process to proceed, at least one must cancel operation to release its control over a requesting segment.

RBF-Type Device Driver Modules

An RBF-type device driver module contains a package of subroutines that perform sector-oriented I/O to or from a specific hardware controller. Such a module is usually re-entrant. Because of this, one copy of one device driver module can simultaneously run several devices that use identical I/O controllers.

The I/O manager allocates a permanent memory area for each device driver. The size of the memory area is given in the device driver module header. The I/O manager and the RBF manager use some of this area. The device driver can use the rest in any manner. This area is used as follows:

The RBF Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use
V.PAGE	$00	1	Port extended address bits A20-A16
V.PORT	$01	2	Device base address (defined by the I/O manager)

Name	Relative Address	Size (Bytes)	Use
V.LPRC	$03	1	ID of the last active process (not used by RBF device drivers)
V.BUSY	$04	1	ID of the current process using driver (defined by RBF) 0 = no current process
V.WAKE	$05	1	ID of the process waiting for I/O completion (defined by the device driver)
V.USER	$06	0	Beginning of file manager specific storage
V.NDRV	$06	1	Maximum number of drives the controller can use (defined by the device driver)
	$07	8	Reserved
DRVBEG	$0F	0	Beginning of the drive tables
TABLES	$0F	DRVMEN*N	Space for number of tables reserved (*n*)
FREE		0	Beginning of space available for driver

These values are defined in files in the DEFS directory on the Development Package disk.

TABLES. This area contains one table for each drive that the controller handles. (The RBF manager assumes that there are as many tables as indicated by V.NDRV.) Some time after the driver Init routine is called, the RBF manager issues a request for the driver to read LSN 0 from a drive table by copying the first part of LSN 0 (up to DD.SIZ) into the table. Following is the format of each drive table:

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 diskette swaps).
DD.FMT	$10	1	Media format.
DD.SPT	$11	2	Number of sectors per track. (Track 0 can use a different value specified by IT.TOS in the device descriptor.)
DD.RES	$13	2	Reserved for future use.
DD.SIZ	$15	0	Minimum size of device descriptor.
V.TRAK	$15	2	Number of the current track (the track that the head is on, and the track updated by the driver).
V.BMB	$17	1	Bit-map use flag: 0 = Bit map is not in use. (Disk driver routines must not alter V.BMB.)
V.FILEHD	$18	2	Open file list for this drive.

Name	Relative Address	Size (Bytes)	Use
V.DISKID	$1A	2	Disk ID.
V.BMAPSZ	$1C	1	Size of bitmap.
V.MAPSCT	$1D	1	Lowest reasonable bitmap sector.
V.RESBIT	$1E	1	Reserved bitmap sector.
V.SCTKOF	$1F	1	Sector/track byte.
V.SCOFST	$20	1	Sector offset split from V.SCTKOF.
V.TKOFST	$21	1	Track offset split from V.SCTKOF.
RESERVED	$22	4	Reserved for future use.
DRVMEN	$26		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)
1 = Double (96 tracks per inch)

RBF Device Driver Subroutines

Like all device driver modules, RBF device drivers use a standard executable memory module format.

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

Ensure that each subroutine exists with the C bit of the condition code register cleared if no error occurred. If an error occurs, set the C bit and return an appropriate error code Register B.

The rest of this chapter describes the RBF device driver subroutines and their entry and exit conditions.

Init Initializes a device and the device's memory area.

Entry Conditions:

Y = *address of the device descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If you want OS-9 to verify disk writes, use the Request Memory system call (F$SRqMem) to allocate a 256-byte buffer area in which a sector can be read back and verified after a write.
- You must initialize the device memory area. For floppy diskette controllers, initialization typically consists of:
  1. Initializing V.NDRV to the number of drives with which the controller works
  2. Initializing DD.TOT (in the drive table) to a non-zero value so that Sector 0 can be read or written
  3. Initializing V.TRAK to $FF so that the first seek finds Track 0
  4. Placing the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F$IRQ)
  5. Initializing the device control registers (enabling interrupts if necessary)
- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These areas contain the 24- bit device address.) Ensure the driver initializes each drive table appropriately for the type of diskette that the driver expects to be used on the corresponding drive.

Read

Reads a 256-byte sector from a disk and places it in a 256-byte sector buffer.

Entry Conditions:

B = *MSB of the disk's LSN*
X = *LSB of the disk's LSN*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The following is a typical routine for using Read:

  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. The driver goes to sleep and waits 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 you read LSN 0, you must copy the first part of this sector 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. Use the drive number (PD.DRV) to compute the offset for 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 the table
```

Write Writes a 256-byte sector buffer to a disk.

Entry Conditions:

B = *MSB of the disk LSN*
X = *LSB of the disk LSN*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code*

Additional Information:

- Following is a typical routine for using Write:

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. The driver then goes to sleep and waits for the I/O to complete. (The IRQ service routine sends 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 is written correctly. Verification usually does not involve a comparison of all of the data bytes.

- If disk writes are to be verified, the Init routine must request the buffer in which to place the sector when it is read back. Do not copy LSN 0 into the drive table when reading it back for verification.

- Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the Read routine.

Getstats and Setstats

Reads or changes device's operating parameters.

Entry Conditions:

U = *address of the device memory area*
Y = *address of the path descriptor*
A = *status code*

Exit Conditions:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Get/set the device's operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wild card calls.
- It might 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. You can use the following offsets to access any value in the register stack:

Reg.	Relative Addr.	Size	6809 Reg.
R$CC	$00	1	Condition Code Reg.
R$D	$01	2	Register D
R$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

- Register D overlays Registers A and B.

Term Terminate a device.

Entry Conditions:

U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- 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).
- Following is a typical routine for using Term:

  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

Services device interrupts.

Additional Information:

- The IRQ Service routine sends a wakeup signal to the process indicated by the process ID in V.WAKE when the I/O is complete. It then clears V.WAKE as a flag to indicate to the main program that the IRQ has indeed occurred.
- When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.
- Although this routine is not included in the device driver module branch table and is not called directly by the RBF manager, it is a key routine in interrupt-driven drivers. Its function is to:

  1. Service the device interrupts (receive data from device or send data to it). The IRQ service routine puts its data into and get its data from buffers that are defined in the device memory area.
  2. Wake up a process that is waiting for I/O to be completed. To do this, the routine checks to see if there is a process ID in V.WAKE (if the bit is 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.

Boot (Bootstrap Module)

Loads the boot file into RAM.

Entry Conditions:

None

Exit Conditions:

D = *size of the boot file* (in bytes)
X = *address at which the boot file was loaded into memory*
CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Boot module is not part of the disk driver. It is a separate module that is stored on the boot track of the system disk with OS9P1 and REL.
- The bootstrap module contains one subroutine that loads the bootstrap file and related information into memory. It uses the standard executable module format with a module type of $C. The execution offset in the module header contains the offset to the entry point of this subroutine.
- The module gets the starting sector number and size of the OS9Boot file from LSN 0. OS-9 allocates a memory area large enough for the Boot file. Then, it loads the Boot file into this memory area.
- Following is a typical routine for using Boot:

  1. Read LSN 0 from the disk into a buffer area. The Boot module 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 module 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 in which the Boot file is loaded.
5. Read the Boot file into this memory area.
6. Return the size of the Boot file and its location. Boot file is loaded.

Chapter 6

Sequential Character File Manager

The Sequential Character File Manager (SCF) supports devices that operate on a character-by-character basis. These include terminals, printers, and modems.

SCF is a re-entrant subroutine package. The I/O manager calls the SCF manager for I/O system handling of sequential, character-oriented devices. The SCF manager includes the extensive I/O editing functions typical of line-oriented operation, such as:

- backspace
- line delete
- line repeat
- auto line feed
- screen pause
- return delay padding

The SCF-type device driver modules are CC3IO, PRINTER, and RS-232. They run the video display, printer, and serial ports respectively. See the *OS-9 Commands* manual for additional Color Computer I/O devices.

SCF Line Editing Functions

The SCF manager supports two sets of read and write functions. I$Read and I$Write pass data with no modification. I$ReadLn and I$WritLn provide full line editing of device functions.

Read and Write

The Read and Write system calls to SCF-type devices correspond to the BASIC09 GET and PUT statements. While they perform little modification to the data they pass, they do filter out keyboard interrupt, keyboard terminate, and pause character. (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 to SCF-type devices correspond to the BASIC09 INPUT, PRINT, READ, and WRITE statements. They provide full line editing of all functions enabled for a particular device.

The system initializes I$ReadLn and I$WritLn functions when you first use a particular device. (OS-9 copies the option table from the device descriptor table associated with the specific device.)

Later, you can alter the calls—either from assembly-language programs (using the Get Status system call), or from the keyboard (using the TMODE command). All bytes transferred by Readln and Writln have the high order bit cleared.

SCF Definitions of the Path Descriptor

The PD.FST and PD.OPT sections of the path descriptor are reserved for and used by the SCF file manager.

The following table describes the SCF manager's use of PD.FST and PD.OPT. For your convenience, the table 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 that can be read or written by the Set Status or Get Status system call. Any values not set by this table default to zero.

> **Note:** You can 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 can go a step further by setting the corresponding control character value in the device descriptor module to zero.

To determine the default settings for a specific device, you can inspect the device descriptor.

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)

Name	Relative Address	Size (Bytes)	Use
SCF 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
SCF Option Section Definition (PD.OPT Section)			
(Copied from the device descriptor)			
PD.DTP	$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, 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: 0 = no auto line feed
PD.NUL	$26	1	End-of-line null count: *n* = number of nulls ($00) sent after each carriage return or carriage return and line feed (*n* = $00-$FF)
PD.PAU	$27	1	End of page pause: 0 = no pause
PD.PAG	$28	1	Number of lines per page
PD.BSP	$29	1	Backspace character
PD.DEL	$2A	1	Delete-line character

Name	Relative Address	Size (Bytes)	Use
SCF Option Section Definition continued (PD.OPT Section)			
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
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
PD.QUT	$31	1	Keyboard-terminate character
PD.BSE	$32	1	Backspace-echo character
PD.OVF	$33	1	Line-overflow character (bell CTRL G)
PD.PAR	$34	1	Device-initialization value (parity)
PD.BAU	$35	1	Software setable baud rate
PD.D2P	$36	2	Offset to second device name string
PP.XON	$38	1	ACIA XON char
PD.XOFF	$39	1	ACIA XOFF char
PD.ERR	$3A	1	Most recent I/O error status
PD.TBL	$3B	2	Copy of device table address
PD.PLP	$3D	2	Path descriptor list pointer
PD.PST	$3F	1	Current path status

PD.EOF specifies the end-of-file character. If this is the first and only character that is input to the SCF device, SCF returns an end-of-file error on Read or Readln.

PD.PSC specifies the pause character, which suspends output to the device 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, the system sends a keyboard terminate signal to the last user of a 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-terminate character. When this character is received, the system sends a keyboard terminate signal to the last user of a path. The system also cancels the current I/O request (if any) by sending error code identical to the keyboard interrupt signal code.

PD.PAR specifies the parity information for external serial devices.

PD.BAU specifies baud rate, word length and stop bit information for serial devices.

PD.XON contains either the character used to enable transmission of characters or a null character that disables the use of XON.

PD.XOFF contains either the character used to disable transmission of characters or a null character that disables the use of XOFF.

SCF-Type Device Descriptor Modules

The following chart shows how the initialization table in the device descriptors is used for SCF-type devices. The values are those the I/O manager 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
(header)	$00-11		Standard device descriptor module header
IT.DVC	$12	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
IT.UPC	$13	1	Case: 0 = upper- and lowercase 1 = uppercase only
IT.BSO	$14	1	Backspace: 0 = backspace 1 = backspace, then space and backspace
IT.DLO	$15	1	Delete: 0 = backspace over line 1 = carriage return
IT.EKO	$16	1	Echo: 0 = echo off
IT.ALF	$17	1	Auto line feed: 0 = auto line feed disabled
IT.NUL	$18	1	End-of-line null count
IT.PAU	$19	1	Pause: 0 = end-of-page pause disabled
IT.PAG	$1A	1	Number of 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

Name	Relative Address	Size (Bytes)	Use
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—used to initialize a device control register when a path is opened to it (parity)
IT.BAU	$27	1	Baud rate
IT.D2P	$28	2	Attached device name string offset
IT.XON	$2A	1	X-ON character
IT.XOFF	$2B	1	X-OFF character
IT.COL	$2C	1	Number of columns for display
IT.ROW	$2D	1	Number of rows for display
IT.WND	$2E	1	Window number
IT.VAL	$2F	1	Data in rest of descriptor is valid
IT.STY	$30	1	Window type
IT.CPX	$31	1	X cursor position
IT.CPY	$32	1	Y cursor position
IT.FGC	$33	1	Foreground color
IT.BGC	$34	1	Background color
IT.BDC	$35	1	Border color

SCF-Type Device Driver Modules

An SCF-type device driver module contains a package of subroutines that perform raw (unformatted) data I/O transfers to or from a specific hardware controller. Such a module is usually reentrant so that one copy of the module can simultaneously run several devices that use identical I/O controllers. The I/O manager allocates a permanent memory area for each controller sharing the driver.

The size of the memory area is defined in the device driver module header. The I/O manager and SCF use some of the memory area. The device driver can use the rest in any way (typically as variables and buffers). Typically, the driver uses the area as follows:

Name	Relative Address	Size (Bytes)	Use
V.PAGE	$00	1	Port extended 24 bit address
V.PORT	$01	2	Device base address (defined by the I/O manager)
V.LPRC	$03	1	ID of the last active process
V.BUSY	$04	1	ID of the active process (defined by RBF): 0 = no active process
V.WAKE	$05	1	ID of the process to reawaken after the device completes I/O (defined by the device driver): 0 = no waiting process
V.USER	$06	0	Beginning of file manager specific storage
V.TYPE	$06	1	Device type or parity
V.LINE	$07	1	Lines left until the end of the page
V.PAUS	$08	1	Pause request: 0 = no pause requested
V.DEV2	$09	2	Attached device memory area
V.INTR	$0B	1	Interrupt character

Name	Relative Address	Size (Bytes)	Use
V.QUIT	$0C	1	Quit character
V.PCHR	$0D	1	Pause character
V.ERR	$0E	1	Error accumulator
V.XON	$0F	1	XON character
V.XOFF	$10	1	XOFF character
V.KANJI	$11	1	Reserved
V.KBUF	$12	2	Reserved
V.MODADR	$14	2	Reserved
V.PDLHD	$16	2	Path descriptor list header
V.RSV	$18	5	Reserved
V.SCF	$1D	0	End of SCF memory requirements
FREE	$1D	0	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 sends this process the proper signal if it receives a quit character or an interrupt character. V.LPRC is defined by SCF.

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.) The process ID is used by SCF to prevent more than one process from using the device at the same time. V.BUSY is defined by SCF.

SCF Device Driver Subroutines

Like all device drivers, SCF device drivers use a standard executable memory module format.

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically an LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY     LBRA      INIT      Initialize driver
          LBRA      READ      Read character
          LBRA      WRITE     Write character
          LBRA      GETSTA    Get status
          LBRA      SETSTA    Set status
          LBRA      TERM      Terminate device
```

If no error occurs, each subroutine exits with the C bit in the Condition Code Register cleared. If an error occurred, each subroutine sets the C bit and returns an appropriate error code in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init Initializes device control registers, and enables interrupts if necessary.

Entry Conditions:

Y = *address of the device descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (V.PAGE and V.PORT contain the device address.) There is no need to initialize the part of the memory area used by the I/O manager and SCF.
- Follow these steps to use Init:

  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.

Read **Reads the next character from the input buffer.**

Entry Conditions:

Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

A = *character read*
CC = carry set on error
B = *error code* (if any)

Additional Information:

- This is a step by step description of a Read operation:

  1. Read gets 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.
  3. Later, when Read receives data, the IRQ service routine leaves the data in a buffer. Then, the routine checks V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine sends a wakeup signal to the waiting process.

- Data buffers are not automatically allocated. If a buffer is used, it defines it in the device memory area.

Write Sends a character (places a data byte in an output buffer) and enables the device output interrupts.

Entry Conditions:

A = *character to write*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the data buffer is full, Write copies its process ID from V.BUSY into V.WAKE. Write then puts itself to sleep.

  Later, when the IRQ service routine transmits a character and makes room for more data, it checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, the routine sends a wakeup signal to that process.

- Write must ensure that the IRQ service routine that starts it begins to place data in 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 allocated automatically. If a buffer is used, define it in the device memory area.

Getsta and Setsta

Gets/sets device operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

A = depends on the function code
Y = *address of the path descriptor*
U = *address of the device memory area*
Other registers depend on the function code.

Exit Conditions:

B = *error code* (if any)
CC = carry set on error
Other registers depend on the function code

Additional Information:

- Any codes not defined by the I/O manager or SCF are passed to the device driver.
- You might need 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 can be found in PD.RGS, which is located in the path descriptor.
- You can use the following offsets to access any value in the register packet:

Name	Relative Address	Size (Bytes)	6809 Register
R$CC	$00	1	Condition Codes Register
R$D	$01	0	Register D
R$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

The function code is retrieved from the R$B on the user stack.

Term

Terminates a device. Term is called when a device is no longer in use (when the link count of the device descriptor module becomes zero).

Entry Conditions:

U = *pointer to the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- To use Term:
  1. Wait until the IRQ service routine empties the output buffer.
  2. Disable the device interrupts.
  3. Remove the device from the IRQ polling list.
- When Term closes the last path to a device, OS-9 returns to the memory pool the memory that the device used. If the device has been attached to the system using the I$Attach system call, OS-9 does not return the static storage for the driver until an I$Detach call is made to the device. Modules contained in the Boot file are never terminated, even if their link counts reach 0.

IRQ Service Routine

Receives device interrupts. When I/O is complete, the routine sends a wakeup signal to the process identified by the process ID in V.WAKE. The routine also clears V.WAKE as a flag to indicate to the main program that the IRQ has occurred.

Additional Information:

- The IRQ Service Routine is not included in device driver branch tables, and is not called directly by SCF. However, it is a key routine in device drivers.
- When the IRQ Service routine finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.
- Here is a typical sequence of events that the IRQ Service Routine performs:

  1. Service the device interrupts (receive data from the device or send data to it). Ensure this routine puts its data into and get its data from buffers that are defined in the device memory area.
  2. Wake up any process that is waiting for I/O to complete. To do this, the routine checks to see if there is a process ID in V.WAKE (a value other than 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 value other than zero. The address of the attached device memory area is in V.DEV2.

     V.PAUS in the attached device storage area to non-zero value. The address of the attached device memory area is in V.DEV2.
  5. If a keyboard terminate or interrupt character is received, signal the process in V.LPRC (last known process) if any.

Chapter 7

The Pipe File Manager (PIPEMAN)

The Pipe file manager handles control of processes that use paths to pipes. Pipes allow concurrently executing processes to send each other data by using the output of one process (the writer) as input to a second process (the reader). The reader gets input from the standard input. The exclamation point (!) operator specifies that the input or output is from or to a pipe. The Pipe file manager allocates a 256-byte block and a path descriptor for data transfer. The Pipe file manager also determines which process has control of the pipe. The Pipe file manager has the standard file manager branch table at its entry point:

```
PipeEnt     lbra Create
            lbra Open
            lbra MakDir
            lbra ChgDir
            lbra Delete
            lbra Seek
            lbra PRead
            lbra PWrite
            lbra PRdLn
            lbra PWrLn
            lbra Getstat
            lbra Putstat
            lbra Close
```

You cannot use MakDir, ChgDir, Delete, and Seek with pipes. If you try to do so, the system returns E$UNKSVC (unknown service request). Getstat and Putstat are also no-action service routines. They return without error.

Create and Open perform the same functions. They set up the 256-byte data exchange buffer, and save several addresses in the path descriptor.

The Close request checks to see if any process is reading or writing through the pipe. If not, OS-9 returns the buffer.

PRead, PWrite, PRdLn, and PWrLn read data from the buffer and write data to it.

The ! operator tells the Shell that processes wish to communicate through a pipe. For example:

```
proc1 ! proc2 [ENTER]
```

In this example, shell forks Proc1 with the standard output path to a pipe, and forks Proc2 with the Standard input path from a pipe.

Shell can also handle a series of processes using pipes. Example:

```
proc1 ! proc2 ! proc3 ! proc4 [ENTER]
```

The following outline shows how to set up pipes between processes:

```
Open /pipe        save path in variable x
Dup path #1       save stdout in variable y
Dup x             make path available
Fork proc1        put pipe in stdout
                  (Dup uses lowest available)
Close #1          make path available
Dup y             restore stdout
Close y           make path available
Dup path #0       save stdin in Y
Close #0          make path available
Dup x             put pipe in stdin
Fork 2            fork process 2
Close #0          make path available
Dup y             restore stdin
Close x           no longer needed
Close y           no longer needed
```

Example: The following example shows how an application can initiate another process with the stdin and stdout routed through a pipe.

```
Open /pipe1      save path in variable a
Open /pipe2      save path in variable b
Dup 0            save stdin in variable x
Dup 1            save stdout in variable y
Close 0          make path available
Close 1          make path available
Dup a            make pipe1 stdin
Dup b            make pipe2 stdout
Fork new process
Close 0          make path available
Close 1          make path available
Dup x            restore stdin
Dup y            restore stdout
Return a&b       return pipe path numbers to caller
```

Chapter 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-input conditions equate file called OS9Defs.

System mode calls are privileged. You can execute them 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 writing device drivers and other system-level applications.

Calling Procedure

To execute any system calls, you need to use an SWI2 instruction:

1. Load the 6809 registers with any appropriate parameters.
2. Execute an SWI2 instruction, followed immediately by a constant byte, which is the request code. In the references in this chapter, the first line is the system call name (for example Close Path) and the second line contains the call's mnemonic name (for example I$Close), the software interrupt Code 2 (103F), and the call's request code (for example, 8F) in hexadecimal.
3. After OS-9 processes the call, it returns any parameters in the 6809 registers. If an error occurs, the C bit of the condition code register is set, and Register B contains the appropriate error code. This feature permits a BCS or BCC instruction immediately following the system call to branch either if there is an error or if no error occurs.

As an example, here is the Close system call:

```
LDA PATHNUM
SWI2
FCB  $8F
BCS  ERROR
```

You can use the assembler's *OS9 directive* to simplify the call, as follows.

```
LDA  PATHNUM
OS9  I$Close
BCS  ERROR
```

The ASM assembler allows any combination of upper- or lower-case letters. The RMA assembler, included in the OS-9 Level Two Development Pak, is case sensitive. The names in this manual have been spelled with upper and lower case letters, matching the defs for RMA.

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 whenever a process opens a path to a file or device. Until the path is closed, you can use this path number in later I/O requests to identify the file or device.

In addition, OS-9 allocates and maintains its own data structures; so, 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 name, the SWI2 code and the request code. Next are the call's entry and exit conditions, followed by additional information about the code where appropriate.

In the system call descriptions, registers not specified as entry or exit conditions are not altered. Strings passed as parameters are normally terminated with a space character and end-of-line character, or with Bit 7 of the last character set.

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.

User Mode System Calls Quick Reference

Following is a summary of the User Mode System Calls referenced in this chapter:

F$AllBit	Sets bits in an allocation bit map
F$Chain	Chains a process to a new module
F$CmpNam	Compares two names
F$CpyMem	Copies external memory
F$CRC	Generates a cyclic redundancy check
F$DelBit	Deallocates bits in an allocation bit map
F$Exit	Terminates a process
F$Fork	Starts a new process
F$GBlkMp	Gets a copy of a system block map
F$GModDr	Gets a copy of a module directory
F$GPrDsc	Gets a copy of a process descriptor
F$Icpt	Sets a signal intercept trap
F$ID	Returns a process ID
F$Link	Links to a memory module
F$Load	Loads a module from mass storage
F$Mem	Changes a process's data area size
F$NMLink	Links to a module; does not map the module into the user's address space
F$NMLoad	Loads a module but does not map it into the user's address space
F$Perr	Prints an error message
F$PrsNam	Parses a pathlist name
F$SchBit	Searches a bit map

F$Send	Sends a signal to a process
F$Sleep	Suspends a process
F$SPrior	Sets a process's priority
F$SSWI	Sets a software interrupt vector
F$STime	Sets the system time
F$SUser	Sets the user ID number
F$Time	Returns the current time
F$UnLink	Unlinks a module
F$UnLoad	Unlinks a module by name
F$Wait	Waits for a signal
I$Attach	Attaches an I/O device
I$Chgdir	Changes a working directory
I$Close	Closes a path
I$Create	Creates a new file
I$Delete	Deletes a file
I$DeletX	Deletes a file from the execution directory
I$Detach	Detaches an I/O device
I$Dup	Duplicates a path
I$GetStt	Gets a device's status
I$MakDir	Creates a directory file
I$Open	Opens a path to an existing file
I$Read	Reads data from a device
I$ReadLn	Reads a line of data from a device
I$Seek	Positions a file pointer
I$SetStt	Sets a device's status
I$Write	Writes data to a device
I$WritLn	Writes a data line to a device

System Mode Calls Quick Reference

Following is a summary of the System Mode Calls referenced in this chapter:

F$Alarm	Sets up an alarm
F$All64	Allocates a 64-byte memory block
F$AllHRAM	Allocates high RAM
F$AllImg	Allocates image RAM blocks
F$AllPrc	Allocates a process descriptor
F$AllRAM	Allocates RAM blocks
F$AllTsk	Allocates a process task number
F$AProc	Enters active process queue
F$Boot	Performs a system bootstrap
F$BtMem	Performs a memory request bootstrap
F$ClrBlk	Clears the specified block of memory
F$DATLog	Converts a DAT block offset to a logical address
F$DelImg	Deallocates image RAM blocks
F$DelPrc	Deallocates a process descriptor
F$DelRAM	Deallocates RAM blocks
F$DelTsk	Deallocates a process task number
F$ELink	Links modules using a module directory entry
F$FModul	Finds a module directory entry
F$Find64	Finds a 64-byte memory block
F$FreeHB	Gets a free high block
F$FreeLB	Gets a free low block
F$GCMDir	Compacts module directory entries
F$GProcP	Gets a process's pointer

F$IODel	Deletes an I/O module
F$IOQu	Puts an entry into an I/O queue
F$IRQ	Makes an entry into IRQ polling table
F$LDABX	Loads Register A from 0,X in Task B
F$LDAXY	Loads A[X,[Y]]
F$LDDDXY	Loads D[D+X,[Y]]
F$MapBlk	Maps the specified block
F$Move	Moves data to a different address space
F$NProc	Starts the next process
F$RelTsk	Releases a task number
F$ResTsk	Reserves a task number
F$Ret64	Returns a 64-byte memory block
F$SetImg	Sets a process DAT image
F$SetTsk	Sets a process's task DAT registers
F$SLink	Performs a system link
F$SRqMem	Performs a system memory request
F$SRtMem	Performs a system memory return
F$SSvc	Installs a function request
F$STABX	Stores Register A at 0,x in Task B
F$VIRQ	Makes an entry in a virtual IRQ polling table
F$VModul	Validates a module

User System Calls

Allocate Bits

Sets bits in an allocation bit map

OS9 F$AllBit 103F 13

Entry Conditions:

D = *number of the first bit to set*
X = *starting address of the allocation bit map*
Y = *number of bits to set*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Bit numbers range from 0 to *n-1*, where *n* is the number of bits in the allocation bit map.
- **Warning:** Do not issue the Allocate Bits call with Register Y set to 0 (a bit count of 0).

Chain

OS9 F$Chain 103F 05

Loads and executes a new primary module without creating a new process

Entry Conditions:

A = *language/type code*
B = *size of the data area* (in pages); must be at least one page
X = *address of the module name or filename*
Y = *parameter area size* (in bytes); defaults to zero if not specified
U = *starting address of the parameter area*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Chain 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 it has less processing overhead. Chain resets the calling process program and data memory areas and begins executing a new primary module. It does not affect open paths. This is a user mode system call.
- **Warning:** Make sure that the hardware stack pointer (Register SP) is located in the direct page before Chain executes. Otherwise the system might crash or return a suicide attempt error. This precaution also prevents a suicide in the event that the new module requires a smaller data area than that in use. Allow approximately 200 bytes of stack space for execution of the Chain system call.
- Chain performs the following steps:

  1. It causes OS-9 to unlink the process's old primary module.

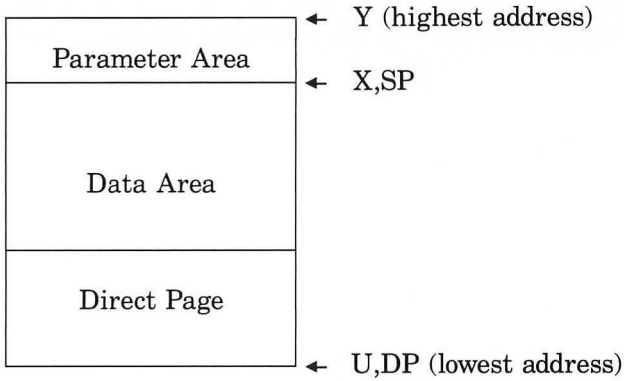
2. OS-9 parses the name string of the new process's primary module (the program that is to be executed first). Then, it causes OS-9 to search the system module directory to see if a module with the same name, type, and language is already in memory.

3. If the module is in memory, it links to it. If the module is not in memory, it uses the name string as the pathlist of a file to load into memory. Then, it links to the first module in this file. (Several modules can be loaded from a single file.)

4. It reconfigures the data memory area to the size specified in the new primary module's header.

5. It intercepts and erases any pending signals.

   The following diagram shows how Chain sets up the data memory area and registers for the new module.



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.)

Compare Names

OS9 F$CmpNam 103F 11

Compares two strings for a match

Entry Conditions:

B = *length of string1*
X = *address of string1*
Y = *address of string2*

Exit Conditions:

CC = carry clear if the strings match

Additional Information:

- The Compare Names call compares two strings and indicates whether they match. Use this call with the Parse Name system call. The second string must have the most significant bit (Bit 7) of the last character set.

Copy External Memory

Reads external memory into the user's buffer for inspection

OS9 F$CpyMem
103F 1B

Entry Conditions:

D = *DAT image pointer*
X = *offset in block to begin copy*
Y = *byte count*
U = *caller's destination buffer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- You can view any system memory through the use of the Copy External Memory call. The call assumes X is the address of the 64K space described by the DAT image given.
- If you pass the entire DAT image of a process, place a value in the X Register that equals the address in the process space. If you pass a partial DAT image (the upper half), then place a value in Register X that equals the offset from the beginning of the DAT image ($8000).
- The support module for this call is OS9p2.

CRC

OS9 F$CRC 103F 17

Calculates the CRC of a module

Entry Conditions:

X = *starting byte address*
Y = *number of bytes*
U = *address of the 3-byte CRC accumulator*

Exit Conditions:

Updates the CRC accumulator.

Additional Information:

- The CRC call 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 can be accumulated over several calls. The CRC accumulator can be any 3-byte memory area. You must initialize it to $FFFFFF before the first CRC call.
- When checking an existing module CRC, the calculation should be performed on the entire module (including the module CRC). The CRC accumulator will contain the CRC constant bytes if the module CRC is correct.
- If the CRC of a new module is to be generated, the CRC is accumulated over the module (excluding CRC). The accumulated CRC is complemented then stored in the correct position in the module.
- Be sure to initialize the CRC accumulator only once for each module checked by CRC.

Deallocate Bits

Clears allocation map bits

OS9 F$DelBit 103F 14

Entry Conditions:

D = *number of the first bit to set*
X = *starting address of the allocation bit map*
Y = *number of bits to set*

Exit Conditions: None

Additional Information:

- The Deallocate Bits call clears bits in the allocation bit map pointed to by Register X. Bit numbers are in the range 0 to *n-1*, where *n* is the number of bits in the allocation bit map.
- **Warning:** Do not call Deallocate Bits with Register Y set to 0 (a bit count of 0).

Exit

Terminates the calling process

OS9 F$Exit 103F 06

Entry Conditions:

B = *status code to return to the parent*

Exit Conditions:

The process is terminated.

Additional Information:

- The Exit system call is the only way a process can terminate itself. Exit deallocates the process's data memory area, and unlinks the process's 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. This is a user mode system call.
- Exit unlinks only the primary module. Unlink any module that is loaded or linked to by the process before calling Exit.

Fork

Creates a child process

OS9 F$Fork 103F 03

Entry Conditions:

A = *language/type code*
B = *size of the optional data area* (in pages)
X = *address of the module name or filename* (See the following example.)
Y = *size of the parameter area* (in pages); defaults to zero if not specified
U = *starting address of the parameter area*; must be at least one page

Exit Conditions:

X = *address of the last byte of the name* + 1 (See the following example.)
A = new process IO number

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Fork creates a new process, a child of the calling process. Fork also sets up the child process's memory and 6809 registers and standard I/O paths.
- Before the Fork call:

T	E	S	T	$0D

X

- After the Fork call:

T	E	S	T	$0D

X

- This is the sequence of Fork's operations:

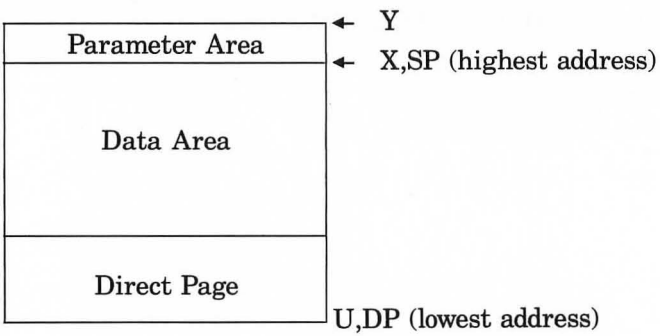
1. OS-9 parses the name string of the new process's primary module (the program that OS-9 executes first). Then, it searches the system module directory to see if the program already is in memory.

2a. The next step depends on whether or not the program is already in memory. If the program is in memory, OS-9 links the module to the process and executes it.

b. 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 can be 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 data memory area and registers are set up as shown in the following diagram. OS-9 uses the execution offset given in the module header to set the program counter to the module's entry point.



D = *size of the parameter area*
PC = *module entry point absolute address*
CC = F=0, I=0, other condition code flags 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, 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.

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, the shell also inserts an end-of-line character at the end of the parameter string.

Register X points to the start byte of the parameter string. If the command line includes the optional memory size specification (*#n* or *#n*K), the shell passes that size as the requested memory size when executing the Fork.

- If any of the preceding operations is unsuccessful, the Fork is terminated and OS-9 returns an error to the caller.
- The child and parent processes execute at the same time unless the parent executes a Wait system call immediately after the Fork. In this case, the parent waits until the child dies before it resumes execution.
- Be careful when recursively calling a program that uses the Fork system call. Another child can be created with each new execution. This continues until the process table becomes full.
- Do not fork a process with a memory size of 0.

Get System Block Map

Gets a copy of the system block map

OS9 F$GBlkMp 103F 19

Entry Conditions:

X = *pointer to the 1024-byte buffer*

Exit Conditions:

D = *number of bytes per block* ($2000) (MMU block size dependent)
Y = *system memory block map size*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get System Block Map call copies the system's memory block map into the user's buffer for inspection. The OS-9 MFREE command uses this call to find out how much free memory exists.
- The support module for this call is OS9p2.

Get Module Directory

Gets a copy of the system module directory

F$GModDr 103F 1A

Entry Conditions:

X = *pointer to the 2048-byte buffer*
Y = *end of copied module directory*
U = *start address of system module directory*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get Module Directory call copies the system's module directory into the user's buffer for inspection. The OS-9 MDIR command uses this call to read the module directory.
- The support module for this call is OS9p2.

Get Process Descriptor

Gets a copy of the process's process descriptor

F$GPrDsc 103F 18

Entry Conditions:

A = *requested process ID*
X = *pointer to a 512-byte buffer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get Process Descriptor call copies a process descriptor into the calling process's buffer for inspection. The data in the process descriptor cannot be changed. The OS-9 PROCS command uses this call to get information about each existing process.
- The support module for this call is OS9p2.

Intercept

OS9 F$Icpt 103F 09

Sets a signal intercept trap

Entry Conditions:

X = *address of the intercept routine*
U = *starting address of the routine's memory area*

Exit Conditions:

Signals sent to the process cause the intercept routine to be called instead of the process being killed.

Additional Information:

- Intercept tells OS-9 to set a signal intercept trap. Then, whenever the process receives a signal, OS-9 executes the process's intercept routine.
- Store the address of the signal handler routine in Register X and the base address of the routine's storage area in Register U.
- Once the signal trap is set, OS-9 can execute the intercept routine at any time because a signal can occur at any time.
- Terminate the intercept routine with an RTI instruction.
- If a process has not used the Intercept system call to set a signal trap, the process terminates if it receives a signal.
- This is the order in which F$Icpt operates:

  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 cannot 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.

Get ID

OS9 F$ID 103F 0C

Return's a caller's process ID and user ID

Entry Conditions:

None

Exit Conditions:

A = *process ID*
Y = *user ID*

Additional Information:

- The *process ID* is a byte value in the range 1 to 255. OS-9 assigns each process a unique process ID.
- 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.
- On a single user system (such as the Color Computer 3), the user ID is inherited from CC3go, which forks the initial shell.

Link

OS9 F$Link 103F 00

Links to a memory module that has the specified name, language, and type

Entry Conditions:

A = *type/language byte*
X = *address of the module name* (See the following example.)

Exit Conditions:

A = *type/language code*
B = *attributes / revision level* (if no error)
X = *address of the last byte of the module name* + 1 (See the following example.)
Y = *module entry point absolute address*
U = *module header absolute address*

Error Output:

CC = C bit set if error encountered

Additional Information:

- The module's link count increases by one whenever Link references its name. Incrementing in this manner keeps track of how many processes are using the module.
- If the module requested is not shareable (not re-entrant), only one process can link to it at a time.
- Before the Link call:

T	E	S	T	$0D

X

- After the Link call:

T	E	S	T	$0D

X

- This is the order in which the Link call operates:

  1. OS-9 searches the module directory for a module that has the specified name, language, and type.

  2. 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 is contained in the module header.)

- 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 one of the following errors:

  - Module not found
  - Module busy (not shareable and in use)
  - Incorrect or defective module header

Load

OS9 F$Load 103F 01

Loads a module or modules from a file

Entry Conditions:

A = *language/type code;* 0 = any language/type
X = *address of the pathlist (filename)* (See the following example.)

Exit Conditions:

A = *language/type code*
B = *attributes / revision level* (if no error)
X = *address of the last byte of the pathlist (filename)* + 1 (See the following example.)
Y = *primary module entry point address*
U = *address of the module header*

Error Output:

CC = carry set if error encountered

Additional Information:

- The Load call loads one or more modules from the file specified by a complete pathlist or from the working execution directory (if an incomplete pathlist is given).
- The file must have the execute access mode bit set. It also must contain one or more with proper module headers.
- OS-9 adds all modules loaded to the system module directory. It links the first module read. The exit conditions apply only to the first module loaded.
- 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

Memory

Changes process's data area size

OS9 F$Mem
103F 07

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)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The memory call 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 and upper boundaries of data memory.
- OS-9 rounds off the size to the next page boundary. In allocating additional memory, OS-9 continues upward from the previous highest address. In deallocating unneeded memory, it continues downward from that address.

Link to a module

OS9 F$NMLink
103F 21

Links to a module; does not map the module into the user's address space

Entry Conditions:

A = *type/language byte*
X = *address of the module name*

Exit Conditions:

A = *type/language code*
B = *module revision*
X = *address of the last byte of the module name* + 1; any trailing blanks are skipped
Y = *storage requirement for the module*

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- Although this call is similar to F$Link, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLink can therefore fork larger programs than can be forked by F$Link.

Load a module

OS9 F$NMLoad
103F 22

Loads one or more modules from a file but does not map the module into the user's address space

Entry Conditions:

A = *type/language byte*
X = *address of the pathlist*

Exit Conditions:

A = *type/language code*
B = *module revision*
X = *address of the last byte of the pathlist* + 1
Y = *storage requirement for the module*

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- If you do not provide a full pathlist for this call, it attempts to load from a file in the current execution directory.
- Although this call is similar to F$Load, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLoad can therefore fork larger programs than can be forked by F$Load.

Print Error

OS9 F$Perr 103F 0F

Writes an error message to a specified path

Entry Conditions:

B = *error code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Print Error writes an error message to the standard error path for the specified process. By default, OS-9 shows:

```
ERROR #decimal number
```

- The error reporting routine is vectored. Using the Set SVC system call, you can replace it with a more elaborate reporting module.

Parse Name

OS9 F$PrsNam 103F 10

Scans an input string for a valid OS-9 name

Entry Conditions:

X = *address of the pathlist* (See the following example.)

Exit Conditions:

X = *address of the optional slash* + 1
Y = *address of the last character of the name* +1
A = *trailing byte* (delimiter character)
B = *length of the name*

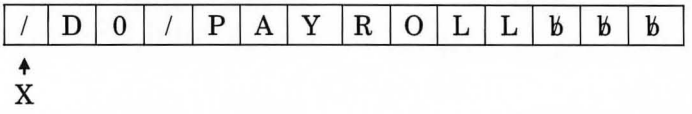
Error Output:

CC = carry set
B = *error code*
Y = *address of the first non-delimiter character in the string*

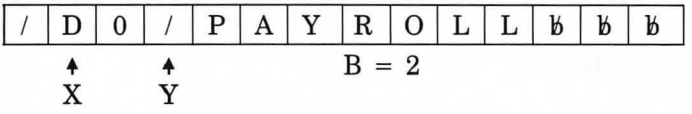
Additional Information:

- Parses, or scans, the input text string for a legal OS-9 name. It terminates the name with any character that is not a legal name character.
- Parse Name is useful for processing pathlist arguments passed to new processes.
- Because Parse Name processes only one name, you might need several calls to process a pathlist that has more than one name. As you can see from the following example, Parse Name finishes with Register Y in position for the next parse.
- If Register Y was at the end of a pathlist, Parse Name returns a bad name error. It then moves the pointer in Register Y past any space characters so that it can parse the next pathlist in a command line.

- Before the Parse Name call:



After the Parse Name call:



Search Bits

OS9 F$SchBit 103F 12

Searches a specified memory allocation bit map for a free memory block of a specified size

Entry Conditions:

D = *starting bit number*
X = *starting address of the map*
Y = *bit count (free bit block size)*
U = *ending address of the map*

Error Output:

CC = C bit set

Exit Conditions:

D = *starting bit number*
Y = *bit count*

Additional Information:

- The Search Bit call 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.

Send

Sends a signal to a specified process

OS9 F$Send 103F 08

Entry Conditions:

A = *destination's process ID*
B = *signal code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The *signal code* is a single byte value in the range 0 through 255.
- If the destination process is sleeping or waiting, OS-9 activates the process so that the process can process the signal.
- If a signal trap is set up, F$Send executes the signal processing routine (Intercept). If none was set up, the signal terminates the destination process, and the signal code becomes the exit status. (See the Wait system call.) An exception is the wakeup signal; that signal does not cause the signal intercept routine to be executed.
- Signal codes are defined as follows:

  0 = System terminate (cannot be intercepted)
  1 = Wake up the process
  2 = Keyboard terminate
  3 = Keyboard interrupt
  128-255 = User defined

- If you try to send a signal to a process that has a signal pending, OS-9 cancels the current Send call, and returns an error. 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.

Sleep

OS9 F$Sleep 103F 0A

Temporarily turns off the calling process

Entry Conditions:

X = One of the following:
sleep time (in ticks)
0 (sleep indefinitely)
1 (sleep for the remainder of the current time slice)

Exit Conditions:

X = *sleep time minus the number of ticks that the process was asleep*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If Register X contains 0, OS-9 turns the process off until it receives a signal. Putting a process to sleep is a good way to wait for a signal or interrupt without wasting CPU time.
- If Register X contains 1, OS-9 turns the process off for the remainder of the process's current time slice. It inserts the process into the active process queue immediately. The process resumes execution when it reaches the front of the queue.
- If Register X contains an integer in the range 2-255, OS-9 turns off the process for the specified number of ticks, *n*. It inserts the process into the active process queue after *n-1* ticks. The process resumes execution when it reaches the front of the queue. If the process receives a signal, it awakens before the time has elapsed.
- When you select processes among multiple windows, you might need to set sleep for two ticks.

Set Priority

OS9 F$SPrior 103F 0D

Changes the priority of a process

Entry Conditions:

A = *process ID*
B = *priority*
0 = lowest
255 = highest

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Set Priority 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.

Set SWI

Sets the SWI2 and SWI3 vectors

OS9 F$SSWI 103F 0E

Entry Conditions:

A = *SWI type code*
X = *address of the user software interrupt routine*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Sets the interrupt vectors for SWI, 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 OS-9 creates a process, it initializes all three vectors 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.

Set Time

Sets the system time and date

OS9 F$STime 103F 16

Entry Conditions:

X = *relative address of the time packet*

Error Output:

CC = C bit set
B = *error code*

Additional Information:

- Set Time sets the current system date and 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

Then, the call makes a link system call to find the clock. If the link is successful, OS-9 calls the clock initialization. The clock initialization:

1. Sets up hardware dependent functions
2. Sets up the F$Time system call via F$SSvc

Set User ID Number

F$SUser 103F 1C

Changes the current user ID without checking for errors or checking the ID number of the caller

Entry Conditions:

Y = *desired user ID number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support module for this call is OS9p1.

Time

Gets the system date and time

OS9 F$Time 103F 15

Entry Conditions:

X = *address of the area in which to store the date and time packet*

Exit Conditions:

X = *date and time*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Time call returns the current system date and time in the form of a 6-byte packet (in binary). OS-9 copies the packet 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

- Time is a part of the clock module and it does not exist if no previous call to F$Time has been made.

Unlink

OS9 F$UnLink 103F 02

Unlinks (removes from memory) a module that is not in use and that has a link count of 0

Entry Conditions:

U = *address of the module header*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Unlink unlinks a module from the current process's address space, decreases its link count by one and, if the link count becomes zero, returns the memory where the module was located to the system for use by other processes.
- You cannot unlink system modules or device drivers that are in use.
- Unlink operates in the following order:

  1. Unlink tells OS-9 that the calling process no longer needs the module.
  2. OS-9 decreases the module's link count by one.
  3. When the resulting link count is zero, OS-9 destroys the module.

     If any other process is using the module, the module's link count cannot fall to zero. Therefore, OS-9 does not destroy the module.

- If you pass a bad address, Unlink cannot find a module in the module directory and does not return an error.

Unlink A Module By Name

F$UnLoad 103F 1D

Decrements a specified module's link count, and removes the module from memory if the resulting link count is zero

Entry Conditions:

A = *module type*
X = *pointer to module name*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call differs from Unlink in that it uses a pointer to the module name, instead of the address of the module header.
- The support module for this call is OS9p2.

Wait

Temporarily turns off a calling process

OS9 F$Wait 103F 04

Entry Conditions: None

Exit Conditions:

A = *deceased child process's ID*
B = *deceased child process's exit status code* (if no error)

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- The Wait call turns off the calling process until a child process *dies*, either by executing an Exit system call, or by receiving a signal. The Wait call helps you save system time.
- OS-9 returns the child's process's ID and exit status to the parent. If the child died because of a signal, the exit status byte (Register B) contains the signal code.
- If the caller has several children, OS-9 activates the caller when the first one dies. Therefore, you need to use one Wait system call to detect the termination of each child.
- OS-9 immediately reactivates the caller if a child dies before the Wait call. If the caller has no children, Wait returns an error. (See the Exit system call for more information.)
- If the Wait call returns with the carry bit set, the Wait function was not successful. If the carry bit is cleared, Wait functioned normally and any error that occurred in the child process is returned in Register B.

I/O User System Calls

Attach

OS9 I$Attach 103F 80

Attaches a device to the system or verifies device attachment

Entry Conditions:

A = *access mode*
X = *address of the device name string*

Exit Conditions:

X = *updated past device name*
U = *address of the device table entry*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Attach does not reserve the device. It only prepares the device for later use by any process.
- OS-9 installs most devices automatically on startup. 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 any special device capabilities
  1 = Read only
  2 = Write only
  3 = Update (read and write)

- Attach operates in this sequence:

  1. OS-9 searches the system module to see if memory contains a device descriptor that has the same name as the device.

  2a. OS-9's next operation depends on whether or not the device is already attached. If OS-9 finds the descriptor and if the device is not already attached, OS-9 links the device's file manager and device driver. It then places the address of the manager and the driver 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 initializes the hardware.

  b. If OS-9 finds the descriptor, and if the device is already attached, OS-9 verifies the attachment.

Change Directory

OS9 I$Chgdir 103F 86

Changes the working directory of a process to a directory specified by a pathlist

Entry Conditions:

A = *access mode*
X = *address of the pathlist*

Exit Conditions:

X = *updated past pathlist*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the access mode is read, write, or update, OS-9 changes the current data directory. If the access mode is execute, OS-9 changes the current execution directory.
- 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 (read and write)
  4 = Execute

Close Path

Terminates an I/O path

OS9 I$Close 103F 8F

Entry Conditions:

A = *path number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Close Path terminates the I/O path to the file or device specified by *path number*. Until you use another Open, Dup, or Create system call for that path, you can no longer perform I/O to the file or device.
- If you close a path to a single-user device, the device becomes available to other requesting processes. OS-9 deallocates internally managed buffers and descriptors.
- The Exit system call automatically closes all open paths. Therefore, you might not need to use the Close Path system call to close some paths.
- Do not close a standard I/O path unless you want to change the file or device to which it corresponds.
- Close Path performs an implied I$Detach call. If it causes the device link count to become 0, the device termination routine is executed. See I$Detach for additional information.

Create File

OS9 I$Create 103F 83

Creates and opens a disk file

Entry Conditions:

A = *access mode* (write or update)
B = *file attributes*
X = *address of the pathlist;* (See the following example.)

Exit Conditions:

A = *path number*
X = *address of the last byte of the pathlist* + 1; skips any trailing blanks (See the following example.)

Error Output:

CC = carry set on error
B = *error code* (if any)

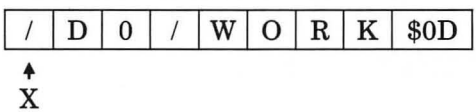
Additional Information:

- OS-9 parses the pathlist and enters the new filename in the specified directory. If you do not specify a directory, OS-9 enters the new filename in the the working directory.
- OS-9 gives the file the attributes passed in Register B, which has bits defined as follows:

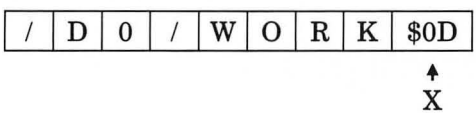
Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Shareable file

- The access mode parameter passed in Register A must have the write bit set if any data is to be written. These access codes are defined as follows: 2 = write; 3 = update. The mode affects the file only until the file is closed.

- You can reopen the file in any access mode allowed by the file attributes. (See the Open system call.)
- Files opened for write can allow faster data transfer than those opened for update because update sometimes needs to pre-read sectors.
- If the execute bit (Bit 2) is set, the file is created in the working execution directory instead of the working data directory.
- Create File causes an implicit I$Attach call. If the device has not previously been attached, the device's initialization routine is called.
- Later I/O calls use the path number to identify the file, until the file is closed.
- OS-9 does not allocate data storage for a file at creation. Instead, it allocates the storage either automatically when you first issue a write or explicitly by the Setstat 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 Open.
- You cannot use Create to make directories. (See the Make Directory system call for instructions on how to do make directories.)
- Before the Create File call:



After the Create File call:



Delete File

OS9 I$Delete 103F 87

Deletes a specified disk file

Entry Conditions:

X = *address of the pathlist* (See the following example.)

Exit Conditions:

X = *address of the last byte of the pathlist* + 1; any trailing blanks are skipped (See the following example.)

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Delete File call 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. The caller must have non-shareable write access to the file or an error results.

Example:

Before the Delete File call:

/	D	0	/	W	O	R	K	♭	♭	♭	M	E	M	O	$0D

X

After the Delete File call:

/	D	0	/	W	O	R	K	♭	♭	♭	M	E	M	O	$0D

X

Delete A File

OS9 I$DeletX 103F 90

Deletes a file from the current data or current execution directory

Entry Conditions:

A = *access mode*
X = *address of the pathlist*

Exit Conditions:

X = *address of the last byte of the pathlist+1*; any trailing blanks are skipped

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Delete A File call removes the disk file specified by the selected pathlist. This function is similar to I$Delete except that it accepts an access mode byte. If the access mode is execute, the call selects the current execution directory. Otherwise, it selects the current data directory.
- If a complete pathlist is provided (the pathlist begins with a slash (/), the access mode the call ignores the access mode.
- Only use this call to delete a file. If you attempt to use I$DeletX to delete a device, the system returns an error.

Detach Device

OS9 I$Detach 103F 81

Removes a device from the system device table

Entry Conditions:

U = *address of the device table entry*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Detach Device call removes a device from both the system and the system device table, assuming the device is not being used by another process. You must use this call to detach devices attached using the Attach system call. Attach and Detach are both used mainly by the IO manager. SCF also uses Attach and Detach to set up its second device (echo device).
- This is the sequence of the operation of Detach Device:
  1. Detach Device calls the device driver's termination routine. Then, OS-9 deallocates any memory assigned to the driver.
  2. OS-9 unlinks the associated device driver and file manager modules.
  3. OS-9 then removes the driver, as long as no other module is using that driver.

Duplicate Path

Returns a synonymous path number

OS9 I$Dup 103F 82

Entry Conditions:

A = *old path number* (number of path to duplicate)

Exit Conditions:

A = *new path number* (if no error)

Error Output:

B = *error code* (if error encountered)
CC = carry set on error

Additional Information:

- The Duplicate Path returns another, synonymous path number for the file or device specified by the *old path number*.
- The shell uses the Duplicate Path call when it redirects I/O.
- System calls can use either path number (old or new) to operate on the same file or device.
- Make sure that no more than one process is performing I/O on any one path at the same time. Concurrent I/O on the same path can cause unpredictable results with RBF files.
- The I$Dup call always uses the lowest available path number. This lets you manipulate standard I/O paths to contain any desired paths.

Get Status

OS9 I$GetStt 103F 8D

Returns the status of a file or device

Entry Conditions:

A = *path number*
B = *function code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Status is a *wildcard* call. Use it 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 "Get Status System Calls" section.

Make Directory

OS9 I$MakDir 103F 85

Creates and initializes a directory

Entry Conditions:

B = *directory attributes*
X = *address of the pathlist*

Exit Conditions:

X = *address of the last byte of the pathlist* +1; Make Directory skips trailing blanks.

Error Output:

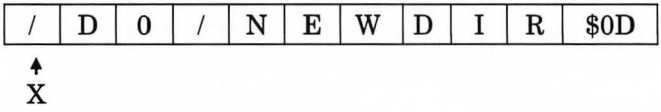
B = *error code* (if any)
CC = carry set on error

Additional Information:

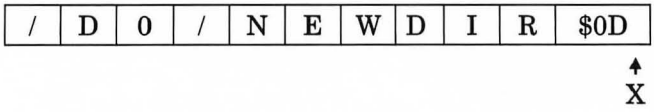
- The Make Directory call creates and initializes a directory as specified by the pathlist. The directory contains only two entries, one for itself (.) and one for its parent directory (..)
- OS-9 makes the calling process the owner of the directory.
- Because the Make Directory call does not open the 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	Single-user
7	Don't care

- Before the Make Directory call:



After the Make Directory call:



Open Path

OS9 I$Open 103F 84

Opens a path to an existing file or device as specified by the pathlist

Entry Conditions:

A = *access mode* (D S PE PW PR E W R)
X = *address of the pathlist* (See the following example.)

Exit Conditions:

A = *path number*
X = *address of the last byte of the pathlist* + 1

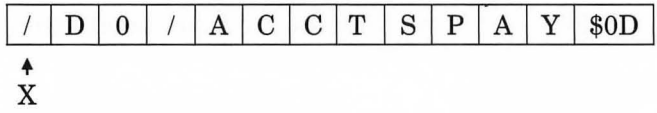
Error Output:

B = *error code* (if any)
CC = carry set on error

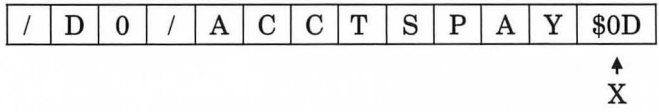
Additional Information:

- 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.
- OS-9 returns a path number for later system calls to use to identify the file.
- The access mode parameter lets you specify which read and/or write operations are to be permitted. When set, each access mode bit enables one of the following: Write, Read, Read and Write, Update, 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 can access a file unless the appropriate public permission bits are set.

- The update mode might be slightly slower than the others because it might require pre-reading of sectors for random access of bytes within sectors.
- Several processes (users) can open files at the same time. Each device has an attribute that specifies whether or not it is shareable.
- Before the Open Path call:



After the Open Path call:



- If the single-user bit is set, the file is opened for single-user access regardless of the settings of the file's permission bits.
- You must open directory files for read or write if the directory bit (Bit 7) is set in the access mode.
- Open Path always uses the lowest path number available for the process.

Read

Reads *n* bytes from a specified path

OS9 I$Read 103F 89

Entry Conditions:

A = *path number*
Y = *number of bytes to read*
X = *address in which to store the data*

Exit Conditions:

Y = *number of bytes read*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Read call reads the specified number of bytes from the specified path. It returns the data exactly as read from the file/device, without additional processing or editing. The path must be opened in the read or update mode.
- If there is not enough data in the specified file to satisfy the read request, the call reads fewer bytes than requested but an end-of-file error is **not** returned. After all data in a file is read, the next I$Read call returns an end-of-file error.
- If the specified file is open for update, the record read is locked out on RBF-type devices.
- The keyboard terminate, keyboard interrupt, and end-of-file characters are filtered out of the Entry Conditions data on SCF-type devices unless the corresponding entries in the path descriptor have been set to zero. You might want to modify the device descriptor so that these values are initialized to zero when the path is opened.

- The call reads the number of bytes requested unless Read encounters any of the following:
  - An end-of-file character
  - An end-of-record character (SCF only)
  - An error

Read Line With Editing

Reads a text line with editing

OS9 I$ReadLn 103F 8B

Entry Conditions:

A = *path number*
X = *address at which to store data*
Y = *maximum number of bytes to read*

Exit Conditions:

Y = *number of bytes read*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Read Line is similar to Read. The difference is that Read Line reads the input file or device until it encounters a carriage return character or until it reaches the maximum byte count specified, whichever comes first. The Read Line also automatically activates line editing 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.
- SCF 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, Read Line does not accept it and a PD.OVF character (usually a bell) is echoed.
- After one Read Line call reads all data in a file, the next Read Line call generates an end-of-file error.
- (For more information about line editing, see "SCF Line Editing Functions" in Chapter 6.)

Seek

Repositions a file pointer

OS9 I$Seek 103F 88

Entry Conditions:

A = *path number*
X = *MS 16 bits of the desired file position*
U = *LS 16 bits of the desired file position*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Seek Call 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.
- You can perform a seek to any value, regardless of the file's size. Later writes automatically expand the file to the required size (if possible). Later reads, however, return an end-of-file condition. Note that a seek to Address 0 is the same as a rewind operation.
- OS-9 usually ignores seeks to non-random access devices, and returns without error.
- On RBF devices, seeking to a new disk sector causes the internal disk buffer to be rewritten to disk if it has been modified. Seek does not change the state of record locking.

Set Status

Sets the status of a file or device

OS9 I$SetStt 103F 8E

Entry Conditions:

A = *path number*
B = *function code*
Other registers depend on the function code.

Error Output:

B = *error code* (if any)
CC = carry set on error
Other registers depend on the function code.

Additional Information:

- Set Status is a wildcard call. Use it 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 "Set Status System Calls" section.

Write

Writes to a file or device

OS9 I$Write 103F 8A

Entry Conditions:

A = *path number*
X = *starting address of data to write*
Y = *number of bytes to write*

Exit Conditions:

Y = *number of bytes written*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Write system call writes to the file or device associated with the path number specified.
- Before using Write, be sure the path is opened or created in the Write or Update access mode. OS-9 writes data to the file or device without processing or editing the data. OS-9 automatically expands the file if you write data past the present end-of-file.

Write Line

OS9 I$WritLn 103F 8C

Writes to a file or device until it encounters a carriage return

Entry Conditions:

A = *path number*
X = *address of the data to write*
Y = *maximum number of bytes to write*

Exit Conditions:

Y = *number of bytes written*

Error Output:

B = *error code* (if any)
CC = *carry set on error*

Additional Information:

- Writes to the file or device that is associated with the path number specified.
- Write Line is similar to Write. The difference is that Write Line writes data until it encounters a carriage return character. It also activates line editing 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.
- Before using Write Line, be sure the path is opened or created in the write or update access mode.
- (For more information about line editing, see "SCF Line Editing Functions" in Chapter 6.)

Privileged System Mode Calls

Set an alarm

OS9 F$Alarm 103F 1E

Sets an alarm to ring the bell at a specified time

Entry Conditions:

X = *relative address of time packet*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- When the system reaches the specified alarm time, it rings the bell for 15 seconds.
- The time packet is identical to the packet used in the F$STime call. See F$STime for additional information on the format of the packet.
- All alarms begin at the start of a minute and any seconds in the packet are ignored.
- The system is limited to one alarm at a time.

Allocate 64

OS9 F$All64 103F 30

Dynamically allocates 64-byte blocks of memory

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*

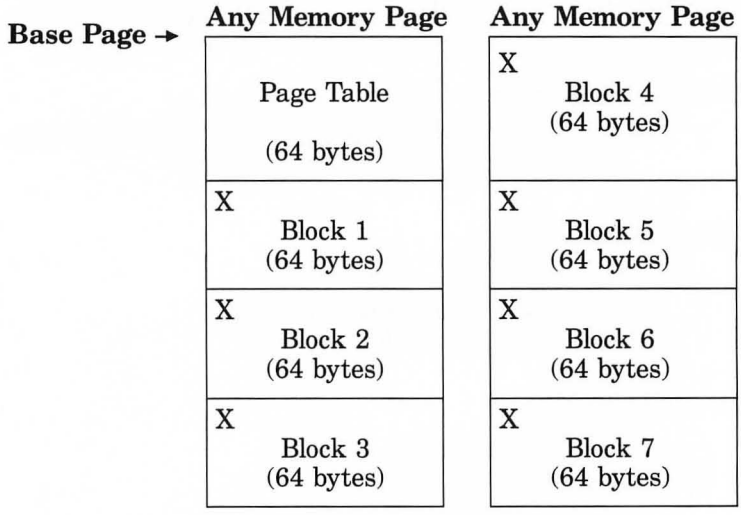
Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Allocate 64 system call allocates the 64-byte blocks of memory by splitting pages (256-byte sections) into four sections.
- OS-9 uses the first 64 bytes of the base page as a page table. This table contains the page number (most significant byte of the address) 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 System Memory system call (F$SRqMem) executes automatically.
- The first byte of each block contains the block number. Routines that use the Allocate 64 call should not alter this byte.

- The following diagram shows how seven blocks might be allocated:



Allocate High RAM

Allocate system memory from high physical memory

OS9 F$AlHRam 103F 53

Entry Conditions:

B = *number of blocks*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- This call searches for the desired number of contiguous free RAM blocks, starting its search at the top of memory. F$AllHRam is similar to F$AllRAM except F$AllRAM begins its search at the bottom of memory.
- Screen allocation routines use this call to provide a better chance of finding the necessary memory for a screen.

Allocate Image

OS9 F$AllImg 103F 3A

Allocates RAM blocks for process DAT image

Entry Conditions:

A = *starting block number*
B = *number of blocks*
X = *process descriptor pointer*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use the Allocate Image system call to allocate a data area for a process. The blocks that Allocate Image defines might not be contiguous.
- The support module for this system call is OS9p1.

Allocate Process Descriptor

Allocates and initializes a 512-byte process descriptor

OS9 F$AllPrc 103F 4B

Entry Conditions: None

Exit Conditions:

U = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The process descriptor table houses the address of the descriptor. Initialization of the process descriptor consists of clearing the first 256 bytes of the descriptor, setting up the state as a system state, and marking as unallocated as much of the DAT image as the system allows—typically, 60-64 kilobytes.
- The support module for this system call is OS9p2. The call also branches to the F$SRqMem call.

Allocate RAM

OS9 F$AllRAM 103F 39

Searches the memory block map for the desired number of contiguous free RAM blocks

Entry Conditions:

B = *number of blocks*

Exit Conditions:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The support module for this system call is OS9p1.

Allocate Process Task Number

OS9 F$AllTsk 103F 3F

Determines whether OS-9 has assigned a task number to the specified process

Entry Conditions:

X = *process descriptor pointer*

Error Output:

CC = C bit set
B = *appropriate error code*

Additional Information:

- If the process does not have a task number, OS-9 allocates a task number and copies the DAT image into the DAT hardware.
- The support module for this call is OS9p1. Allocate Process Task number also branches to F$ResTsk and F$SetTsk.

Insert Process

OS9 F$AProc 103F 2C

Inserts a process into the queue for execution

Entry Conditions:

X = *address of the process descriptor*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Insert Process system call inserts a process into the active process queue so that OS-9 can schedule the process for execution.
- OS-9 sorts all processes in the queue by process age (the 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, OS-9 sets its age 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. OS-9 gives such a process higher priority because the process usually is executing critical routines that affect shared system resources.

Bootstrap System

OS9 F$Boot 103F 35

Links either the module named Boot or the module specified in the INIT module

Entry Conditions: None

Error Output:

CC = *C bit set on error*
B = *appropriate error code*

Additional Information:

- When it calls the linked module, Boot expects to receive a pointer giving it the location and size of an area in which to search for the new module.
- The support module for this call is OS9p1. Bootstrap System also branches to the F$Link and F$VModul system calls.

Bootstrap Memory Request

OS9 F$BtMem 103F 36

Allocates the requested memory (rounded to the nearest block) as contiguous memory in the system's address space

Entry Conditions:

D = *byte count requested*

Exit Conditions:

D = *byte count granted*
U = *pointer to memory allocated*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This call is identical to F$SRqMem.
- The Bootstrap Memory Request support module is OS9p1.

Clear Specified Block

Marks blocks in the process DAT image as unallocated

OS9 F$ClrBlk 103F 50

Entry Conditions:

B = *number of blocks*
U = *address of first block*

Exit Conditions: None

Additional Information:

- After Clear Specified Block deallocates blocks, the blocks are free for the process to use for other data or program areas. If the block address passed to Clear Specified Block is invalid or if the call attempts to clear the stack area, it returns E$IBA.
- The support module for the call is OS9p2.

DAT to Logical Address

OS9 F$DATLog 103F 44

Converts a DAT image clock number and block offset to its equivalent logical address

Entry Conditions:

B = *DAT image offset*
X = *block offset*

Exit Conditions:

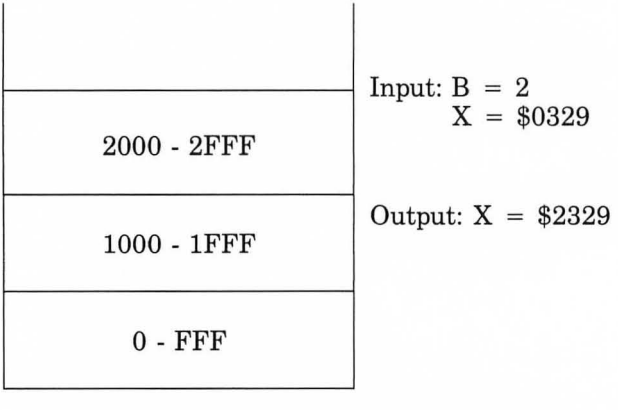
X = *logical address*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- Following is a sample conversion:



- The support module for this call is OS9p1.

Deallocate Image RAM Blocks

Deallocates image RAM blocks

OS9 F$DelImg 103F 3B

Entry Conditions:

A = *number of starting block*
B = *block count*
X = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This system call deallocates memory from a process's address space. It frees the RAM for system use and frees the DAT image for the process. Its main use is to let the system clean up after a process death.
- The support module for this call is OS9p2.

Deallocate Process Descriptor

Returns a process descriptor's memory to a free memory pool

OS9 F$DelPrc 103F 4C

Entry Conditions:

A = *process ID*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- Use this call to clear the system scratch memory and stack area associated with the process.
- The support module for this call is OS9p2.

Deallocate RAM blocks

OS9 F$DelRAM 103F 51

Clears a block's RAM In Use flag in the system memory block map

Entry Conditions:

B = *number of blocks*
X = *starting block number*

Exit Conditions: None

Additional Information:

- The Deallocate RAM Blocks call assumes the blocks being deallocated are not associated with any DAT image.
- The support module for this call is OS9p2.

Deallocate Task Number

OS9 F$DelTsk 103F 40

Releases the task number that the process specified by the passed descriptor pointer

Entry Conditions:

X = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The support module for this call is OS9p1.

Link Using Module Directory Entry

Performs a link using a pointer to a module directory entry

OS9 F$ELink 103F 4D

Entry Conditions:

B = *module type*
X = *pointer to module directory entry*

Exit Conditions:

U = *module header address*
Y = *module entry point*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This call differs from Link in that you supply a pointer to the module directory entry rather than a pointer to a module name.
- The support module for this call is OS9p1.

Find Module Directory Entry

Returns a pointer to the module directory entry

OS9 F$FModul 103F 4E

Entry Conditions:

A = *module type*
X = *pointer to the name string*
Y = *DAT image pointer* (for name)

Exit Conditions:

A = *module type*
B = *module revision number*
X = *updated name string;* (if Register A contains 0 on entry)
U = *module directory entry pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Find Module Directory Entry call returns a pointer to the module directory entry for the first module that has a name and type matching the specified name and type. If you pass a module type of zero, the system call finds any module.
- The support module for this call is OS9p1.

Find 64

OS9 F$Find64 103F 2F

Returns the address of a 64-byte memory block

Entry Conditions:

A = *block number*
X = *address of the block*

Exit Conditions:

Y = *address of the block*
CC = carry set if block not allowed or not in use

Additional Information:

- OS-9 uses Find 64 to find path descriptors when given their block number. The block number can be any positive integer.

Get Free High Block

OS9 F$FreeHB 103F 3E

Searches the DAT image for the highest set of contiguous free blocks of the specified size

Entry Conditions:

B = *block count*
Y = *DAT image pointer*

Exit Conditions:

A = *starting block number*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Get Free High Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is OS9p1.

Get Free Low Block

OS9 F$FreeLB 103F 3D

Searches the DAT image for the lowest set of contiguous free blocks of the specified size

Entry Conditions:

B = *block count*
Y = *DAT image pointer*

Exit Conditions:

A = *starting block number*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Get Free Low Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is OS9p1.

Compact Module Directory

Compacts the entries in the module directory

OS9 F$GCMDir 103F 52

Entry Conditions: None

Exit Conditions: None

Additional Information:

- This function is only for internal OS-9 system use. You should never call it from a program.

Get Process Pointer

Gets a pointer to a process

F$GProcP 103F 37

Entry Conditions:

A = *process ID*

Exit Conditions:

B = *pointer to process descriptor* (if no error)

Error Output:

CC = carry set on error
B = *error code* (If an error occurs (E$BPrcID))

Additional Information:

- The Get Process Pointer call translates a process ID number to the address of its process descriptor in the system address space. Process descriptors exist only in the system task address space. Because of this, the address returned only refers to system address space.
- The support module for this call is OS9p2.

I/O Delete

OS9 F$IODel 103F 33

Deletes an I/O module that is not being used

Entry Conditions:

X = *address of an I/O module*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The I/O Delete deletes the specified I/O module from the system, if the module is not in use. This system call is used mainly by the I/O MANAGER, and can be of limited or no use for other applications.
- This is the order in which I/O Delete operates:
  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 Delete returns information to the Unlink system call after determining whether a device is busy.

I/O Queue

OS9 F$IOQu 103F 2B

Inserts the calling process into another process's I/O queue, and puts the calling process to sleep

Entry Conditions:

A = *process number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The I/O Queue call links the calling process into the I/O queue of the specified process and performs an untimed sleep. The IO Manager and the file managers are primary and extensive users of I/O Queue.
- Routines associated with the specified process send a wake-up signal to the calling process.

Set IRQ

OS9 F$IRQ 103F 2A

Adds a device to or removes it from the polling table

Entry Conditions:

D = *address of the device status register*
X = 0 (to remove a device) or *the address of a packet* (to add a device)
- the address at X is the flip byte
- the address at X+1 is the mask byte
- the address at X+2 is the priority byte

Y = *address of the device IRQ service routine*
U = *address of the service routine's memory area*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Set IRQ is used mainly by device driver routines. (See "Interrupt Processing" in Chapter 2 for a complete discussion of the interrupt polling system.)
- Packet Definitions:

  The Flip Byte. Determines 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).

  The 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.

  The Priority Byte. Contains the device priority number (0 = lowest priority, 255 = highest priority).

Load A From Task B

Loads A from 0,X in task B

F$LDABX 103F 49

Entry Conditions:

B = *task number*
X = *pointer to data*

Exit Conditions:

A = *byte at 0,X in task address space*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The value in Register X is an offset value from the beginning address of the Task module. The Load A From Task B call returns one byte from this logical address. Use this system call to get one byte from the current process's memory in a system state routine.

Get One Byte

Loads A from [X, [Y]]

F$LDAXY 103F 46

Entry Conditions:

X = *block offset*
Y = *DAT image pointer*

Exit Conditions:

A = *contents of byte at DAT image (Y) offset by X*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get One Byte system call gets the contents of one byte in the specified memory block. The block is specified by the DAT image in (Y), offset by (X). The call assumes that the DAT image pointer is to the actual block desired, and that X is only an offset within the DAT block. The value in Register X must be less than the size of the DAT block. OS-9 does not check to see if X is out of range.

Get Two Bytes

F$LDDDXY 103F 48

Loads D from [D+X],[Y]

Entry Conditions:

D = *Offset to the offset within the DAT image*
X = *Offset within the DAT image*
Y = *DAT image pointer*

Exit Conditions:

D = *contents of two bytes at [D+X,Y]*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Get Two Bytes loads two bytes from the address space described by the DAT image pointer. If the DAT image pointer is to the entire DAT, then make D+X equal to the process address for data. If the DAT image is not the entire image (64K), then you must adjust D+X relative to the beginning of the DAT image. Using D+X lets you keep a local pointer within a block, and also lets you point to an offset within the DAT image that points to a specified block number.

Map Specific Block

F$MapBlk 103F 4F

Maps the specified block(s) into unallocated blocks of process space

Entry Conditions:

X = *starting block number*
B = *number of blocks*

Exit Conditions:

U = *address of first block*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The system maps blocks from the top down. It maps new blocks into the highest available addresses in the address space. See Clear Specified Block for information on unmapping.

Move Data

F$Move 103F 38

Moves data bytes from one address space to another

Entry Conditions:

A = *source task number*
B = *destination task number*
X = *source pointer*
Y = *byte count*
U = *destination pointer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- You can use the Move Data system call to move data bytes from one address space to another, usually from system to user, or vice versa.
- The support module for this call is OS9p1.

Next Process

OS9 F$NProc 103F 2D

Executes the next process in the active process queue

Entry Conditions: None

Exit Conditions:

Control does not return to caller.

Additional Information:

- The Next Process system call takes the next process out of the active process queue and initiates its execution. If the queue contains no process, OS-9 waits for an interrupt, and then checks the queue again.
- The process calling Next Process must already be in one of the three process queues. If it is not, it becomes unknown to the system even though the process descriptor still exists and can be displayed by a PROCS command.

Release A Task

F$RelTsk 103F 43

Releases a specified DAT task number from use by a process, making the task's DAT hardware available for use by another task

Entry Conditions:

B = *task number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support module for this call OS9p1.

Reserve Task Number

Reserves a DAT task number

F$ResTsk 103F 42

Entry Conditions: none

Exit Conditions:

B = *task number* (if no error)

Error Output:

CC = carry set on error
B = *error code* if an error occurs

Additional Information:

- The Reserve Task Number call finds a free DAT task number, reserves it, and returns the task number to the caller. The caller often then assigns the task number to a process.
- The support module for this call is OS9p1.

Return 64

Deallocates a 64-byte block of memory

OS9 F$Ret64 103F 31

Entry Conditions:

A = *block number*
X = *address of the base page*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- See the Allocate 64 system call for more information.

Set Process DAT Image

Copies all or part of the DAT image into a process descriptor

F$SetImg 103F 3C

Entry Condition:

A = *starting image block number*
B = *block count*
X = *process descriptor pointer*
U = *new image pointer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- While copying part or all of the DAT image, this system call also sets an image change flag in the process descriptor. This flag guarantees that as a process returns from the system call. The call updates the hardware to match the new process DAT image.
- The support module for this call is OS9p1.

Set Process Task DAT Registers

Writes to the hardware DAT registers

F$SetTsk 103F 41

Entry Conditions:

X = *pointer to process descriptor*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call sets the process task hardware DAT registers, and clears the image change flag in the process descriptor. It also writes to DAT RAM the process's segment address information.
- The support module for this call is OS9p1.

System Link

F$SLink 103F 34

Adds a module from outside the current address space into the current address space

Entry Conditions:

A = *module type*
X = *module name string pointer*
Y = *name string DAT image pointer*

Exit Conditions:

A = *module type*
B = *module revision (if no error)*
X = *updated name string pointer*
Y = *module entry point*
U = *module pointer*

Error Output:

CC = carry set on error
B = *error code* (If an error occurs)

Additional Information:

- The I/O System uses the System Link call to link into the current process's address space those modules specified by a device name in a user call. User calls such as Create File and Open Path use this System Link.
- The support module for this call is OS9p1.

Request System Memory

OS9 F$SRqMem 103F 28

Allocates a block of memory of the specified size from the top of available RAM

Entry Conditions:

D = *byte count*

Exit Conditions:

U = *starting address of the memory area*
D = *new memory size*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Request System Memory call rounds the size request to the next page boundary.
- This call allocates memory only for system address space.

Return System Memory

Deallocates a block of contiguous pages

OS9 F$SRtMem 103F 29

Entry Conditions:

U = *starting address of memory to return;* must point to an even page boundary
D = *number of bytes to return*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Register U must point to an even page boundary.
- This call deallocates memory for system address space only.

Set SVC

Adds or replaces a system call

OS9 F$SSvc 103F 32

Entry Conditions:

Y = *address of the system call initialization table*

Error Output:

CC = C bit set
B = *error code*

Additional Information:

- Set SVC adds or replaces a system call, which you have written, to OS-9's user and system mode system call tables.
- 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:

Relative Address	Use	
$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	← More entries
	$80	← End-of-table mark

- If the most significant bit of the function code is set, OS-9 updates the system table.

  If the most significant bit of the function code is not set, OS-9 updates the system and user tables.

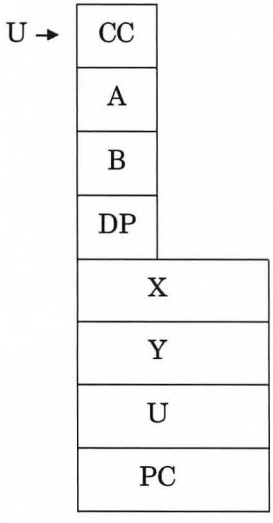
- The function request codes are in the range $29-$34. IO calls are in the range $80-$90

- To use a privileged system call, you must be executing a program that resides in the system map and that executes in the system state.

- The system call handler routine must process the system call and return from the subroutine with an RTS instruction.

- The handler routine might 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
$00	R$CC
$01	R$D
$01	R$A
$02	R$B
$03	R$DP
$04	R$X
$06	R$Y
$08	R$U
$0A	R$PC

Codes $70-$7F are reserved for user definition.

Store A Byte In A Task

Stores A at 0,X in Task B

F$STABX 103F 4A

Entry Conditions:

A = *byte to store*
B = *destination task number*
X = *logical destination address*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call is similar to the assembly language instruction "STA 0,X". The difference is that in the system call, X refers to an address in the given task's address space, instead of the current address space.
- The support module for this system call is OS9p1.

Install virtual interrupt

Installs a virtual interrupt handler routine

OS9 F$VIRQ 103F 27

Entry Conditions:

D = *initial count value*
X = 0 to delete entry
    1 to install entry
Y = *address of 5-byte packet*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- Install VIRQ for use with devices in the Multi-Pak Expansion Interface. This call is explained in detail in Chapter 2.

Validate Module

OS9 F$VModul 103F 2E

Checks the module header parity and CRC bytes of a module

Entry Conditions:

D = *DAT image pointer*
X = *new module block offset*

Exit Conditions:

U = *address of the module directory entry*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the values of the specified module are valid, OS-9 searches the module directory for a module with the same name. If one exists, OS-9 keeps in memory the module that has the higher revision level. If both modules have the save revision level, OS-9 retains the module in memory.

Get Status System Calls

You use the Get Status system calls with the RBF manager subroutine GETSTA. The OS-9 Level Two system reserves function Codes 7-127 for use by Microware. You can define Codes 128-255 and their parameter-passing conventions for your own use. (See the sections on device drivers in Chapters 4, 5, and 6.)

The Get Status routine passes the register stack and the specified function code to the device driver.

Following are the Get Status functions and their codes.

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

Entry Conditions:

A = *path number*
B = $00
X = *address to receive status packet*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use SS.OPT to determine the current settings for editing functions, such as echo and auto line feed.

SS.RDY

(Function code $01). Tests for data available on SCF-supported devices

Entry Conditions:

A = *path number*
B = $01

Exit Conditions:

If the device is ready:
CC = carry clear
B = $00

If the device is not ready:
CC = carry set
B = $F6 (E$SRNDY)

Error Output:

CC = carry set
B = *error code*

SS.SIZ

(Function code $02). Gets the current file size on a RBF-supported devices only

Entry Conditions:

A = *path number*
B = $02

Exit Conditions:

X = *most significant 16 bits of the current file size*
U = *least significant 16 bits of the current file size*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.POS

(Function code $05). Gets the current file position (RBF-supported devices only)

Entry Conditions:

A = *path number*
B = $05

Exit Conditions:

X = *most significant 16 bits of the current file position*
U = *least significant 16 bits of the current file position*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.EOF

(Function code $06). Tests for the end of the file (EOF)

Entry Conditions:

A = *path number*
B = $06

Exit Conditions:

If there is no EOF:
CC = carry clear
B = $00

If there is an EOF:
CC = carry set
B = $D3 (E$EOF)

Error Output:

CC = carry set
B = *error code*

SS.DevNm

(Function Code $0E). Returns a device name

Entry Conditions:

A = *path number*
B = $0E
X = *address of 32-byte buffer for name*

Exit Conditions:

X = *address of buffer, name moved to buffer*

SS.DSTAT

(Function code $12). Returns the display status

Entry Conditions:

A = *path number*
B = $12

Exit Conditions:

A = *color code of the pixel at the cursor address*
X = *address of the graphics display memory*
Y = *graphics cursor address;* X = MSB, Y = LSB

Additional Information:

- This function is supported only with the VDGINT module and deals with VDG-compatible graphics screens. See SS.AAGBf for information regarding Level Two operation.

SS.JOY

(Function code $13). Returns the joystick values

Entry Conditions:

A = *path number*
B = $13
X = *joystick number*
0 = (right joystick)
1 = (left joystick)

Exit Conditions:

A = *fire button down*
0 = none
1 = Button 1
2 = Button 2
3 = Button 1 and Button 2
X = *selected joystick x value* (0-63)
Y = *selected joystick y value* (0-63)

Note: Under Level 1, the following values are returned by this call:

A = fire button status
$FF = fire button is on
$00 = fire button is off

SS.AlfaS

(Function code $1C). Returns VDG alpha screen memory information

Entry Conditions:

A = *path number*
B = $1C

Exit Conditions:

A = *caps lock status*
    $00 = lower case
    $FF = upper case
X = *memory address of the buffer*
Y = *memory address of the cursor*

Additional Information:

- SS.AlfaS maps the screen into the user address space. The call requires a full block of memory for screen mapping. This call is only for use with VDG text screens handled by VDGINT.
- The support module for this call is VDGINT.
- **Warning:** Use extreme care when poking the screen, since other system variables reside in screen memory. Do not change any addresses outside of the screen.

SS.Cursr

(Function code $25). Returns VDG alpha screen cursor information

Entry Conditions:

A = *path number*
B = $25

Exit Conditions:

A = *character code of the character at the current cursor address*
X = *cursor X position (column)*
Y = *cursor Y position (row)*

Additional Information:

- SS.Cursr returns the character at the current cursor position. It also returns the X-Y address of the cursor relative to the current device's window or screen. SS.Cursr works only with text screens.
- The support module for this call is VDGINT.

SS.ScSiz

(Function code $26). Returns the window or screen size

Entry Conditions:

A = *path number*
B = $26

Exit Conditions:

X = *number of columns on screen/window*
Y = *number of rows on screen/window*

Additional Information:

- Use this call to determine the size of an output screen. The values returned depend on the device in use:

  For non-CCIO devices, the call returns the values following the XON/XOFF bytes in the device descriptor.

  For CCIO devices, the call returns the size of the window or screen in use by the specified device.

  For window devices. the call returns the size of the current working area of the window.

- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.KySns

(Function code $27). Returns key down status

Entry Conditions:

A = *path number*
B = $27

Exit Conditions:

A = *keyboard scan information*

Additional Information:

- Accumulator A returns with a bit pattern representing eight keys. With each keyboard scan, OS9 updates this bit pattern. A set bit (1) indicates that a key was pressed since the last scan. A clear bit (0) indicates that a key was not pressed. Definitions for the bits are as follows:

Bit	Key
0	[SHIFT]
1	[CTRL] or [CLEAR]
2	[ALT] or [@]
3	[↑] (up arrow)
4	[↓] (down arrow)
5	[←] (left arrow)
6	[→] (right arrow)
7	Space Bar

The bits can be masked with the following equates:

```
SHIFTBIT    equ    %00000001
CNTRLBIT    equ    %00000010
ALTERBIT    equ    %00000100
UPBIT       equ    %00001000
DOWNBIT     equ    %00010000
LEFTBIT     equ    %00100000
RIGHTBIT    equ    %01000000
SPACEBIT    equ    %10000000
```

- The support module for this call is CC3IO.

SS.ComSt

(Function code $28). Return serial port configuration information

Entry Conditions:

A = *path number*
B = $28

Exit Conditions:

Y = *high byte: parity*
low byte: baud rate
(See the Setstat call SS.ComSt for values)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The SCF manager uses this call when performing an SS.Opt Getstat on an SCF-type device. User calls to SS.ComSt do not update the path descriptor. Use the SS.OPT Getstat call for most applications, because it automatically updates the path descriptor.

SS.MnSel

(Function code $87). Requests that the high-level menu handler take control of menu selection

Entry Conditions:

A = *path number*
B = $87

Exit Conditions:

A = *menu ID* (if valid selection)
0 (if invalid selection)
B = *item number of menu* (if valid selection)

Error Output:

CC = carry set on error
B = *error code* (if invalid selection)

Additional Information:

- After detecting a valid mouse click (when the mouse is pointing to a control area on a window), a process needs to call SS.MnSel. This call tells the enhanced window interface to handle any menu selection being made. If accumulator A returns with 0, then no selection has been made. The calling process needs to test and handle other returned values.
- A condition where Register A returns a valid menu ID number and Register B returns 0 signals the selection of a menu with no items. The application can now take over and do a special graphics pull down of its own. The menu title remains highlighted until the application calls the SS.UMBar SetStat to update the menu bar.
- The support module for this call is WindInt.

SS.Mouse

(Function code $89). Gets mouse status

Entry Conditions:

A = *path number*
B = $89
X = *data storage area address*
Y = *mouse port select:*
0 = automatic selection
1 = right side
2 = left side

Exit Conditions:

X = *data storage area address*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.Mouse returns information on the current mouse and its fire button. The following list defines the 32-byte data packet that SS.Mouse creates:

Pt.Valid	rmb 1	Is returned info valid? (0 = no, 1 = yes)	
Pt.Actv	rmb 1	Active side (0 = off, 1 = right, 2 = left)	
Pt.ToTm	rmb 1	Timeout initial value	
Pt.TTTo	rmb 1	Time until timeout	
	rmb 2	RESERVED	
Pt.TSSt	rmb 2	Time since counter start	
Pt.CBSA	rmb 1	Current button state	(Button A)
Pt.CBSB	rmb 1	Current button state	(Button B)
Pt.CCtA	rmb 1	Click count	(Button A)
Pt.CCtB	rmb 1	Click count	(Button B)
Pt.TTSA	rmb 1	Time this state counter	(Button A)
Pt.TTSB	rmb 1	Time this state counter	(Button B)
Pt.TLSA	rmb 1	Time last state counter	(Button A)
Pt.TLSB	rmb 1	Time last state counter	(Button B)
	rmb 2	RESERVED	
Pt.BDX	rmb 2	Button down frozen Actual X	
Pt.BDY	rmb 2	Button down frozen Actual Y	
Pt.Stat	rmb 1	Window pointer type location	
Pt.Res	rmb 1	Resolution (0-640 by 0 = 10/1 = 1)	
Pt.AcX	rmb 2	Actual X value	
Pt.AcY	rmb 2	Actual Y value	
Pt.WRX	rmb 2	Window relative X	
Pt.WRY	rmb 2	Window relative Y	
Pt.Siz	equ .	Packet size 32 bytes	
SPt.SRX	rmb 2	System use, screen relative X	
SPt.SRY	rmb 2	System use, screen relative Y	
SPt.Siz	equ .	Size of packet for system use	

- Button Information:

  Pt.Valid. The valid byte gives the caller an indication of whether the information contained in the returned packet is accurate. The information returned by this call is only valid if the process is running on the current interactive window. If the process is on a non-interactive window, the byte is zero and the process can ignore the information returned.

Pt.Actv. This byte shows which port is selected for use by all mouse functions. The default value is Right (1). You can change this value with the SS.GIP Setstat call.

Pt.ToTm. This is the initial value used by Pt.TTTo.

Pt.TTTo. This is the count down value (as of the instant the Getstat call is made). This value starts at the value contained in the Pt.ToTm and counts down to zero at a rate of 60 counts per second. The system maintains all counters until this value reaches 0, at which point it sets all counters and states to 0. The mouse scan routine changes into a quiet mode which requires less overhead than when the mouse is active. The timeout begins when both buttons are in the up (open) state. The timer is reinitialized to the value in Pt.ToTm when either button goes down (closed).

Pt.TSSt. This counter is constantly increasing, beginning when either button is pressed while the mouse is in the quiet state. All counts are a number of ticks (60 per second). The timer counts to $FFFF, then stays at that value if additional ticks occur.

Pt.CBSA. These flag bytes indicate the state of the button
Pt.CBSB. at the last system clock tick. A value of 0 indicates that the button is up (open). A value other than zero indicates that the button is down (closed). Button A is available on all Tandy joysticks and mice. Button B is only available for products that have two buttons.

The system scans the mouse buttons each time it scans the keyboard.

Pt.CCtA. This is the number of clicks that have occurred
Pt.CCtB. since the mouse went into an active state. A click is defined as pressing (closing) the button, then releasing (opening) the button. The system counts the clicks as the button is released.

Pt.TTSA. This counter is the number of ticks that have
Pt.TTSB. occurred during the current state, as defined by Pt.CBSx. This counter starts at one (counts the tick when the state changes) and increases by one for each tick that occurs while the button remains in the same state (open or closed).

Pt.TLSA. This counter is the number of ticks that have
Pt.TLSB. occurred during the time that a button is in a state opposite of the current state. Using this count and the TTSA/TTSB count, you can determine how a button was in the previous state, even if the system returns the packet after the state has changed. Use these counters, along with the state and click count values, to define any type of click, drag, or hold convention you want.

Reserved. Two packet bytes are reserved for future expansion of the returned data.

- Position Information:

Pt.BDX. These values are copies of the Pt.AcX and Pt.AcY
Pt.BDY. values when either of the buttons change from an open state to a closed state.

Pt.Stat. This byte contains information about the area of the screen on which the mouse is positioned. Pt.Valid must be a value other than 0 for this information to be accurate. If Pt.Valid is 0, this value is also 0 and not accurate. Three types of areas are currently defined:

0 = content region or current working area of the window
1 = control region (for use by Multi-View)
2 = off window, or on an area of the screen that is not part of the window

Pt.Res. This value is the current resolution for the mouse. The mouse must always return coordinates in the range of 0-639 for the X axis and 0-191 for the Y axis. If the system is so configured, you can use the high-resolution mouse adapter which provides a 1 to 1 ratio with these values plus 1. If the adapter is not in use, the resolution is a ratio of 1 to 10 on the X axis and 1 to 3 on the Y axis. The keyboard mouse provides a resolution of 1 to 1. The values in Pt.Res are:

0 = low res (x:10, y:3)
1 = high res (x,y:1)

Pt.AcX. The values read from the mouse returned in the
Pt.AcY. resolution as described under Pt.Res.

Pt.WRX. The values read from the mouse minus the
Pt.WRY. starting coordinates of the current window's working area. These values return the coordinates in numbers relative to the type of screen. For example, the X axis is in the range 0-639 for high-resolution screens and 0-319 for low resolution screens. You can divide the window relative values by 8 to obtain absolute character positions. These values are most helpful when working in non-scaled modes.

- The support modules for this call are CC3IO, GrfInt, and WindInt.

SS.Palet

(Function code $91). Gets palette information

Entry Conditions:

A = *path number*
B = $91
X = *pointer to the 16-byte palette information buffer*

Exit Conditions:

X = *pointer to the 16-byte palette information buffer*

Additional Information:

- SS.Palet reads the contents of the 16 screen palette registers, and stores them in a 16-byte buffer. When you make the call, be sure the X register points to the desired buffer location. The pointer is retained on exit. The palette values returned are specific to the screen on which the call is made.
- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.ScTyp

(Function code $93). Returns the type of a screen to a calling program.

Entry Conditions:

A = *path*
B = $93

Exit Conditions:

A = *screen type code*
1 = 40 x 24 text screen
2 = 80 x 24 text screen
3 = not used
4 = not used
5 = 640 x 192, 2-color graphics screen
6 = 320 x 192, 4-color graphics screen
7 = 640 x 192, 4-color graphics screen
8 = 320 x 192, 16-color graphics screen

Additional Information:

- Support modules for this system call are GrfInt and WindInt.

SS.FBRgs

(Function code $96). Returns the foreground, background and border palette registers for a window.

Entry Conditions:

A = *path number*
B = $96

Exit Conditions:

A = *foreground palette register number*
B = *background palette register number* (if carry clear)
X = *least significant byte of border palette register number*

Error Output:

B = *error code if any*
CC = *carry set on error*

Additional Information:

- Support modules for SS.FBRgs are GrfInt and WindInt.

SS.DFPal

(Function code $97). Returns the default palette register settings.

Entry Conditions:

A = *path number*
B = $97
X = *pointer to 16-byte data space*

Exit Conditions:

X = *default palette data moved to user space*

Error Output:

B = *error code,* if any
CC = carry set on error

Additional Information:

- You can use SS.DFPal to find the values of the default palette registers that are used when a new screen is allocated by GrfInt or WindInt. The corresponding SetStat can alter the default registers. This GetStat/SetStat pair is for system configuration utilities and should not be used by general applications.

Set Status System Calls

Use the Set Status system calls with the RBF manager subroutine SETSTA. The OS-9 Level Two system reserves function Codes 7-127 for use by Microware. You can define Codes 200-255 and their parameter-passing conventions for your own use. (See the sections on device drivers in Chapters 4, 5, and 6.)

Following are the Set Status functions and their codes.

SS.OPT

(Function code $00). Writes the option section of the path descriptor

Entry Conditions:

A = *path number*
B = $00
X = *address of the status packet*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.OPT writes the option section of the path descriptor from the 32-byte status packet pointed to by Register X. Use this system call to set the device operating parameters, such as echo and line feed.

SS.SIZ

(Function code $02). Changes the size of a file for RBF-type devices

Entry Conditions:

A = *path number*
B = $02
X = *most significant 16 bits of the desired file size*
U = *least significant 16 bits of the desired file size*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.RESET

(Function code $03). Restores the disk drive head to Track 0 in preparation for formatting and error recovery (use only with RBF-type devices)

Entry Conditions:

A = *path number*
B = $03

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.WTRK

(Function code $04). Formats (writes) a track on a disk (RBF-type devices only)

Entry Conditions:

A = *path number*
B = $04
U = *track number* (least significant 8 bits)
X = *address of the track buffer*
Y = *side/density*
Bit B0 = side
0 = Side 0
1 = Side 1
Bit B1 = density
0 = single
1 = double

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- For hard disks or floppy disks that have a "format entire diskette command," SS.WTRK formats the entire disk only when the *track number* is zero.

SS.SQD

(Function code $0C). Starts the shutdown procedure for a hard disk that has sequence-down requirements prior to removal of power. (Use only with RBF-type devices.)

Entry Conditions:

A = *path number*
B = $0C

Exit Conditions: None

SS.KySns

(Function code $27). Turns the key sense function on and off

Entry Conditions:

A = *path number*
B = $27
X = *key sense switch value*
0 = normal key operation
1 = key sense operation

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- When SS.KySns switches the keyboard to key sense mode, the CC3IO module suspends transmission of keyboard characters to the SCF manager and the user. While the computer is in key sense mode, the only way to detect key press is with SS.KySns.
- The support module for this call is CC3IO.

SS.ComSt

(Function code $28). Used by the SCF manager to configure a serial port

Entry Conditions:

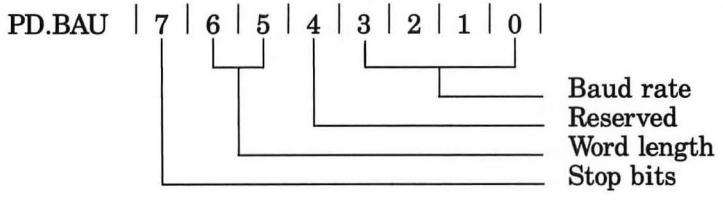
A = *path number*
B = $28
Y = *high byte: parity*
low byte: baud rate

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

Baud Configuration. The high order byte of Y determines the baud rate, the word length, and number of stop bits. The byte is configured as follows:

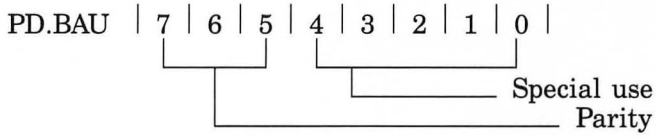


Stop bits:
0 = 1
1 = 2

Word length:
00 = 8 bit
01 = 7 bit

Baud rate:
0000 = 110
0001 = 300
0010 = 600
0011 = 1200
0100 = 2400
0101 = 4800
0110 = 9600
0111 = 19200
1xxx = undefined

- **Parity Configuration.** The low order byte of Y determines parity. The byte is configured as follows:



Parity:

xx0	= none
001	= odd (ACIAPAK and MODPAK only)
011	= even (ACIAPAK and MODPAK only)
101	= transmit: mark receive: ignore
111	= transmit: space receive: ignore

- The SCF manager uses SS.ComSt to inform a driver that serial port configuration information has been changed in the path descriptor. After calling SS.ComSt, a user routine must call the SS.OPT SetStat to correctly update the path descriptor.
- This call is for the use of the SCF manager. Use SS.OPT for changing baud, stop bit, and parity values.

SS.Close

(Function code $2A). Informs a device driver that a path is closed.

Additional Information:

This call is used internally by OS-9's SCF file manager and is not available for user programs. It can be used only by device drivers and file managers.

SS.AAGBf

(Function code $80). Reserves an additional graphics buffer

Entry Conditions:

A = *path number*
B = $80

Exit Conditions:

X = *buffer address*
Y = *buffer number (1-2)*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.AAGBf allocates an additional 8K graphics buffer. The first buffer (Buffer 0) must be allocated by using the DISPLAY GRAPHICS command. To use the DISPLAY GRAPHICS command, send control code $0F to the standard terminal driver. SS.AAGBf can allocate up to two additional buffers (Buffers 1 and 2), one at a time.
- After calling SS.AAGBf, Register X contains the address of the new buffer. Register Y contains the buffer number.
- To deallocate all graphics buffers, use the END GRAPHICS control code.
- When SS.AAGBf allocates a buffer, it also maps the buffer into the application's address space. Each buffer uses 8K of the available memory in the application's address space. Also, if SS.DStat is called, Buffer 0 is also mapped into the application's address space. Allocation of all three buffers reduces the application's free memory by 24K.
- The support module for this call is VDGINT.

SS.SLGBf

(Function code $81). Selects a graphics buffer

Entry Conditions:

A = *path number*
B = $81
X = $00 *select buffer for use*
$01-$FF *select buffer for use and display*
Y = *buffer number* (0-2)

Exit Conditions:

X = *unchanged from entry*
Y = *unchanged from entry*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use DISPLAY GRAPHICS to allocate the first graphics buffer. Use SS.AAGBf to allocate the second and third graphics buffers.
- Save each return address when writing directly to a screen. It is not necessary to save return addresses when using operating system graphics commands.
- SS.SLGBf does not update hardware information until the next vertical retrace (60Hz rate). Programs that use SS.AAGBf to change current draw buffers need to wait long enough to ensure that OS-9 has moved the current buffer to the screen.
- The screen shows the buffer only if the buffer is selected as the interactive device. If the device does not possess the keyboard, OS-9 stores the information until the device is selected as the interactive device. When the device is selected as the interactive device, the display shows the selected device's screen.
- The support module for this call is VDGINT.

SS.MpGPB

(Function code $84). Maps the Get/Put buffer into a user address space

Entry Conditions:

A = *path number*
B = $84
X = *high byte: buffer group number*
low byte: buffer number
Y = *action to take*
1 = map buffer
0 = unmap buffer

Exit Conditions:

X = *address of the mapped buffer*
Y = *number of bytes in buffer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support modules for this call are GrfInt and WindInt.
- SS.MpGPB maps a Get/Put buffer into the user address space. You can then save the buffer to disk or directly modify the pixel data contained in the buffer. Use extreme care when modifying the buffer so that you do not write outside of the buffer data area.

SS.WnSet

(Function code $86). Set up a high level window handler

Entry Conditions:

A = *path number*
B = $86
X = *window data pointer* (if Y = WT.FSWin or WT.Win)
Y = *window type code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The C language data structures for windowing are defined in the wind.h file in the DEFS directory of the system disk.
- The support module for this call is WindInt.

SS.SBar

(Function code $88). Puts a scroll block at a specified position

Entry Conditions:

A = *path number*
B = $88
X = *horizontal position of scroll block*
Y = *vertical position of scroll block*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- WT.FSWin-type windows have areas at the bottom and right sides to indicate their relative positions within a larger area. These areas are called scroll bars. SS.SBar gives an application the ability to maintain relative position markers within the scroll bars. The markers indicate

the location of the current screen within a larger screen. Calling SS.SBar, updates both scroll markers.

- The support module for this call is WindInt.

SS.Mouse

(Function code $89). Sets the sample rate and button timeout for a mouse

Entry Conditions:

A = *path number*
B = $89
X = *mouse sample rate and timeout*
most significant byte = *mouse sample rate*
least significant byte = *mouse timeout*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS. Mouse allows the application to define the mouse parameters. The sample rate is the number of clock ticks between the actual readings of the mouse status.
- The support module for the call is CC3IO.

SS.MsSig

(Function code $8A). Sends a signal to a process when the mouse button is pressed

Entry Conditions:

A = *path number*
B = $8A
X = *user defined signal code* (low byte only)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.MsSig sends the process a signal the next time a mouse button changes state (from open to closed). Once SS.MsSig sends the signal, the process must repeat the Setstat each time that it needs to set up the signal.
- Processes using SS.MsSig should have an intercept routine to trap the signal. By intercepting the signal, other processes can be notified when the change occurs. Therefore, the other processes do not need to continually poll the mouse.
- The SS.Relea Setstat clears the pending signal request, if desired. It also clears any pending signal from SS.SSig. Because of this, if you want to clear only one signal, you must reset the other signal after calling SS.MsSig.
- The support module for this call is CC3IO.

SS.AScrn

(Function code $8B). Allocates and maps a high-resolution screen into an application address space

Entry Conditions:

A = *path number*
B = $8B
X = *screen type*
0 = 640 x 192 x 2 colors (16K)
1 = 320 x 192 x 4 colors (16K)
2 = 160 x 192 x 16 colors (16K)
3 = 640 x 192 x 4 colors (32K)
4 = 320 x 192 x 16 colors (32K)

Exit Conditions:

X = *application address space of screen*
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.AScrn is particularly useful in systems with minimal memory when you want to allocate a high resolution graphics screen with all screen updating handled by a process.
- This call uses VDGInt (GRFINT is not required).
- All screens are allocated in multiples of 8K blocks. You can allocate a maximum of three buffers at one time. To select between buffers, use the SS.DScrn Setstat call.
- Screen memory is allocated but not cleared. The application using the screen must do this.
- Screens must be allocated from a VDG-type device—a standard 32-column text screen must be available for the device.
- The support module for this call is VDGINT.

SS.DScrn

(Function code $8C). Causes VDGINT to display a screen that was allocated by SS.AScrn

Entry Conditions:

A = *path number*
B = $8C
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.DScrn shows the requested screen if the requested screen is the current interactive device.
- The support module for this call is VDGINT.

SS.FScrn

(Function code $8D). Frees the memory of a screen allocated by SS.AScrn

Entry Conditions:

A = *path number*
B = $8D
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Do not attempt to free a screen that is currently on the display.
- SS.FScrn returns the screen memory to the system and removes it from an application's address space.
- The support module for this call is VDGINT.

SS.PScrn

(Function code $8E). Converts a screen to a different type

Entry Conditions:

A = *path number*
B = $8E
X = *new screen type*
    0 = 640 x 192 x 2 colors (16K)
    1 = 320 x 192 x 4 colors (16K)
    2 = 160 x 192 x 16 colors (16K)
    3 = 640 x 192 x 4 colors (32K)
    4 = 320 x 192 x 16 colors (32K)
Y = *screen number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.PScrn changes a screen allocated by SS.AScrn to a new screen type. You can change a 32K screen to either a 32K screen, or a 16K screen. You can change a 16K screen only to another 16K screen type. SS.PScrn updates the current display screen at the next clock interrupt.
- However, if you change a 32K screen to a 16K screen, OS-9 does not reclaim the extra 16K of memory. This means that you can later change the 16K screen back to a 32K screen.
- The support module for this call is VDGINT.

SS.Montr

(Function code $92). Sets the monitor type

Entry Conditions:

A = *path number*
B = $92
X = *monitor type*
0 = color composite
1 = analog RGB
2 = monochrome composite

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.Montr loads the hardware palette registers with the codes for the default color set for three types of monitors. The system default initializes the palette for a composite color monitor.
- The monochrome mode removes color information from the signals sent to a monitor.
- When a composite monitor is in use, a conversion table maps colors from RGB color numbers. In RGB and monochrome modes, the system uses the RGB color numbers directly.
- The support modules for this call are VDGINT and GrfDrv.

SS.GIP

(Function code $94). Sets the system wide mouse and key repeat parameters

Entry Conditions:

A = *path number*
B = $94
X = *mouse resolution;* in the most significant byte
    0 = low resolution mouse
    1 = optional high resolution adapter
  = *mouse port location;* in the least significant byte
    1 = right port
    2 = left port
Y = *key repeat start constant;* in the most significant byte
  = *key repeat delay;* in the least significant byte
    00XX = no repeat
    FFFF = unchanged

Error Output:

CC = carry set if error
B = *error code,* if any

Additional Information:

- Because this function affects system-wide settings, it is best to use it from system configuration utilities and not from general application program.
- The support module for this call is CC3IO.

SS.UMBAR

(Function code $95). Requests the high level menu manager to update the menu bar.

Entry Conditions:

A = *path number*
B = $95

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- An application can call SS.UMBar when it needs to redraw menu bar information, such as when it enables or disables menus, or when it completes a window *pull down* and needs to restore the menu.
- The support module for this call is WindInt.

SS.DFPal

(Function code $97). Sets the default palette register values

Entry Conditions:

A = *path number*
B = $97
X = *pointer to 16 bytes of palette data*

Exit Conditions:

X = unchanged, bytes moved to system defaults
CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use SS.DFPal to alter the system-wide palette register defaults. The system uses these defaults when it allocates a new screen using the DWSet command.
- Because this function affects system wide settings, it is best to use it from system configuration utilities, not general application programs.

SS.Tone

(Function code $98). Creates a sound through the terminal output device.

Entry Conditions:

A = *path number*
B = $98
X = *duration and amplitude of the tone*
    LSB = duration in ticks (60-sec) in the range 0-255
    MSB = amplitude of tone in the range 0-63
Y = *relative frequency counter* (0 = low, 4095 = high)

Exit Conditions:

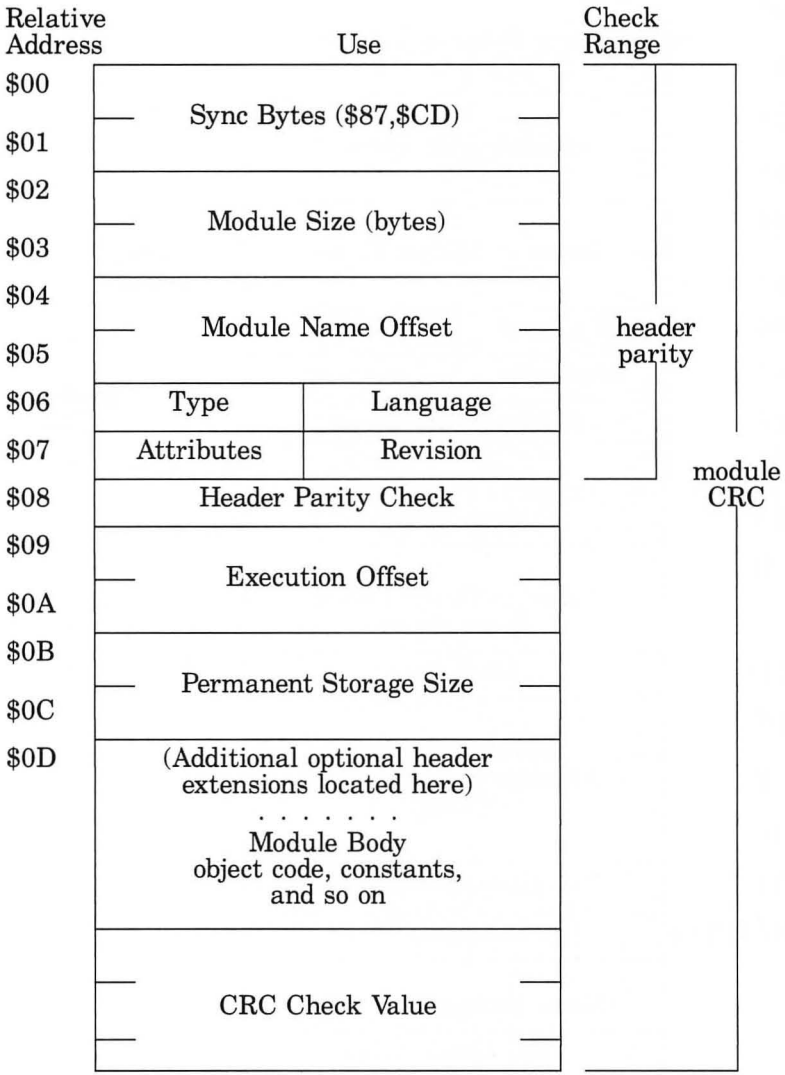
These are the same as the entry conditions. There are no error conditions.

Additional Information:

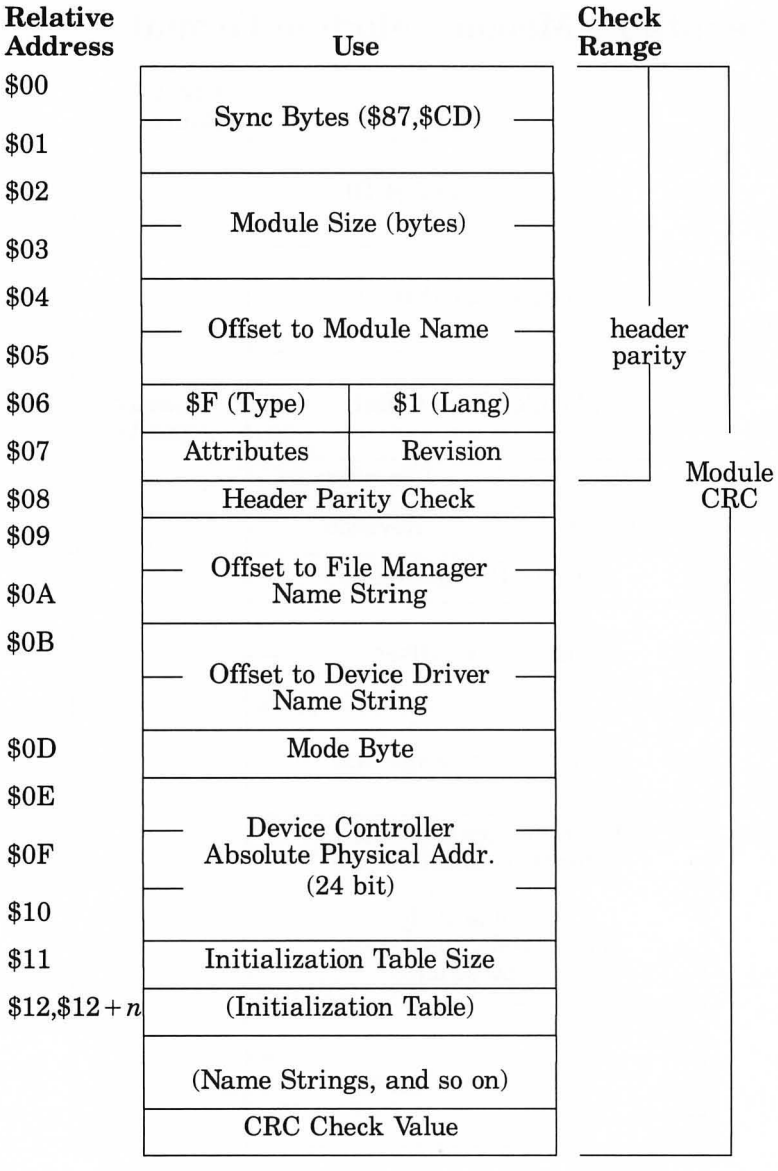
- This call produces a programmed IO tone through the speaker of the monitor used by the terminal device. You can make the call on any valid path open to term or to a window device.
- The system does not mask interrupts during the time the tone is being produced.
- The frequency of the tone is a relative number ranging from 0 for a low frequency to 4095 for a high frequency. The widest variation of tones occurs at the high range of the scale.

Memory Module Diagrams

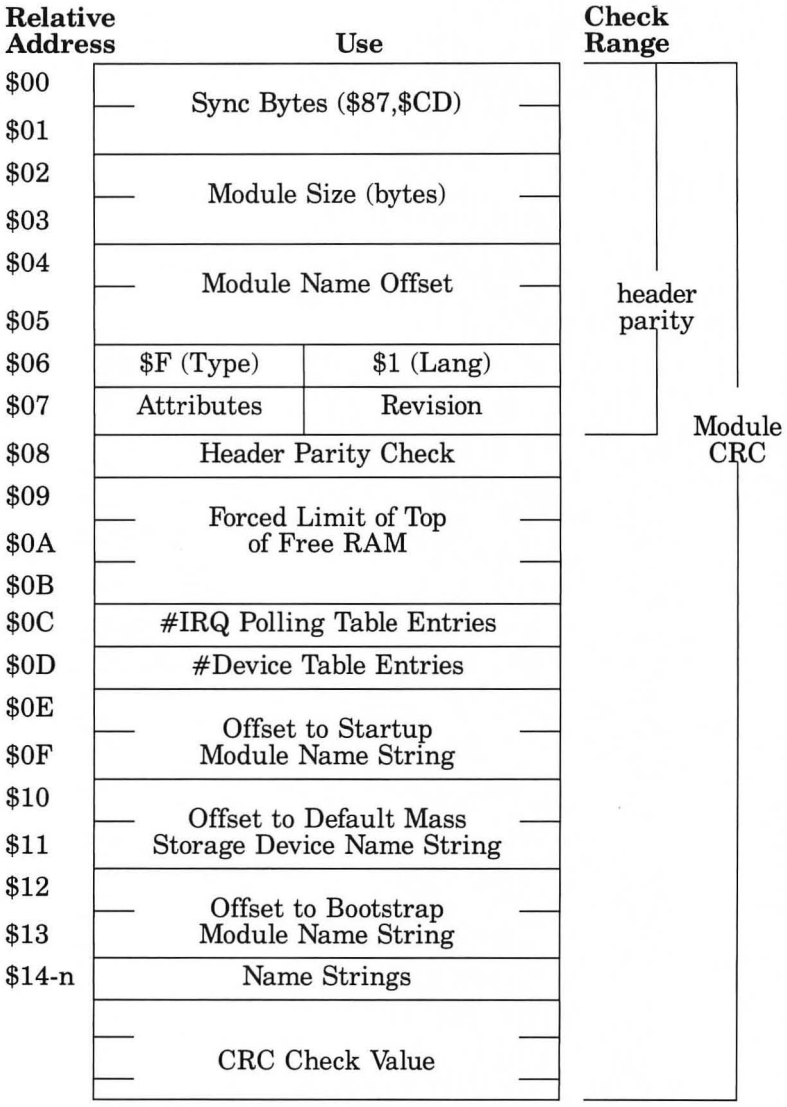
Executable Memory Module Format



Device Descriptor Format



INIT Module Format



Standard Floppy Disk Format

Color Computer 3

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	12	00
(repeated 18 times)	3	F5
	1	track number (0-34)
	1	side number (0-1)
	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)		

System Error Codes

The error codes are shown in both hexadecimal and decimal. The error codes listed include OS-9 system error codes, BASIC error codes, and standard windowing system error codes.

Code HEX	Code DEC	Code Meaning
$01	001	UNCONDITIONAL ABORT — An error occurred from which OS-9 cannot recover. All processes are terminated.
$02	002	KEYBOARD ABORT — You pressed [BREAK] to terminate the current operation.
$03	003	KEYBOARD INTERRUPT — You pressed [SHIFT][BREAK] either to cause the current operation to function as a background task with no video display or to cause the current task to terminate.
$B7	183	ILLEGAL WINDOW TYPE — You tried to define a text type window for graphics or used illegal parameters.
$B8	184	WINDOW ALREADY DEFINED — You tried to create a window that is already established.
$B9	185	FONT NOT FOUND — You tried to use a window font that does not exist.
$BA	186	STACK OVERFLOW — Your process (or processes) requires more stack space than is available on the system.
$BB	187	ILLEGAL ARGUMENT — You have used an argument with a command that is inappropriate.
$BD	189	ILLEGAL COORDINATES — You have given coordinates to a graphics command which are outside the screen boundaries.
$BE	190	INTERNAL INTEGRITY CHECK — System modules or data are changed and no longer reliable.
$BF	191	BUFFER SIZE IS TOO SMALL — The data you assigned to a buffer is larger than the buffer.

Code HEX	Code DEC	Code Meaning
$C0	192	ILLEGAL COMMAND — You have issued a command in a form unacceptable to OS-9.
$C1	193	SCREEN OR WINDOW TABLE IS FULL — You do not have enough room in the system window table to keep track of any more windows or screens.
$C2	194	BAD/UNDEFINED BUFFER NUMBER — You have specified an illegal or undefined buffer number.
$C3	195	ILLEGAL WINDOW DEFINITION — You have tried to give a window illegal parameters.
$C4	196	WINDOW UNDEFINED — You have tried to access a window that you have not yet defined.
$C8	200	PATH TABLE FULL — OS-9 cannot open the file, because the system path table is full.
$C9	201	ILLEGAL PATH NUMBER — The path number is too large, or you specified a non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL — Your system cannot handle an interrupt request, because the polling table does not have room for more entries.
$CB	203	ILLEGAL MODE — The specified device cannot perform the indicated input or output function.
$CC	204	DEVICE TABLE FULL — The device table does not have enough room for another device.
$CD	205	ILLEGAL MODULE HEADER — OS-9 cannot load the specified module because its sync code, header parity, or Cyclic Redundancy Code is incorrect.
$CE	206	MODULE DIRECTORY FULL — The module directory does not have enough room for another module entry.

Code HEX	Code DEC	Code Meaning
$CF	207	MEMORY FULL — Process address space is full or your computer does not have sufficient memory to perform the specified task.
$D0	208	ILLEGAL SERVICE REQUEST — The current program has issued a system call containing an illegal code number.
$D1	209	MODULE BUSY — Another process is already using a non-shareable module.
$D2	210	BOUNDARY ERROR — OS-9 has received a memory allocation or deallocation request that is not on a page boundary.
$D3	211	END OF FILE — A read operation has encountered an end-of-file character and has terminated.
$D4	212	RETURNING NON-ALLOCATED MEMORY — The current operation has attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT — The file structure of the specified device is damaged.
$D6	214	NO PERMISSION — The attributes of the specified file or device do not permit the requested access.
$D7	215	BAD PATH NAME — The specified pathlist contains a syntax error, for instance an illegal character.
$D8	216	PATH NAME NOT FOUND — The system cannot find the specified pathlist.
$D9	217	SEGMENT LIST FULL — The specified file is too fragmented for further expansion.
$DA	218	FILE ALREADY EXISTS — The specified filename already exists in the specified directory.
$DB	219	ILLEGAL BLOCK ADDRESS — The file structure of the specified device is damaged.

Code HEX	Code DEC	Code Meaning
$DC	220	PHONE HANGUP-DATA CARRIER DETECT LOST — The data carrier detect is lost on the RS-232 port.
$DD	221	MODULE NOT FOUND — The system received a request to link a module that is not in the specified directory.
$DF	223	SUICIDE ATTEMPT — The current operation has attempted to return to the memory location of the stack.
$E0	224	ILLEGAL PROCESS NUMBER — The specified process does not exist.
$E2	226	NO CHILDREN — The system has issued a *wait service* request but the current process has no dependent process to execute.
$E3	227	ILLEGAL SWI CODE — The system received a software interrupt code that is less than 1 or greater than 3.
$E4	228	PROCESS ABORTED — The system received a signal Code 2 to terminate the current process.
$E5	229	PROCESS TABLE FULL — A fork request cannot execute because the process table has no room for more entries.
$E6	230	ILLEGAL PARAMETER AREA — A fork call has passed incorrect high and low bounds.
$E7	231	KNOWN MODULE — The specified module is for internal use only.
$E8	232	INCORRECT MODULE CRC — The CRC for the module being accessed is bad.
$E9	233	SIGNAL ERROR — The receiving process has a previous, unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE — The system cannot locate the specified module.

Code HEX	Code DEC	Code Meaning
$EB	235	BAD NAME — The specified device, file, or module name is illegal.
$EC	236	BAD MODULE HEADER — The specified module header parity is incorrect.
$ED	237	RAM FULL — No free system random access memory is available: the system address space is full, or there is no physical memory available when requested by the operating system in the system state.
$EE	238	UNKNOWN PROCESS ID — The specified process ID number is incorrect.
$EF	239	NO TASK NUMBER AVAILABLE — All available task numbers are in use.

Device Driver Errors

I/O device drivers generate the following error codes. In most cases, the codes are hardware-dependent. Consult your device manual for more details.

Code HEX	Code DEC	Code Meaning
$F0	240	UNIT ERROR — The specified device unit doesn't exist.
$F1	241	SECTOR ERROR — The specified sector number is out of range.
$F2	242	WRITE PROTECT — The specified device is write-protected.
$F3	243	CRC ERROR — A Cyclic Redundancy Code error occurred on a read or write verify.
$F4	244	READ ERROR — A data transfer error occurred during a disk read operation, or there is a SCF (terminal) input buffer overrun.

Code HEX	Code DEC	Code Meaning
$F5	245	WRITE ERROR — An error occurred during a write operation.
$F6	246	NOT READY — The device specified has a *not ready* status.
$F7	247	SEEK ERROR — The system attempted a seek operation on a non-existent sector.
$F8	248	MEDIA FULL — The specified media has insufficient free space for the operation.
$F9	249	WRONG TYPE — An attempt is made to read incompatible media (for instance an attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY — A non-shareable device is in use.
$FB	251	DISK ID CHANGE — You changed diskettes when one or more files are open.
$FC	252	RECORD IS LOCKED-OUT — Another process is accessing the requested record.
$FD	253	NON-SHAREABLE FILE BUSY — Another process is accessing the requested file.
$FE	254	I/O DEADLOCK ERROR — Two processes have attempted to gain control of the same disk area at the same time.

Index

ACIAPAK 8-135
active process 2-12 - 2-13
    queue 2-14, 8-98
    state 2-13 - 2-14
address
    find 64K block 8-85
    lines 2-7
    polling 2-17
    space, add module 8-104
age, process 2-14
alarm, set 8-66
allocate
    high RAM 8-69
    image 8-70
    memory 8-76
    memory blocks 8-67 - 8-68
    process descriptor 8-71
    process task number 8-73
    RAM 8-72
allocation
    bit map 8-7
    map sector 5-1
    of memory 2-5 - 2-7
    polling 2-17
allocation map

    clear 8-13
    disk 5-3
alpha screen
    cursor 8-118
    memory 8-117
ASM assembler 8-2
assembler, RMA 8-2
attach a device 8-44 - 8-45
attribute
    byte 5-5,
    file 5-12

background color, get 8-129
bell, set alarm 8-66
bit map 2-5
    allocation 8-7
bit map (cont'd.)
    search memory allocation 8-33
block
    allocate system memory 8-105
    deallocate system memory 8-106
    map into process space 8-96
    number 2-7
    scroll 8-139
block map, system 8-18
boot
    file, load 5-26
    module, link 8-75
booting OS-9 1-3
bootstrap
    memory request 8-76
    system 8-75
border color, get 8-129
buffer
    map (Get/Put) 8-138
    reserve graphics 8-136
button
    state, mouse 8-124 - 8-125, 8-126
    timeout, mouse 8-140
byte
    attribute 5-5
    deallocate 64-byte block 8-101
    get from memory block 8-94
    get two bytes 8-95
    read from path 8-59 - 8-60
    store in task 8-109

calling process
    insert in I/O queue 8-91
    terminate 8-14
    turn off 8-35, 8-43
CC3DISK 1-2

CC3GO module 2-19
CC3IO 1-2, 6-1
chain 8-8 - 8-9
change
    device operating parameters 5-23
    directory 8-46
character
    read SCF input 6-13
    write, SCF 6-14
ChgDir 4-4
child process 2-13
    create 8-15 - 8-17
clear specified block 8-77
click 8-126
CLOCK 1-2
clock
    module 1-2, 2-19
    real-time 2-12, 2-17
close
    file 4-7
    path 8-47, 8-135
codes
    signal 2-15
    system error C-1
command interpreter 1-4
communication, interprocess 2-15
compact module directory 8-88
compare strings 8-10
compatibility with Level One 2-1
concurrent execution 7-1 - 7-3
copy external memory 8-11
count, link 2-5
counter start, mouse 8-124
CPU 2-7
    time 4-11
CRC
    calculate 8-12
    validate module 8-111
    value 3-1 - 3-3
create
    directory 8-55 - 8-56
create (cont'd.)
    file 8-48 - 8-49
current
    data directory 8-51
    execution directory 8-51
cursor positioning 4-5
cyclic redundancy check 3-1 - 3-3

DAT
    hardware 8-99
    registers 8-103
    to logical address 8-78
data
    available, SCF test 8-113
    directory 8-51
    stream 4-3
    transfer, pipes 7-1 - 7-3
    move in memory 8-97
DAT image 8-70
    conversion 8-78
    copy into process descriptor 8-102
    deallocate block 8-77
    high block 8-86
    low block 8-87
    pointer 8-95
DAT task number
    release 8-99
    reserve 8-100
date
    get system 8-40
    set 8-38
deadlock 5-13
deadly embrace 5-13
deallocate
    image RAM blocks 8-79
    map bits 8-13
    process descriptor 8-80
    RAM blocks 8-81
    task number 8-82
default palette registers 8-129, 8-149
delete file 8-50 - 8-51

descriptions, system call 8-2
descriptor
    get process 8-20
    path 4-18
    pointer 8-82
    process 2-13
detach device 8-52
device
    add or remove from polling table 8-92
    attach 8-44 - 8-45
    attachment, verify 8-44 - 8-45
    controller 5-15
    control registers, initialize 6-12
    control registers, SCF 6-12
    descriptor 1-4, 4-2, 4-17, A-2
    detach 8-52
    modules 5-15
    modules, RBF 5-8 - 5-10
    modules, SCF 6-6 - 6-8
    name, get 8-115
    open path to 8-57 - 8-58
    operating parameters, RBF 5-23
    operating parameters, SCF 6-15
    status 2-17 - 2-18, 8-63
    status, get 8-54
    table 4-2, 8-52
    terminate, RBF 5-24
    terminate, SCF 6-16
    write to 8-64 - 8-65
device driver 1-3, 4-11
    close path 8-135
    modules 4-8
    name 5-15
    SCF 6-9 - 6-17
    SCF subroutines 6-10 - 6-17
    subroutines, RBF 5-16 - 5-27
device driver modules, RBF 5-13 - 5-17
device interrupt 5-25
    SCF 6-17
directory
    attribute byte 5-5
    change 8-46
    disk 5-5
    entry, module 8-83
    get module 8-19
    make 8-55 - 8-56
    module 2-12, 8-88
disk
    directories 5-5
    sector read 5-19, 5-21
disk allocation map 5-3
    sector 5-1
diskette format B-1
display
    screen 8-143
    status, get 8-115
drag 8-126
drive head, restore 8-131
duplicate path 8-53

editing, line 6-1, 8-61
end-of-file, test for 8-114
equate file 2-4
equivalent logical address 8-78
error
    codes, system C-1 - C-6
    message, write 8-30
    print 8-30
exclamation point, pipes 7-1 - 7-3
execute
    mode 5-11
    system calls 8-1 - 8-2
execution
    directory 8-51
    offset, module 3-7
exit calling process 8-14
external memory, read 8-11

fatal signal 2-13
file
    attribute byte 5-12
    closing 4-7
    create 4-4, 5-12, 8-48 - 8-49
    deadlock 5-13
    delete 4-5, 8-50 - 8-51
    descriptor 5-3 - 5-4
    execute mode 5-11
    get pointer position 8-114
    line reading/writing 4-6
    load module 8-29
    locking 5-12
    non-shareable 5-12
    opening 4-4
    open path 8-57 - 8-58
    permission bits 5-4
    pipe 7-1 - 7-3
    pointer 4-5, 8-62
    position, RBF 8-114
    read 5-1, 4-5
    sharing 5-12
    size, get 8-114
    status, get 8-54, 8-114
    update mode 5-11
    write line to 8-64 - 8-65
    writing 4-6
file manager 1-3
    modules 4-3
    name 5-15
find
    64-byte block 8-85
    module directory entry 8-84
fire button 8-123 - 8-127
FIRQ 4-12
    interrupt 2-17
flag, RAM In Use 8-81
flip byte 2-17
floppy diskette format B-1
foreground color, get 8-129
FORK 2-8
fork, child process 8-15 - 8-17
FORMAT 5-2
format
    device descriptor 4-17, A-2
    INIT module A-3
    memory module 3-6 - 3-7, A-1
    of device driver modules 4-10
    track 8-132
function
    calls 2-4 - 2-5, 8-1
    key sense 8-133

get
    a byte 8-94
    free high block 8-86
    free low block 8-87
    ID 8-22
    process pointer 8-89
    status 8-54
    Status system calls 8-112 - 8-130
    system time 8-40
Get/Put buffer, map 8-138
GETSTA 8-112
    SCF 6-15
GetStat 4-6
Getstats 5-23
graphics buffer
    reserve 8-136
    select 8-137
graphics interface 1-2
GRFINT 1-2

handler routine, virtual interrupt 8-110
hard disk shutdown 8-133
hardware
    controller, SCF 6-9
    DAT registers 8-103
    vector 2-16
header
    module 3-1 - 3-2
    parity 8-111

header (cont'd.)
pattern, floppy diskette B-1
high block, memory search 8-86
high-level
menu handler 8-122
menu manager 8-148
window handler 8-139
high-resolution
mouse adapter 8-126
screen, allocate 8-142
hold, button 8-126
I/O
calls 2-4 - 2-5, 8-1
device accessing 2-11
module, delete 8-90
path, close 8-47
queue, insert calling process 8-91
I/O system 1-3 - 1-4
calls 2-1, 8-2
system modules 1-1 - 1-4, 4-1
transfers 4-8
ID
return caller's process 8-22
set user 8-39
identification sector 5-1
image, allocate 8-70
INIT 1-2, 5-18
INIT module 2-17
format A-3
link 8-75
Init, SFC 6-12
initialization table, SCF device 6-6 - 6-8
initialize device memory 5-18
input buffer, read SCF character 6-13
insert process 8-74
install virtual interrupt 8-110
intercept, set signal 8-21
interface
graphics 1-2
VDG 4-2
Windint 4-2
interprocess communication 2-15
interrupt
device 5-25
enable, SCF 6-12
FIRQ 2-17
processing 2-1
IOMAN 1-2
IRQ 4-12
add/remove device from polling table 8-92
interrupt 2-17
polling 2-17
polling table 2-18
service routine 5-25
IRQSVC routine 4-13
IRQSV 4-11

joystick value, get 8-116

kernel 1-2
key
repeat parameters, set 8-147
sense function 8-133
status, get 8-120
keyboard scan 2-17

language byte 3-4
line
editing 6-1, 8-61
reads 4-6, 8-61
writes 4-6, 8-65
link
to memory module 8-23 - 8-24, 8-28
using module directory entry 8-83
link count 2-5
decrease 8-42

load
boot file 5-26
byte from memory block 8-94
from task offset 8-93
module 8-25 - 8-26, 8-29
two bytes 8-95
lock, end-of-lock 5-12
locking
files 5-12
record 5-10 - 5-11
logical
address space 2-6, 2-8
sector number 5-1
LSN 5-2, 5-5

macro 2-4
MAKDIR 4-4
make directory 8-55 - 8-56
manager
file 1-3
random block 1-3
sequential file 1-3
map
block 8-96
search allocation 8-33
mask byte 2-18
memory
allocate 8-76
allocate blocks 8-67 - 8-68
allocate high RAM 8-69
change process data size 8-27
deallocate 2-5
find low block 8-87
free screen 8-144
map 2-6
module format 3-6 - 3-7, A-1
module, link 8-23 - 8-24
move data 8-97
page 2-5
pool 8-80
request, bootstrap 8-76
memory (cont'd.)
segment 2-8
memory allocation 2-5 - 2-7
memory block 2-7
find 64K 8-85
get byte 8-94
get high 8-86
map 8-81
map, search 8-72
memory management 2-1, 2-5 - 2-12
unit 2-7 - 2-8
menu
manager, update request 8-148
selection 8-122
message, write error 8-30
MMU registers 2-8
mnemonic name, LSN 5-2
MODPAK 8-135
module
add into address space 8-104
body 3-1 - 3-2
clock 2-19
CRC calculate 8-12
decrease link count 8-42
delete I/O module 8-90
device descriptor 5-15
device driver 4-8
file manager 4-3
finding 2-12
format 3-1 - 3-3
link 8-28
link count, decrease 8-42
linking 1-2
load 8-25 - 8-26, 8-29
load and execute primary 8-8 - 8-9
name 3-3
RBF-type device drivers 5-13 - 5-17
SCF device descriptor 6-6 - 6-8

module (cont'd.)
types 3-1, 3-5
unlink 8-41
validate 8-111
module directory 2-5, 2-12
compact 8-88
entry, link using 8-83
find 8-84
get 8-19
pointer 8-84
module header 3-1 - 3-3, 5-15
SCF device driver 6-9
monitor, set type 8-146
mouse
button state 8-125
button timeout 8-140
click 8-122
coordinates 8-127
countdown 8-125
countup 8-125
parameters, set 8-147
port 8-125
resolution 8-126
screen position 8-126
send signal to process 8-141
status, get 8-123
timeout 8-124
window working area 8-127
move data 8-97
multiplexer 2-8
multiprogramming 2-12 - 2-16
management 2-1
multitasking 1-2

name parse 8-31 - 8-32
names, compare 8-10
next process 8-98
NMI interrupt 2-17
non-shareable file 5-12
number, path 8-53

open
file 8-48 - 8-49
path 8-57 - 8-58
operation of memory management 2-8 - 2-12
OS-9
Level One compatibility 2-1
modules 1-2
scheduler 2-14 - 2-15
OS9P3 2-1
module 2-2

packet size 8-124
palette, get information 8-127
palette register 8-129
set default 8-149
settings 8-129
parameters, mouse and key repeat 8-147
parent
directory 5-3
process 2-13
parity 8-135
parse name 8-31 - 8-32
path
close 8-47, 8-135
duplicate 8-53
open 8-57 - 8-58
read bytes 8-59 - 8-60
table 4-2
path descriptor 4-18, 5-5 - 5-8
read option section 8-112
SCF 6-2 - 6-6
write option section 8-130
permanent storage size, module 3-7
physical address space 2-7
pipe file manager 4-3
PIPEMAN 1-2 - 1-3, 4-3
pipes 4-3, 7-1 - 7-3

process descriptor 2-13 - 2-14, 8-102
deallocate 8-80
descriptor, allocate 8-71
get 8-20
pointer 8-82
processes
active 2-12
data size, change 8-27
process ID 2-13
return caller's 8-22
pseudo vector 2-16
PutStat 4-6

RAM 2-5 - 2-7
allocate 8-69, 8-72
allocate blocks 8-70
allocation 2-13
blocks, deallocate 8-81
blocks, deallocate image 8-79
interrupt vector 2-18
random
access 5-1
block file manager 1-3, 4-3
RBF
change file size 8-131
format track 8-132
get file size 8-114
manager 4-3
tables 5-14 - 5-17
read
bytes 8-59 - 8-60
device operating parameters 5-23
disk sector 5-19
external memory 8-11
input character, SCF 6-13
line 6-2, 8-61
mode 5-11
system call 6-1
real-time clock 2-12, 2-17
record locking 5-10
reference
System Mode calls 8-5 - 8-6
User Mode system calls 8-3 - 8-4
registers
DAT 8-103
MMU 2-8
release a task 8-99
request system memory 8-105
reserved memory 2-5 - 2-7
reserve task number 8-100
return
64 bytes 8-101
system memory 8-106
RMA assembler 8-2
ROOT directory 5-3, 5-5
RTS instruction 2-18

SCF
configure serial port 8-134 - 8-135
data available test 8-113
device control registers 6-12
Getsta 6-15
manager 4-3
path descriptor 6-2 - 6-6
terminate device 6-16
scheduler, OS-9 2-14 - 2-15
screen
allocate high-resolution 8-142
convert type 8-145
display 8-143
free memory 8-144
mouse position 8-126
palette 8-127
size, get 8-119
type 8-128, 8-142, 8-145
scroll block, install 8-139
search bits 8-33

sector 5-3
    pattern, floppy diskette B-1
seek, file pointer 8-62
segment, memory 2-8
select graphics buffer 8-137
send signal 8-34
sequential character
    file manager 1-3, 4-3
    I/O 6-1
serial port configuration 8-121
service
    request processing 2-1
    routine, IRQ 5-25
set
    alarm 8-66
    date 8-38
    IRQ 8-92
    priority 8-36
    process DAT image 8-102
    process task DAT registers 8-103
    status 8-63
    SVC 8-107 - 8-108
    SWI 8-37
    time 8-38
    user ID 8-39
Setstats 5-23
Set Status system calls 8-130 - 8-150
shareable bit 3-5
sharing, file 5-12
shell 1-4
shutdown hard disk 8-133
signal 2-15 - 2-16
    codes 2-15
    fatal 2-13
    from mouse to process 8-141
    intercept trap 2-15 - 2-16
    intercept, set 8-21
    send to process 8-34
single-user
    attribute 5-12
    bit, files 5-12
size
    of screen 8-119
    of window 8-119
sleep
    calling process 8-35
sleeping process 2-14, 2-16
slices, time 2-12
sound, create 8-150
speaker, create sound 8-150
state
    active 2-13
    of button 8-126
    sleeping 2-14
    suspend 4-13
    waiting 2-13
static storage address 2-18
status
    display 8-115
    get, SCF 6-15
    get mouse 8-123 - 8-127
    of key 8-120
    register 2-17
    set, SCF 6-15
status, get 8-54
status, set 8-63
store byte in a task 8-109
string, scan input 8-31 - 8-32
strings, compare 8-10
subroutines
    RBF device driver 5-16 - 5-27
    SCF device drivers 6-10 - 6-17
suspend
    bit 4-13 - 4-14
    state 4-13
SWI, set 8-37
SWI2 instruction 2-4
symbolic names 2-4
sync byte 3-3
synonymous path number, return 8-53

system
block map, get 8-18
boot 1-3
bootstrap 8-75
date, get 8-40
device, attach 8-44
error codes C-1 - C-6
initialization 2-1
link 8-104
mode call reference 8-5 - 8-6
time, get 8-40
system call
add 8-107 - 8-108
descriptions 8-2, 2-4
execution 8-1 - 8-2
get status 8-112 - 8-130
mnemonics names 8-1
User Mode reference 8-3 - 8-4
system memory
allocate high RAM 8-69
block map 8-81
deallocate 8-106
module directory, get 8-19
request 8-105
system modules 1-1 - 1-4

table
device 8-52
IRQ polling 2-18
RBF 5-14 - 5-17
SCF device descriptor 6-6 - 6-8
VIRQ 2-20
task
map 2-12
offset, load from 8-93
register 2-8
release 8-99
store byte 8-109
task number 8-73
DAT 8-100
deallocate 8-82
terminal, create sound 8-150
terminate
a device 5-24
calling process 8-14
SCF device 6-16
ticks 4-11
time
CPU 4-11
get system 8-40
set 8-38
sharing 2-11
slice 2-16, 2-12
timeout, mouse 8-124
track
format 8-132
restore drive head 8-131
trailer pattern, floppy diskette B-1
trap, signal intercept 2-15 - 2-16
type
convert screen 8-145
of screen 8-128
set monitor 8-146
window screen 8-142

unlink module 8-41 - 8-42
update mode 5-11
user calls 2-5
user ID 2-13
set 8-39
User Mode system calls reference 8-3 - 8-4

validate module 8-111
VDG 1-2
alpha screen cursor 8-118
alpha screen memory 8-117
interface 4-2
vector
pseudo 2-16
set SWI 8-37
vectoring 2-16

verify device attachment 8-44 - 8-45
video display generator 1-2
VIRQ 2-19 - 2-20
    polling table 2-19 - 2-20
virtual interrupt, install 8-110

wait
    calling process 8-43
    state 2-13 - 2-14
waiting process 2-13
wildcard 4-6
WINDINT 1-2
Windint interface 4-2
window
    descriptors 1-2
    high-level handler 8-139
    pointer location 8-124
    screen, type 8-142
    size, get 8-119
    type 8-145
    working area, mouse 8-127
working directory, change 8-46
write
    character to SCF output 6-14
    disk sector 5-21
    path descriptor 8-130 - 8-131
    to file or device 8-64
write line 8-65
    line system call 6-2

OS-9
Windowing
System

Contents

Chapter 1 Types of OS-9 Windows 1-1
Device Windows 1-1
Overlay Windows 1-2
Opening a Device Window 1-2
Opening an Overlay Window 1-4

Chapter 2 Overview of Commands and Parameters 2-1
Parameters 2-1

Chapter 3 General Commands 3-1
Background Color 3-2
Bold Switch 3-3
Border Color 3-4
Change Working Area 3-5
Default Color 3-6
Define GET/PUT Buffer 3-7
Device Window End 3-10
Device Window Protect Switch 3-11
Device Window Set 3-12
Foreground Color 3-14
Select Font 3-15
Graphics Cursor Set 3-17
Get Block 3-18
GET/PUT Buffer Load 3-19
Kill GET/PUT Buffer 3-20
Logic Set 3-21
Overlay Window End 3-23
Overlay Window Set 3-24
Change Palette 3-26
Proportional Switch 3-28
Pattern Set 3-29
Put Block 3-32
Scale Switch 3-33
Window Select 3-34
Transparent Character Switch 3-35

Chapter 4 Drawing Commands 4-1
Draw Arc 4-2
Draw Bar 4-3
Relative Draw Bar 4-3

Draw Box 4-4
Relative Draw Box 4-4
Draw Circle 4-5
Draw Ellipse 4-6
Flood Fill 4-7
Draw Line 4-8
Relative Draw Line 4-8
Draw Line and Move 4-9
Relative Draw Line and Move 4-9
Draw Point 4-10
Relative Draw Point 4-10
Put Graphics Cursor 4-11
Set Draw Pointer 4-12
Relative Set Draw Pointer 4-12

Chapter 5 Text Commands 5-1

Index

Types of OS-9 Windows

Unlike many operating systems, OS-9 has a built-in windows program. This driver, the Windowing System, lets you lay one or more smaller screen displays, called *windows*, on your screen display.

With these windows, you can perform several tasks at the same time. For example, suppose you are writing a business letter using a word processor in one window. You can go to a spreadsheet program in another window, get a price quote you need, return to the word processor, and include the price in the letter.

The Windowing System allows as many windows as your computer's memory can support, with a maximum of 32 at one time.

In OS-9, there are two types of windows: device and overlay.

Device Windows

A *device window* is one that can run a program or utility. This is the type of window you would use in the word processor/spreadsheet example given above. Each device window acts as an individual terminal.

The device windows are designated as devices **/w1 - /w7**. You open a device window as you do any other OS-9 device. You tell OS-9 the window's parameters—including whether the window is for text or graphics. If you want to run a process in the window, you can start an execution environment, such as a *shell*, on the window. (See "Opening a Device Window," later in this chapter, and the DWSet command in Chapter 3.)

> **Note:** If you want only to send output to the device window—without running a process in the window—do not start a shell on the window.

Device windows cannot overlay each other, and their boundaries cannot overlap.

Overlay Windows

An *overlay window* is a window that you open on top of a device window. (You can place overlay windows over other overlay windows, but there must always be a device window at the bottom of the stack.) The purpose of overlay windows is to display computer dialog. You cannot fork a shell to an overlay window; however, you can **run** a shell in an overlay window. Overlay windows assume the screen type of the device windows they overlay.

Opening a Device Window

To open a device window, follow these steps:

1. If you want to allocate memory for the window, use OS-9's **iniz** command. Type:

   ```
   iniz /wnumber [ENTER]
   ```

   where *number* is the number of the device window you wish to open (1-7). If you do not specify *number*, OS-9 uses the next available device window number.

   If you do not use the **iniz** command, memory is allocated dynamically (as needed) to the window.

2. Next, you send an escape sequence to OS-9 that tells it the window's parameters. These parameters include the screen type, size, and colors. For example:

   ```
   wcreate /w -s=2 20 10 40 10 01 00 00 [ENTER]
   ```

   or

   ```
   display 1b 20 02 14 0a 28 0a 01 00 00 /w
   [ENTER]
   ```

   sends the escape sequence for the next available window to the DWSet command. The **wcreate** command lets you use decimal numbers, while the **display** command requires hexadecimal numbers.

If you wish to send an escape sequence to a specific window, route the command to that device. For example:

```
wcreate /w2 -s=2 20 10 40 10 01 00 00 ENTER
```

sends the escape sequence to device **/w2**. The functions of the codes, as used in the **wcreate** command, are as follows:

2	sets a screen type of 80 x 24 (text only).
20	starts the window at character/column 20.
10	starts the window at line/row 10.
40	sets a window size of 40 characters.
10	sets a window size of 10 lines.
01	sets the foreground color to blue.
00	sets the background color to white.
00	sets the border to white.

If you do not send escape sequences, OS-9 uses default descriptors for the windows. The defaults are:

Window Number	Screen Type (chars./line)	Starting Position (horiz., vert.)	Size (columns, rows)
1	40 (text)	0,0	27,11
2	40 (text)	28,0	12,11
3	40 (text)	0,12	40,12
4	80 (text)	0,0	60,11
5	80 (text)	60,0	19,11
6	80 (text)	80,0	80,12
7	80 (text)	0,0	80,24

3. Use OS-9's **shell** command to fork a shell to the window. Type:

```
shell i=/wnumber & ENTER
```

where *number* is the number used in the **iniz** or **wcreate** command. The **i=** parameter creates an *immortal* shell. Creating an immortal shell protects the window and its shell from being destroyed if you accidentally exit the shell using CTRL BREAK. If you omit the **i=** parameter, the shell is forked to the next available device window.

You now have a window that can run its own tasks. Information displayed in that window is automatically scaled to the window's size.

Opening an Overlay Window

To open an overlay window, use the Overlay Window Set function. (See OWSet in Chapter 3, "General Commands.")

Overview of Commands and Parameters

The windowing commands are divided among three chapters, based on their functions.

Chapter 3 describes the *general commands*. These commands let you create windows and buffers, access buffers, set switches, and maintain the window environment.

Chapter 4 describes the *drawing commands*. Besides letting you draw all kinds of images (circles, ellipses, arcs, and boxes, to name a few), these commands also enable you to color areas or to fill them with patterns.

Chapter 5 describes the *text commands*. Use these commands to manipulate the text cursor and the text attributes. Text commands operate on hardware text screens (Screen Types 1 and 2) and graphics windows if a font is selected.

Each command description lists the command's name, code, and parameters. To call a Windowing System command using OS-9's **display** command, type `display`, followed by the command code and the values you want to supply for the parameters.

Parameters

The following is a complete list of the parameter abbreviations used in Chapters 3, 4, and 5. All parameters represent a single byte of information.

Parameter	Description
HBX	*high order byte of x value*
LBX	*low order byte of x value*
HBY	*high order byte of y value*
LBX	*low order byte of y value*
HBXo	*high order byte of x-offset value (relative)*
LBXo	*low order byte of x-offset value (relative)*
HBYo	*high order byte of y-offset value (relative)*
LBYo	*low order byte of y-offset value (relative)*
HBR	*high order byte of radius*
LBR	*low order byte of radius*

Parameter	Description
HBL	*high order byte of length*
LBL	*low order byte of length*
HSX	*high order byte of size in x direction*
LSX	*low order byte of size in x direction*
HSY	*high order byte of size in y direction*
LSY	*low order byte of size in y direction*
HBRx	*high order byte of radius in x direction*
LBRx	*low order byte of radius in x direction*
GRP	*GET/PUT buffer group number* (1-254)
BFN	*GET/PUT buffer number* (1-255)
LCN	*logic code number*
PRN	*palette register number* (0-15, wraps mod 15)
CTN	*color table number* (0-63, wraps mod 64)
FNM	*font number*
CPX	*character position x* (0-xmax)
CPY	*character position y* (0-ymax)
STY	*screen type*
SVS	*save switch* (0 = nosave, 1 = save area under overlay)
SZX	*size in x (columns)*
SZY	*size in y (rows)*
XDR	*dimension ratio x used with YDR as YDR/XDR*
YDR	*dimension ratio y*
BSW	*binary switch* (0 = off, 1 = on)

Chapter 3

General Commands

The general commands let you set up and customize windows. They also let you set up and access image buffers and select colors for the screen.

BColor Background Color

Function: Lets you choose a color palette register to use as the background color. See the Palette command for setting up the actual colors.

Code: 1B 33

Parameters: PRN

BoldSw Bold Switch

Function: Enables or disables boldfacing for text on graphics screens. If boldface is on, the screen displays subsequent characters in bold. If boldface is off, the screen displays subsequent characters in the regular font.

Code: 1B 3D

Parameters: BSW

BSW = switch
00 = off (Default)
01 = on

Notes:

- You can use BoldSw with any font.
- Boldface is not supported on hardware text screens (Screen Types 1 and 2).

Border Border Color

Function: Lets you change the palette register used for the screen border. See the Palette command for setting up the actual colors.

Code: 1B 34

Parameters: PRN

Notes:

- You set the border by selecting a palette register to use for the border register. When the actual color is changed in the palette register selected by the command, the color of the screen border changes to the new color. In general, the border register usually matches the background palette register.

CWArea Change Working Area

Function: Lets you alter the working area of the window. Normally, the system uses this call for high-level windowing, but you can use it to restrict output to a smaller area of the window.

Code: 1B 25

Parameters: CPX CPY SZX SZY

Notes:

- You cannot change a window's working area to be larger than the predefined area of the window as set by DWSet or OWSet.
- All drawing and window updating commands are done on the *current working area* of a window. The working area defaults to the entire size of the window. Scaling, when in use, is also performed relative to the current working area of a window. The CWArea command allows users to restrict the working area of a window to smaller than the full window size. Functions which might be done by opening a non-saved overlay window to draw or clear an image and then closing the overlay can be accomplished by using this command to shorten execution time where an actual overlay window is not needed.

DefColr Default Color

Function: Sets the palette registers back to their default values. The actual values of the palette registers depend on the type of monitor you are using. (See **montype** in *OS-9 Level Two Commands*.)

Code: 1B 30

Parameters: None

Notes:

- The default color definitions apply only to high-resolution graphics and text displays.
- The system sets the palette registers to a proper compatibility mode when switching to screens using the older VDG emulation modes. See the table below:

Window System Color Modes		VDG-Compatible Modes	
Palette	Color	P# Color	P# Color
00 & 08	White	00 Green	08 Black
01 & 09	Blue	01 Yellow	09 Green
02 & 10	Black	02 Blue	0A Black
03 & 11	Green	03 Red	0B Buff
04 & 12	Red	04 Buff	0C Black
05 & 13	Yellow	05 Cyan	0D Green
06 & 14	Magenta	06 Magenta	0E Black
07 & 15	Cyan	07 Orange	0F Orange

- The SetStat call lets you change the default color palette definition when using the windowing system. Default colors in the VDG-Compatible Mode cannot be changed. See the *OS-9 Level Two Technical Reference* manual for information on SetStat.
- The system's default colors are used whenever you create a new window.

DfnGPBuf Define GET/PUT Buffer

Function: Lets you define the size of the GET/PUT buffers for the system. Once you allocate a GET/PUT buffer, it remains allocated until you use the KilBuf command to delete it.

OS-9 allocates memory for GET/PUT buffers in 8K blocks that are then divided into the different GET/PUT buffers. Buffers are divided into buffer groups. Therefore, all commands dealing with GET/PUT buffers must specify both a group number and a buffer number within that group.

Code: 1B 29

Parameters: GRP BFN HBL LBL

Technical:

The buffer usage map is as follows:

Group Number	Buffer Number[1]	Use
0	1 - 255	Internal use only (returns errors)
1 - 199	1 - 255	General use by applications[2]
200 - 254	1 - 255	Reserved (Microware use only)[3]
255	1 - 255	Internal use only (returns errors)

[1] Buffer Number 0 is invalid and cannot be used.

[2] The application program should request its user ID via the GetID system call to use as its group number for buffer allocation.

[3] The standard group numbers are defined as follows:

Note: The names, buffer groups, and buffer numbers are defined in the assembly definition file. The decimal number you use to call these are in parentheses next to the name. For example, to select the Arrow pointer, Grp_Ptr and Ptr_Arr, you use 202,1 as the group/buffer number.

Grp_Fnt(200) = font group for system fonts
- Fnt_S8x8(1) = standard 8x8 font
- Fnt_S6x8(2) = standard 6x8 font
- Fnt_G8x8(3) = standard graphics font

The standard fonts are in the file SYS/StdFonts.

Grp_Clip(201) = clipboarding group (for Multiview)

Grp_Ptr(202) = graphics cursor (pointer) group
- Ptr_Arr(1) = arrow pointer (hp=0,0)
- Ptr_Pen(2) = pencil pointer (hp=0,0)
- Ptr_LCH(3) = large cross hair pointer (hp=7,7)
- Ptr_Slp(4) = sleep indicator (hourglass)
- Ptr_Ill(5) = illegal indicator
- Ptr_Txt(6) = text pointer (hp=3,3)
- Ptr_SCH(7) = small cross hair pointer (hp=3,3)

hp = hit point, the coordinates of the actual point on the object at which the cursor should be centered.

The standard pointers are in the file SYS/StdPtrs.

Grp_Pat2(203) = two color patterns
Grp_Pat4(204) = four color patterns
Grp_Pat6(205) = sixteen color patterns
- Pat_Dot(1) = dot pattern
- Pat_Vrt(2) = vertical line pattern
- Pat_Hrz(3) = horizontal line pattern
- Pat_XHtc(4) = cross hatch pattern
- Pat_LSnt(5) = left slanted lines
- Pat_RSnt(6) = right slanted lines
- Pat_SDot(7) = small dot pattern
- Pat_BDot(8) = large dot pattern

Each pattern is found within each of the pattern groups.

Standard patterns are in the files SYS/StdPats_2, SYS/StdPats_4, and SYS/StdPats_16.

All files have GPLoad commands imbedded in file, along with the data. To load fonts, pointers, or patterns, simply merge them to any window device: For example:

```
merge SYS/StdFonts ENTER
```

sends the standard font to standard output which may be redirected to another device if the current output device is not a window device (such as when term is a VDG screen).

You only need to load fonts once for the entire system. Once a Get/Put buffer is loaded, it is available to all devices and processes in the system.

DWEnd Device Window End

Function: Ends a current device window. DWEnd closes the display window. If the window was the last device window on the screen, DWEnd also deallocates the memory used by the window. If the window is an interactive window, OS-9 automatically switches you to a new device window, if one is available.

Code: 1B 24

Parameters: None

Notes:

- DWEnd is only needed for windows that have been attached via the iniz utility or the I$Attach system call. Non-attached windows have an implied DWEnd command that is executed when you close the path.

DWProtSw Device Window Protect Switch

Function: Disables and enables device window protection. By default, device windows are protected so that you cannot overlay them with other device windows. This type of protection helps avoid the possibility of destroying the contents of either or both windows.

Code: 1B 36

Parameters: BSW

BSW = switch
00 = off
01 = on (Default)

Notes:

- We recommend that you not turn off device window protection. If you do, however, use extreme discretion because you might destroy the contents of the windows. OS-9 does not return an error if you request that a new window be placed over an area of the screen which is already in use by an unprotected window.

DWSet Device Window Set

Function: Lets you define a window's size and location on the physical screen. Use DWSet after opening a path to a device window.

Code: 1B 20

Parameters: STY CPX CPY SZX SZY PRN1 PRN2 PRN3

PRN1 = Foreground
PRN2 = Background
PRN3 = Border (if STY $\geq$ 1)

Notes:

- The **iniz** and **display** commands open paths to the device window.
- When using DWSet in a program, you must first open the device.
- Output to a new window is ignored until OS-9 receives a DWSet command, unless defaults are present in the device descriptor (/w1-/w7). If defaults are present in the device descriptors, OS-9 automatically executes DWSet, using those defaults.
- When OS-9 receives the DWSet, it allocates memory for the window, and clears the window to the current background color. If the standard font is already in memory, OS-9 assigns it as the default font. If the standard font is not in memory, you must execute a font set (Font) command after loading the fonts to produce text output on a graphics window.
- Use the Screen Type code (STY) to define the resolution and color mode of the new screen. If the screen type code is zero, OS-9 opens the window on the process's currently selected screen. If the code is 01, OS-9 opens the window on the currently displayed screen. If the code is non-zero, OS-9 allocates a new screen for the window. The following describes the acceptable screen types:

Code	Screen Size	Colors	Memory	Type
FF	Current Displayed Screen[1]			
00	Process's Current Screen			
01	40 x 24	8 & 8	2000	Text
02	80 x 24	8 & 8	4000	Text
05	640 x 192	2	16000	Graphics
06	320 x 192	4	16000	Graphics
07	640 x 192	4	32000	Graphics
08	320 x 192	16	32000	Graphics

[1] Use the Current Displayed Screen option only in procedure files to display several windows on the same physical screen. All programs should operate on that process's current screen.

- The location of the window on the physical screen is determined by the diagonal line defined by:

  (CPX,CPY) and (CPX + SZX, CPY + SZY)

- The foreground, background, and border register numbers (PRN1, PRN2, and PRN3) define the palette registers used for the foreground and background colors. See the Palette command in this chapter for more information.

- When an implicit or explicit DWSet command is done on a window, the window automatically clears to the background color.

- All windows on the screen must be of the same type (either text or graphics).

- Values in the palette register affect all windows on the screen. However, you can choose which register to use for foreground and background for each window. That is, OS-9 maintains palette registers and border register numbers for the entire screen and foreground and background register numbers for each individual window.

- OS-9 deallocates memory for a screen when you terminate the last window on that screen.

FColor Foreground Color

Function: Lets you select a color palette register for the foreground color. See the Palette command for setting the actual colors.

Code: 1B 32

Parameters: PRN

Font Select Font

Function: Lets you select/change the current font. Before you can use this command, the font must be loaded into the specified GET/PUT group and buffer (using GPLoad). See the GPLoad command for information on loading font buffers.

Code: 1B 3A

Parameters: GRP BFN

Notes:

- You can select proportional spacing for the font by using PropSw.
- All font data is a 2-color bit map of the font.
- Each character in the font data consists of 8 bytes of data. The first byte defines the top scan line, the second byte defines the second scan line, and so on. The high-order bit of each byte defines the first pixel of the scan line, the next bit defines the next pixel, and so on. For example, the letter "A" would be represented like this:

Byte	Pixel Representation
10	. . . # . . . .
28	. . # . # . . .
44	. # . . . # . .
44	. # . . . # . .
7c	. # # # # # . .
44	. # . . . # . .
44	. # . . . # . .
00	. . . .

  Note that 6x8 fonts ignore the last 2 bits per byte.

- The fonts are ordered with characters in the following ranges:

$00-$1F	International characters (see mapping below)
$20-$7F	Standard ASCII characters

International characters or any characters in the font below character $20 (hex) are printed according to the following table:

Character position in font	Char1	or	Char2
$00	$C1		$E1
$01	$C2		$E2
$02	$C3		$E3
$03	$C4		$E4
$04	$C5		$E5
$05	$C6		$E6
$06	$C7		$E7
$07	$C8		$E8
$08	$C9		$E9
$09	$CA		$EA
$0A	$CB		$EB
$0B	$CC		$EC
$0C	$CD		$ED
$0D	$CE		$EE
$0E	$CF		$EF
$0F	$D0		$F0
$10	$D1		$F1
$11	$D2		$F2
$12	$D3		$F3
$13	$D4		$F4
$14	$D5		$F5
$15	$D6		$F6
$16	$D7		$F7
$17	$D8		$F8
$18	$D9		$F9
$19	$DA		$FA
$1A	$AA		$BA
$1B	$AB		$BB
$1C	$AC		$BC
$1D	$AD		$BD
$1E	$AE		$BE
$1F	$AF		$BF

GCSet Graphics Cursor Set

Function: Creates a GET/PUT buffer for defining the graphics cursor that the system displays. You must use GCSet to display a graphic cursor.

Code: 1B 39

Parameters: GRP BFN

Notes:

- To turn off the graphics cursor specify GRP as 00.
- A system standard buffer or a user-defined buffer can be used for the graphics cursor.

GetBlk Get Block

Function: Saves an area of the screen to a GET/PUT buffer. Once the block is saved, you can put it back in its original location or in another on the screen, using the PutBlk command.

Code: 1B 2C

Parameters: GRP BFN HBX LBX HBY LBY HSX LSX HSY LSY

HBX/LBX = *x-location of block* (upper left corner)
HBY/LBX = *y-location of block*
HSX/LSX = *x-dimension of block*
HSY/LSY = *y-dimension of block*

Notes:

- The GET/PUT buffer maintains information on the size of the block stored in the buffer so that the PutBlk command works more automatically.
- If the GET/PUT buffer is not already defined, GetBlk creates it. If the buffer is defined, the data must be equal to or smaller than the original size of the buffer.

GPLoad GET/PUT Buffer Load

Function: Preloads GET/PUT buffers with images that you can move to the screen later, using PutBlk.

If the GET/PUT buffer is not already created, GPLoad creates it.

If the buffer was previously created, the size of the passed data must be equal to or smaller than the original size of the buffer. Otherwise, GPLoad truncates the data to the size of the buffer.

Code: 1B 2B

Parameters: GRP BFN STY HSX LSX HSY LSY HBL LBL (Data...)

STY = *format*
HSX/LSX = *x-dimension of stored block*
HSY/LSY = *y-dimension of stored block*
HBL/LBL = *number of bytes in data*

Notes:

- Buffers are maintained in a linked list system.
- Buffers to be used most should be allocated last to minimize the search time in finding the buffers.
- When loading a Font GET/PUT Buffer, the parameters are as follows:

  GRP BFN STY HSX LSX HSY LSY HBL LBL (Data...)

  GRP = 254
  STY = 5
  HSX/LSX = *x-dimension size of Font 6 or 8*
  HSY/LSY = *y-dimension size of Font 8*
  HBL/LBL = *size of font data* (not including this header information)

  See the Font command for more information on font data.

KilBuf Kill GET/PUT Buffer

Function: Deallocates the buffer specified by the group and buffer number. To deallocate the entire group of buffers, set the buffer number to 0.

Code: 1B 2A

Parameters: GRP BFN

Notes:

- KilBuf returns memory used by the buffer to a free list. When an entire block of memory has been put on the free list, the block is returned to the system.

LSet Logic Set

Function: Lets you create special effects by specifying the type of logic used when storing data, which represents an image, to memory. The specified logic code is used by all draw commands until you either choose a new logic or turn off the logic operation. To turn off the logic function, set the logic code to 00.

Code: 1B 2F

Parameters: LCN

LCN = *logic code number*
00 = No logic code; store new data on screen
01 = AND new data with data on screen
02 = OR new data with data on screen
03 = XOR new data with data on screen

Notes:

- The following tables summarize logic operations in bit manipulations:

AND	First Operand	Second Operand	Result
	1	1	1
	1	0	0
	0	1	0
	0	0	0

OR	First Operand	Second Operand	Result
	1	1	1
	1	0	1
	0	1	1
	0	0	0

XOR	First Operand	Second Operand	Result
	1	1	0
	1	0	1
	0	1	1
	0	0	0

- Data items are represented as palette register numbers in memory. Since logic is performed on the palette register number and not the colors in the registers, you should choose colors for palette registers carefully so that you obtain the desired results. You may want to choose the colors for the palette registers so that LSet appears to *and*, *or*, and *xor* the colors rather than the register numbers. For example:

Palette #	Color	Alternative Order
0	White	Black
1	Blue	Blue
2	Black	Green
3	Green	White

OWEnd Overlay Window End

Function: Ends a current overlay window. OWEnd closes the overlay window and deallocates memory used by the window. If you opened the window with a *save switch* value of hexadecimal 01, OS-9 restores the area under the window. If you did not, OS-9 does not restore the area and any further output is sent to the next lower overlay window or to the device window, if no overlay window exists.

Code: 1B 23

Parameters: None

OWSet Overlay Window Set

Function: Use OWSet to create an overlay window on an existing device window. OS-9 reconfigures current device window paths to use a new area of the screen as the current logical device window.

Code: 1B 22

Parameters: SVS CPX CPY SZX SZY PRN1 PRN2

SVS = *save switch*
00 = Do not save area overlayed
01 = Save area overlayed and restore at close
PRN1 = *background palette register*
PRN2 = *foreground palette register*

Notes:

- If you set SVS to zero, any writes to the new overlay window destroy the area under the window. You might want to set SVS to zero if your system is already using most of its available memory. You might also set SVS to zero whenever it takes relatively little time to redraw the area under the overlay window once it is closed.
- If you have ample memory, specify SVS as 1. Doing this causes the system to save the area under the new overlay window. The system restores the area when you terminate the new overlay window. (See OWEnd.)
- The size of the overlay window is specified in standard characters. Use the same resolution (number of characters) as the device window that will reside beneath the overlay window. Have your program determine the original size of the device window at startup (using the SS.ScSiz GETSTAT call), if the device window does not cover the entire screen. See the *OS-9 Level Two Technical Reference* manual for information on the SS.ScSiz GETSTAT call.

- Overlay windows can be created on top of other overlay windows; however, you can only write to the top most window. Overlay windows are "stacked" on top of each other logically. To get back down to a given overlay, you must close (OWEnd) any overlay windows that reside on top of the desired overlay window.
- Stacked overlay windows do not need to reside directly on top of underlying overlay windows. However, all overlay windows must reside within the boundaries of the underlying device window.

Palette Change Palette

Function: Lets you change the color associated with each of the 16 palette registers.

Code: 1B 31

Parameters: PRN CTN

Notes:

- Changing a palette register value causes all areas of the screen using that palette register to change to the new color. In addition, if the border is set to that palette register, the border color also changes. See the Border command for more information.
- Colors are made up by setting the red, green, and blue bits in the color byte which is inserted in the palette register. The bits are laid out as follows:

Bit	Color
0	Blue low
1	Green low
2	Red low
3	Blue high
4	Green high
5	Red high
6	unused
7	unused

By using six bits for color (2 each for red, green and blue) there is a possibility of 64 from which to choose. Some of the colors are defined as shown:

```
White          : 00111111 = $3F (all color bits set)
Black          : 00000000 = $00 (no color bits set)
Standard Blue  : 00001001 = $09 (both blue bits set)
Standard Green : 00010010 = $12 (both green bits set)
Standard Red   : 00100100 = $24 (both red bits set)
```

Note: These colors are for RGB monitors. The composite monitors use a different color coding and do not directly match pure RGB colors. To get composite color from the RGB colors, the system uses conversion tables. The colors were assigned to match the RGB colors as close as possible. There are, however, a wider range of composite colors, so the colors without direct matches were assigned to the closest possible match. The white, black, standard green, and standard orange are the same in both RGB and composite.

PropSw Proportional Switch

Function: Enables and disables the automatic proportional spacing of characters. Normally, characters are not proportionally spaced.

Code: 1B 3F

Parameters: BSW

BSW = *switch*
00 = off (Default)
01 = on

Notes:

- Any standard software font used in a graphics screen can be proportionally spaced.
- Proportional spacing is not supported on hardware text screens.

PSet Pattern Set

Function: Selects a preloaded GET/PUT buffer as a pattern RAM array. This pattern is used with all draw commands until you either change the pattern or turn it off by passing a parameter of 00 as GRP (Group Number).

Code: 1B 2E

Parameters: GRP BFN

Notes:

- The pattern array is a 32 x 8 pixel representation of graphics memory. The color mode defines the number of bits per pixel and pixels per byte. So, be sure to take the current color mode into consideration when creating a pattern array.
- The GET/PUT buffer can be of any size, but only the number of bytes as described by the following table are used:

Color Mode	Size of Pattern Array
2	4 bytes x 8 = 32 bytes (1 bit per pixel)
4	8 bytes x 8 = 64 bytes (2 bits per pixel)
16	16 bytes x 8 = 128 bytes (4 bits per pixel)

- The buffer must contain at least the number of bytes required by the current color mode. If the buffer is larger than required, the extra bytes are ignored.
- To turn off patterning, set GRP to 00.

- The following example creates a two color pattern of vertical lines. A two color pattern is made up of 1's and 0's. The diagram below shows the bit set pattern (note that one pixel is equal to one bit):

  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010
  10101010101010101010101010101010

  When the binary for the 2x8 pixel data is compressed into byte data, notice that each row consists of 4 bytes of data. The pattern now looks like this:

  $55 $55 $55 $55
  $55 $55 $55 $55
  $55 $55 $55 $55
  $55 $55 $55 $55 $55 = 01010101
  $55 $55 $55 $55
  $55 $55 $55 $55
  $55 $55 $55 $55
  $55 $55 $55 $55

  To load the pattern in the system, use the GPLoad command. To load this particular pattern into Group 2 and Buffer 1, the command would be:

```
display 1b 2b 02 01 00 20 00 08 00 20 55 55 ...55 [ENTER]
```

32 times
number of bytes (32)
y size of pattern (8)
x size of pattern (32)
buffer number
group number
GPLoad code

- When making a pattern using 4 colors, a pixel is made up of two bits instead of one. This means that the pattern consists of 64 bytes instead of 32. The diagram below shows the bit set pattern for the same vertical pattern using 4 colors:

```
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
```

When the binary for the 4x8 pixel data is compressed into byte data, notice that each row consists of 8 bytes of data. The pattern now looks like this:

```
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
$CC $CC $CC $CC $CC $CC $CC $CC
```

$CC = 1100

To load the pattern in the system, use the GPLoad command as described for the 2 color example but specify $40 (64) bytes instead of $20 (32).

- When making a pattern using 16 colors, a pixel is made up of four bits instead of one. This means that the pattern consists of 128 bytes. Each line in the bit pattern would look like this:

  11110000...{repeat pattern for 16 total sets}...11110000

  When the binary for the 8x8 pixel data is compressed into byte data, the pattern is a series of $F0.

  To load the pattern in the system, use the GPLoad command and specify $80 (128) bytes as the size.

PutBlk Put Block

Function: Moves a GET/PUT buffer, previously copied from the screen or loaded with GPLoad, to an area of the screen.

Code: 1B 2D

Parameters: GRP BFN HBX LBX HBY LBY

HBX/LBX = *x-location of block* (upper left corner)
HBY/LBY = *y-location of block*

Notes:

- The dimensions of the block were saved in the GET/PUT buffer when you created it. OS-9 uses these dimensions when restoring the buffer.
- The screen type conversion is automatically handled by the PutBlk routine in the driver.
- GET/PUT buffers cannot be scaled. The image will be clipped if it does not fit within the window.

ScaleSw Scale Switch

Function: Disables and enables automatic scaling. Normally, automatic scaling is enabled. When scaling is enabled, coordinates refer to a relative location in a window that is proportionate to the size of the window. When scaling is disabled, coordinates passed to a command will be the actual coordinates for that type of screen relative to the origin of the window.

Code: 1B 35

Parameters: BSW

BSW = *switch*
00 = off
01 = on (Default)

Notes:

- A useful application of disabled scaling is the arrangement of references between a figure and text.
- All coordinates are relative to the window's origin (0,0). The valid range for the coordinates:

  Scaling enabled:

  y = 0-191
  x = 0-639

  Scaling disabled:

  y = 0-*size of y - 1*
  x = 0-*size of x - 1*

Select Window Select

Function: Causes the current process's window to become the active (display) window. You can select a different window by using the form:

```
display 1B 21 >/wnumber
```

where *number* is the desired window number. If the process that executes the select is running on the current interactive (input/display) window, the selected window becomes the interactive window, and the other window becomes passive.

Code: 1B 21

Parameters: None

Notes:

- The keyboard is attached to the process's selected window through the use of the CLEAR key. This lets you input data from the keyboard to different windows by using the CLEAR key to select the window.
- All display windows that occupy the same screen are also displayed.
- The device window which owns the keyboard is the current interactive window. The interactive window is always the window being displayed. Only one process may receive input from the keyboard at a time. Many processes may be changing the output information on their own windows; however, you can only see the information that is displayed on the interactive window and any other window on the same screen as the interactive window.

TCharSw Transparent Character Switch

Function: Defines the character mode to be used when putting characters on the graphics screens.

In the default mode (transparent off), the system uses block characters that draw the entire foreground and background for that cell.

When in transparent mode, the only pixels that are changed are the ones where the character actually has pixels set in its font. When transparent mode is off, all pixels in the character block are set: foreground or font pixels in the foreground color and others in the background color.

Code: 1B 3C

Parameters: BSW

BSW = *switch*
00 = off (Default)
01 = on

Chapter 4

Drawing Commands

All drawing commands relate to an invisible point of reference on the screen called the *draw pointer*. Originally, the draw pointer is at position 0,0. You can change the position by using the SetDPtr and RSetDPtr commands described in this chapter. In addition, some draw commands automatically update the draw pointer. For example, the LineM command draws a line from the current draw pointer position to the specified end coordinates and moves the draw pointer to those end coordinates. The Line command draws a line but does not move the pointer.

Also, note that all draw commands are affected by the pattern and logic commands described in Chapter 3.

Do not confuse the draw pointer with the graphics cursor. The graphics cursor is the graphic representation of the mouse/joystick position on the screen.

In this chapter, commands that use relative coordinates (*offsets*) are listed with their counterparts that use absolute coordinates. For example, RSetDPtr is listed under SetDPtr.

Arc3P Draw Arc

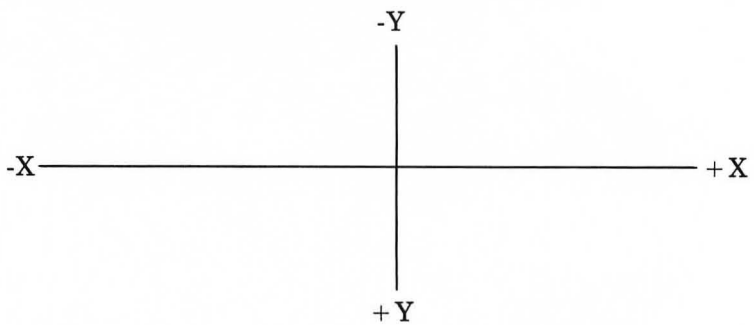
Function: Draws an arc with its midpoint at the current draw pointer position. You specify the curve by both the X and Y dimensions, as you do an ellipse. In this way, you can draw either elliptical or circular arcs. The arc is clipped by a line defined by the (X01,Y01) - (X02,Y02) coordinates. These coordinates are signed 16 bit values and are relative to the center of the ellipse. The draw pointer remains in its original position.

Code: 1B 52

Parameters: HBRx LBRx HBRy LBRy HX01 LX01 HY01 LY01 HX02 LX02 HY02 LY02

Notes:

- The resulting arc depends on the order in which you specify the line coordinates. Arc3P first draws the line from Point 1 to Point 2 and then draws the ellipse in a clockwise direction.
- The coordinates of the screen are as follows:



Bar Draw Bar

Function: Draws and fills a rectangle that is defined by the diagonal line from the current draw pointer position to the specified position. The box is drawn in the current foreground color. The draw pointer returns to its original location.

Code: 1B 4A

Parameters: HBX LBX HBY LBY

RBar Relative Draw Bar

Function: Draws and fills a rectangle that is defined by the diagonal line from the current draw pointer position to the point specified by the offsets. The box is drawn in the current foreground color. The draw pointer returns to its original location. This is a relative command.

Code: 1B 4B

Parameters: HBXo LBXo HBYo LBYo

Box Draw Box

Function: Draws a rectangle that is defined by the diagonal line from the current draw pointer position to the specified position. The box is drawn in the current foreground color. The draw pointer returns to its original location.

Code: 1B 48

Parameters: HBX LBX HBY LBY

RBox Relative Draw Box

Function: Draws a rectangle that is defined by the diagonal line from the current draw pointer position to the point specified by the offsets. The box is drawn in the current foreground color. The draw pointer returns to its original location. This is a relative command.

Code: 1B 49

Parameters: HBXo LBXo HBYo LBYo

Circle Draw Circle

Function: Draws a circle of the specified radius with the center of the circle at the current draw pointer position. The circle is drawn in the current foreground color. The draw pointer remains in its original location.

Code: 1B 50

Parameters: HBR LBR

Ellipse Draw Ellipse

Function: Draws an ellipse with its center at the current draw pointer position. The X value specifies the horizontal radius, and the Y value specifies the vertical radius. The ellipse is drawn in the current foreground color. The draw pointer remains in its original location. This is a relative command.

Code: 1B 51

Parameters: HBRx LBRx HBRy LBRy

FFill Flood Fill

Function: Fills the area where the background is the same color as the draw pointer. Filling starts at the current draw pointer position, using the current foreground color. The draw pointer returns to its original location. This is a relative command.

Code: 1B 4F

Parameters: None

Line Draw Line

Function: Draws a line from the current draw pointer position to the specified point, using the current foreground color. The draw pointer returns to its original location.

Code: 1B 44

Parameters: HBX LBX HBY LBY

RLine Relative Draw Line

Function: Draws a line from the current draw pointer position to the point specified by the *x,y* offsets, using the current foreground color. The draw pointer returns to its original location. This is a relative command.

Code: 1B 45

Parameters: HBXo LBXo HBYo LBYo

LineM Draw Line and Move

Function: Draws a line from the current draw pointer position to the specified point, using the current foreground color. The draw pointer stays at the new location.

Code: 1B 46

Parameters: HBX LBX HBY LBY

RLineM Relative Draw Line and Move

Function: Draws a line from the current draw pointer position to the point specified by the offsets, using the current foreground color. The draw pointer stays at the new location. This is a relative command.

Code: 1B 47

Parameters: HBXo LBXo HBYo LBYo

Point Draw Point

Function: Draws a pixel at the specified coordinates, using the current foreground color.

Code: 1B 42

Parameters: HBX LBX HBY LBY

RPoint Relative Draw Point

Function: Draws a pixel at the location specified by the offsets, using the current foreground color. This is a relative command.

Code: 1B 43

Parameters: HBXo LBXo HBYo LBYo

PutGC Put Graphics Cursor

Function: Puts and displays the graphics cursor at the specified location. The coordinates passed to this command are not window-relative. The horizontal range is 0 to 639. The vertical range is 0 to 191. The default position is 0,0.

This command is useful for applications running under GrfInt so that you can display a graphics cursor under WindInt even if you don't want mouse control of the cursor.

Code: 1B 4E

Parameters: HBX LBX HBY LBY

SetDPtr Set Draw Pointer

Function: Sets the draw pointer to the specified coordinates. The new draw pointer position is used as the beginning point in the next draw command if other coordinates are not specified.

Code: 1B 40

Parameters: HBX LBX HBY LBY

RSetDPtr Relative Set Draw Pointer

Function: Sets the draw pointer to the point specified by the offsets. The new draw pointer position is used as the beginning point in the next draw command if other coordinates are not specified. This is a relative command.

Code: 1B 41

Parameters: HBXo LBXo HBYo LBYo

Chapter 5

Text Commands

The text commands let you control the cursor's position and movement and also the way text prints on the display. These commands can be used on either text or graphics windows.

The text commands are:

Code	Description
01	Homes the cursor.
02 *x y*	Positions cursor to *x,y*. Specify coordinates as (*x* + $20) and (*y* + $20).
03	Erases the current line.
04	Erases from the current character to the end of the line.
05 20	Turns off the cursor.
05 21	Turns on the cursor.
06	Moves the cursor right one character.
07	Rings the bell.
08	Moves the cursor left one character.
09	Moves the cursor up one line.
0A	Moves the cursor down one line.
0B	Erases from the current character to the end of the screen.
0C	Erases the entire screen and homes the cursor.
0D	Sends a carriage return.
1F 20	Turns on reverse video.
1F 21	Turns off reverse video.
1F 22	Turns on underlining.
1F 23	Turns off underlining.
1F 24	Turns on blinking.[1]

Code	Description
1F 25	Turns off blinking.[1]
1F 30	Inserts a line at the current cursor position.
1F 31	Deletes the current line.
1B 3C BSW	See TCharSw in Chapter 3.[2]
1B 3D BSW	See BoldSw in Chapter 3.[2]
1B 3F BSW	See PropSw in Chapter 3.[2]

[1] Blink is not supported for text on graphics screens.

[2] These characteristics are supported for text on graphics screens only.

Index

active window 3-34
AND 3-21
arc, draw 4-2
ARC3P 4-2

background color 3-2, 3-13
BAR 4-3
bar, draw 4-3
bar, relative draw 4-3
BCOLOR 3-2
bell, ring 5-1
blinking 5-1, 5-2
    off 5-2
    on 5-1
boldface 3-3, 5-2
BOLDSW 3-3, 5-2
BORDER 3-4, 3-26
border color 3-4, 3-13, 3-26
BOX 4-4
box, draw 4-4
box, relative draw 4-4
buffer, kill 3-20
buffer, load 3-19
buffers 3-7, 3-19
buffers
    close 3-20
    define 3-7
    font 3-19
    get/put 3-19
    get/put 3-7
        patterns 3-29
        save 3-18
    group numbers 3-7
    kill 3-20
    load 3-19
    logic 3-21
    pattern 3-29
        16-color 3-31
        2-color 3-30
        4-color 3-31
    pattern array 3-29
    pattern size 3-29
    put 3-32
        block 3-32
buffers (*cont'd*)
        screen type 3-32
        size 3-32
    save 3-18

carriage return 5-1
change font 3-15
character
    erase 5-1
    transparent 3-35
CIRCLE 4-5
circle, draw 4-5
close buffer 3-20
close overlay window 3-23
close, window 3-10, 3-23
color 3-26
    background 3-2, 3-13
    border 3-4, 3-13, 3-26
    composite 3-27
    default 3-6
    foreground 3-13, 3-14, 3-26
    graphics 3-6
    high-resolution 3-6
    palette 3-26
    RGB 3-26
    VDG-emulation 3-6
command parameters 2-1
commands
    drawing 2-1, 4-1
    general 2-1, 3-1
    text 2-1, 5-1
composite colors 3-27
create windows
    device 1-2
    overlay 1-4
current window 3-13, 3-34
cursor 3-17
    home 5-1
    graphics 3-17
        put 4-11
    move 5-1
    off 5-1
    on 5-1

cursor (*cont'd*)
position 5-1
set 3-17
CWAREA 3-5

default color 3-6
default windows 1-3
DEFCOLR 3-6
define buffers 3-7
define device windows 3-12
delete line 5-2
device descriptors 1-1, 3-12
device windows 1-1
background color 3-13
border color 3-13
color
background 3-13
border 3-13
foreground 3-13
define 3-12
end 3-10
foreground color 3-13
keyboard 3-34
location 3-13
process window 3-13
protect 3-11
select 3-34
set 3-12
DFNGPBUF 3-7
DISPLAY 1-2, 2-1, 3-12
draw pointer 4-1
relative set 4-12
set 4-12
draw
arc 4-2
bar 4-3
bar, relative 4-3
box 4-4
box, relative 4-4
circle 4-5
ellipse 4-6
fill 4-7
flood fill 4-7
line 4-8
line and move 4-9

draw (*cont'd*)
line and move, relative 4-9
line, relative 4-8
point 4-10
point, relative 4-10
pointer 4-1
drawing commands 4-1
DWEND 3-10
DWPROTSW 3-11
DWSET 1-1, 1-2, 3-12

ELLIPSE 4-6
ellipse, draw 4-6
end overlay window 3-23
end window 3-10, 3-23
erase character 5-1
erase line 5-1, 5-2
erase screen 5-1
erase to end of screen 5-1
escape sequence 1-2

FCOLOR 3-14
FFILL 4-7
fill, draw 4-7
flood fill, draw 4-7
FONT 3-15
font 3-12, 3-15, 3-19
bit map 3-15
boldface 3-3, 5-2
change 3-15
current 3-15
data 3-15
load 3-19
order 3-15
proportional 3-15, 3-28, 5-2
font bit map 3-15
font data 3-15
font load 3-19
font order 3-15
foreground color 3-13, 3-14, 3-26

GCSET 3-17

general commands 2-1, 3-1
get/put buffers
    close 3-20
    define 3-7
    font 3-19
    group numbers 3-7
    kill 3-20
    load 3-19
    logic 3-21
    patterns 3-29
    put 3-32
    save 3-18
GETBLK 3-18
GPLOAD 3-9, 3-15, 3-19, 3-32
graphic patterns 3-29
graphics
    boldface 3-3
    colors 3-6
    cursor 3-17, 4-11
        put 4-11
        set 3-17
    font, proportional 3-28
    transparent 3-35
group numbers 3-7, 3-8
    Grp_Clip 3-8
    Grp_Fnt 3-8
    Grp_Pat2 3-8
    Grp_Pat4 3-8
    Grp_Pat6 3-8
    Grp_Ptr 3-8

high-resolution, colors 3-6
home cursor 5-1

I$ATTACH 3-10
immortal shell 1-3
INIZ 1-2, 3-10, 3-12
insert line 5-2
interactive window 3-34

KILBUF 3-20
kill buffer 3-20

LINE 4-8
line
    delete 5-2
    draw 4-8
    erase 5-1, 5-2
    insert 5-2
    relative draw 4-8
line and move, draw 4-9
line and move, relative draw 4-9
LINEM 4-1, 4-9
load get/put 3-19
load font 3-15, 3-19
location, window 3-13
logic operations 3-21, 4-1
logic set 3-21
LSET 3-21

memory 1-1, 3-12
MONTYPE 3-6
move cursor 5-1

open windows
    device 1-2
    overlay 1-4
OR 3-21
overlay window 1-1, 1-2, 3-23, 3-24
overlay window
    end 3-23
    no-save 3-24
    save 3-24
    select 3-34
    set 3-24
    size 3-24
    stacked 3-25
OWEND 3-23
OWSET 1-4, 3-24

PALETTE 3-26
palette colors 3-26
parameters, command 2-1
pattern 3-29, 4-1
    16-color 3-31
    2-color 3-30

pattern (*cont'd*)
    4-color 3-31
    array 3-29
    size 3-29
POINT 4-10
point, draw 4-10
point, relative draw 4-10
pointer, draw 4-1
position cursor 5-1
process window 3-13
proportional characters 3-28, 5-2
proportional font 3-28
PROPSW 3-28, 5-2
protect device windows 3-11
PSET 3-29
put block 3-32
put buffer 3-32
    screen type 3-32
    size 3-32
put graphics cursor 4-11
PUTBLK 3-18, 3-19, 3-32
PUTGC 4-11

RBAR 4-3
RBOX 4-4
reverse video 5-1
RGB colors 3-26
ring bell 5-1
RLINE 4-8
RLINEM 4-9
RPOINT 4-10
RSETDPTR 4-1, 4-12

save, get/put 3-18
save window 3-18
SCALESW 3-33
scaling 3-33
scaling, automatic 3-33
scaling coordinates 3-33
screen type 3-12
screen, erase 5-1
SELECT 3-34
select window 3-34
set
    device window 3-12
    draw pointer 4-12
    draw pointer, relative 4-12
    overlay window 3-24
SETDPTR 4-1, 4-12
SETSTAT 3-6
SHELL 1-3
shell, fork 1-3
stacked overlay windows 3-25

TCHARSW 3-35, 5-2
text commands 5-1
text
    boldface 3-3, 5-2
    proportional 3-28, 5-2
transparent character 3-35, 5-2

underline
    off 5-1
    on 5-1

video, reverse 5-1

WCREATE 1-2
windows 1-1
    background color 3-2, 3-13
    boldface 3-3, 5-2
    border color 3-4, 3-13, 3-26
    buffers 3-7, 3-19
        kill 3-20
        load 3-19
        patterns 3-29
        put 3-32
    close 3-10, 3-23
    color 3-26
        background 3-2, 3-13
        border 3-4, 3-13, 3-26
        composite 3-27

windows (*cont'd*)
default 3-6
foreground 3-13, 3-14, 3-26
RGB 3-26
current 3-13
cursor 3-17
default 1-3
default color 3-6
device 1-1
define 3-12
end 3-10
opening 1-2
protect 3-11
select 3-34
set 3-12
device descriptors 1-1, 3-12
end 3-10
fonts 3-15, 3-19
foreground color 3-13, 3-14, 3-26
graphics cursor 3-17, 4-11
interactive 3-34
keyboard 3-34
location 3-13
logic operations 3-21
maximum 1-1
windows (*cont'd*)
memory 1-1, 3-12
overlay 1-1, 1-2, 3-23, 3-24
end 3-23
no-save 3-24
opening 1-4
save 3-24
select 3-34
set 3-24
size 3-24
stacked 3-25
process window 3-13
process 3-13
protect 3-11
put buffer 3-32
save 3-18
scaling 3-33
screen type 3-12
select 3-34
size 3-5
transparent mode 3-35, 5-2
type 1-1
work area 3-5
work area 3-5

XOR 3-21

OS-9
Glossary

OS-9 Glossary

active processes. Operations that the system is currently executing.

active state. An operating or working condition. A procedure in an active state is processing data and not waiting for another procedure to end.

address. A number that identifies a location in your computer's memory.

age. A count of the number of switches (process changes) the system has made since a process's last time slice.

anonymous directory. A directory referenced by its hierarchical position using the period (.) character. One period refers to the current directory. Two periods refer to the parent of the current directory, and so on.

application program. A process or group of processes designed to accomplish specific tasks, such as word processing, data management, game playing, and so on.

argument. Data you supply to a process or command for it to evaluate.

array. Data arranged so that each item is located by its row and column position. Single-dimensioned arrays have one or more rows and one column. Multi-dimensioned arrays have one or more rows and two or more columns.

ASCII code. American Standard Code for Information Interchange. A method of defining alphabetic and numeric characters and other symbols by giving each a unique value. For instance, the ASCII value for A is 65, and the ASCII value for B is 66.

assembler. A program that produces machine code from source code (code from a low-level computer language).

assembly language. A system for coding computer instructions to perform tasks. You can use assembly language code to directly manipulate data within a computer; therefore, assembly language needs less interpretation than higher level languages like BASIC or Pascal.

attribute. *See* file attribute.

background processing. Executing one or more procedures and at the same time continuing to operate in OS-9 or in another procedure.

backup. An identical copy of the contents of one disk on another disk.

base. The lowest value allowed in a function or operation. For instance, BASIC09 allows a base value of 1 for array structures unless you indicate otherwise.

batch file. *See* procedure file.

baud. Bits-per-second. A unit for measuring the speed of data flow between devices.

binary. A numbering system using only two digits, 0 and 1. In this system, shifting the position of a digit to the left raises the value of the digit by the power of 2. For instance, 1 is the binary equivalent of 1, 10 = 2, 100 = 4, 1000 = 16, and so on.

bit. The smallest unit of a computer's memory. Eight bits form a byte. Each bit can have a value of either 0 or 1.

bit map. A storage area of 256 bytes. Each bit represents one page (256 bytes) of your computer's memory. If a bit is set (equals 1) then its associated memory page is allocated. If a bit is reset (equals 0) then its associated memory page is free.

block. A group of data, often comprising 256 bytes.

block-oriented device. A device that receives data, sends data, or both, in groups of 256 bytes.

Boolean logic. A binary type of algebra developed by George Boole.

Boolean data type. A type of variable that can have only two values, True or False. Boolean data types usually store the results of comparisons, such as: is A greater than B (A>B), does Y equal X (Y=X), and so on.

boot. The process of loading and initializing OS-9.

bootfile. A disk file containing modules to be loaded during an OS-9 boot.

bootlist. A disk file containing a list of module names to be used by OS9Gen to create a bootfile.

bootstrap module. A program that contains the code necessary to initialize OS-9.

border. An area around a screen or window that defines the boundaries of the screen or window.

branch. To leave one routine and begin execution of another routine within a program or procedure.

breakpoints. Locations in a program or procedure at which you want execution to pause.

buffer. A temporary storage area through which OS-9 transfers data.

byte. A unit of computer memory storage that contains a value in the range 0-255.

byte data type. A numeric type of variable that can contain unsigned eight-bit integer data (in the range 0-255 decimal).

call. (1) To transfer execution to another routine, then return to the calling procedure with obtained values intact and available for use by the calling routine. (2) A built-in OS-9 routine that performs a system function.

CC3Disk. The floppy diskette driver module.

CC3IO. The system input/output driver.

chaining. A process of calling and turning over system control to a new procedure.

checksum. A value calculated from the contents of a file or module that the system can later use to verify whether the contents of the file or module are uncorrupted.

child or child process. A process begun from another (parent) process.

close. The process of deallocating the path to a device or file.

cluster. A group of sectors. In OS-9 for the Color Computer, a cluster consists of only one sector.

code. Numeric data that can be used by a computer to perform a task.

command. The name of an OS-9 program or function.

command line. One or more commands with all their parameters, options, and modifiers.

command modifiers. Data or values appended to a command that change the way the command functions.

command options. Data that you can include in a command line to specify the way the command functions.

command parameters. Data or values appended to a command that define or customize the command.

command separator. A semicolon. You can use a semicolon to separate several commands on the same command line.

compile. To create machine language code (object code) from a program written with a computer language. Also to translate high level code (from a high level language such as BASIC) into low level code (code that is like machine language).

complement. A value that is derived by subtracting a number from a constant. For example, the 10s complement of 4 is 6. In binary, a value is complemented by changing all the 1 digits to 0 and all the 0 digits to 1, then adding 1 to the least significant (rightmost) digit.

complex data structure. A group of data that contains two or more types of data structures. *See* data structure.

constant. A value or block of data that is fixed (does not change during the run of a program or procedure).

CPU. Central Processing Unit. An integrated circuit (chip) that controls the operation of a computer.

current directory. The directory in which OS-9 looks for data files or stores data files unless you specify otherwise.

current line. When editing, the line on which the editing cursor or pointer is located.

cursor (text). A colored box that shows where the next character is to appear on the screen. A text cursor appears on both text and graphics windows or screens.

cyclic redundancy check (CRC). A value the system calculates from the data stored in a module. The system calculates a new value each time it attempts to load the module. If the calculated value does not match the CRC value contained in the module, the system cannot load the module.

cylinder. A disk track that includes both sides of a disk. *See also* track.

DAT. Dynamic address translation. The memory management system used by OS-9 Level Two.

data directory. The directory in which OS-9 automatically saves files unless you specify otherwise.

data structure. A unit of data, organized for access.

data type. A method for representing data, such as character (ASCII value), integer (whole number), or real (floating point number).

deadlock. *See* deadly embrace.

deadly embrace. A situation in which two processes attempt to gain control of the same disk areas at the same time.

debug. To find and correct program errors.

decompile. To translate machine language code into a computer language code.

delimiter. A character that divides items. For instance, in OS-9, the semicolon is a delimiter that divides two commands on the same line.

descriptor. *See* device descriptors.

device. A data source, destination, or both. OS-9 devices can exist in your computer's memory (such as a window or a RAM disk), or they can be external equipment (such as a printer or disk drive).

device descriptors. Small tables that define a device, its name, its driver, and its file manager. Device descriptors also contain port initialization data and port address information.

device drivers. Modules that handle basic input/output functions for specific devices. Each device you use with your computer must have its own driver to interpret the code you send it.

device name. A unique system word for a device. The name for disk Drive 0 is /D0, the name for Window 1 is /W1, and so on.

device table. *See* device descriptor.

device window. An OS-9 device from which you can run a program or utility. Access device windows in the same manner as you do other devices. Each device window has its own input and output buffers. Refer to windows using device names (/W followed by a number), such as /W1, /W2, /W3, and so on.

directory. A file in which OS-9 stores a list of other files, including their names, locations on the disk, attributes, and so on.

disk allocation map. Logical Sector Number 1 on a disk. The data in LSN1 indicates which sectors are allocated to files and which sectors are free.

double click. To press and release the mouse button twice in quick succession when the pointer is over the desired location.

drag. To hold down the mouse button and move the mouse to a new position before releasing the button.

draw pointer. An indicator that determines where the next graphics draw command will begin unless you specify otherwise.

driver. *See* device driver.

dump. To write the contents of a video screen, a memory location or a file to another terminal, memory location, or file.

echo. To cause data being sent to one device to go to another device, as well.

edit. To change the data or values in a file or in your computer's memory.

edit buffer. An alternate workspace for the OS-9 Macro Editor.

edit macro. A series of commands you can execute with only a single command.

edit pointer. An indicator that determines where the next edit command is to operate unless you specify otherwise.

editor. A program that provides special commands to aid you in changing the contents of a file.

error code. A code that OS-9 displays when it cannot understand what you want it to do or when your computer or a peripheral malfunctions. Use the displayed code number to look up a description of the error.

error path. The route through which OS-9 sends error codes and other information to display on the screen. The error path is designated as Path Number 2.

error trap. A routine in a procedure that checks for an error and provides an alternate action (other than terminating execution and displaying a system error message).

executable file. A program file that you can run by typing and entering its filename.

execute. To start a procedure, program, or command (cause it to run).

execution directory. The disk directory that contains your system's command files.

execution modifiers. *See* command modifiers.

execution offset. The location in a program or subroutine at which execution begins, calculated from the beginning of the module.

expressions. Data items joined by arithmetic operators. *See also* operator.

expression stack. A memory location in which BASIC09 stores temporary results while it evaluates an expression.

file. (1) A block of information your computer uses for a particular function or program. A file can contain an operating system, a language, an application program, or text. (2) A collection of associated records, such as information about each book in a library.

file attribute. Data that identifies a file, for instance its size, security status, language type, and so on.

file locking. Protecting a file to ensure that one process does not change it while another process is using it.

file pointer. An indicator that determines where in a file the next read or write operation is to occur unless you or the system indicates otherwise.

file security. A set of attributes that determines who can use a file and in what manner.

filename. A set of characters that uniquely identifies and locates a block of data stored on a disk.

filter. To alter data in some manner as it passes between two devices or between two memory locations.

flag. A symbol or value that indicates when a certain condition exists in a procedure.

font. A character set. A group of alphabetic and numeric characters and other symbols of a particular style and shape.

foreground. (1) An OS-9 task that takes priority over other concurrently running tasks. (2) Characters or designs on a screen or window.

fork. The process of initializing one procedure from another procedure.

format. To magnetically organize a diskette so that the computer can use it to store data.

function. In BASIC09, an operation that BASIC performs on data. A function always returns (produces) a value of some type.

Get/Put buffer. A buffer in which you or the system can store fonts, screen patterns, graphic displays, overlay windows, and other recallable data. The system allocates Get/Put buffers in 8-kilobyte blocks.

getstat. An OS-9 routine that gets (returns) the state or status of a specific system operation.

global variable. A variable that is available to all procedures and routines in a program.

graphics. An arrangement of elements (lines, dots, and so on) on your computer's screen.

graphics cursor. An indicator (either visible or invisible) that determines where the next graphic function is to occur on the screen unless you or a program specifies otherwise. In applications, you often move the graphics cursor using a mouse.

graphics pointer. *See* graphics cursor.

graphics screen. A screen in which all pixels are represented by bits in a memory map. You create images on the screen by manipulating the bits using special OS-9 or computer language commands.

graphics window. A window created on a graphics screen. You can display both graphics (drawn images) and text on a graphics window. The text generated on a graphics window/ screen uses software fonts that you or the system must load into memory.

group. An organization of related data or files. For instance, OS-9's graphics buffers are organized into groups that you reference by number.

hardware. The physical parts of your computer, including its disk drives, keyboard, integrated circuits (chips), and so on.

header. Data located at the beginning of a file or module to identify its type, size, verification values, and so on.

hexadecimal. A number system to a base of 16 (using 16 digits). Hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Shifting a hexadecimal digit one place to the left causes its value to be multiplied by 16.

high order bit. The most significant or leftmost bit in a byte. If the high order bit is 0, it represents a value of 0. If the high order bit is 1, it represents a value of 128.

I/O. Input/Output.

identification sector. Logical Sector Number 0 on a disk. LSN0 contains a description of the physical and logical organization of a disk.

immortal shell. An OS-9 shell that does not die on receiving an EOF signal (such as when you press CTRL BREAK).

integer data type. A type of variable that can store whole numbers in the range -32768 to 32767.

interactive window. A window that is getting input from the keyboard. This window is currently on the displayed screen.

interface. To link devices or modules together in order to transfer data.

internal integrity check. A system of internal values that OS-9 can use to make certain that its system modules and functions are accurate.

IOMAN. The input/output manager that provides common processing for all input/output operations.

IRQ. Interrupt request. A signal that causes the execution of one process to halt and the execution of another process to begin. The system retains the values of the first process so that it can later continue its execution.

kernel. OS-9 software that supervises the OS-9 system and provides basic system services, such as multitasking and memory management, and that links all system modules.

key sequence. Two or more keys you press at the same time to produce a specific function.

keyboard mouse. An OS-9 function that lets you use the keyboard arrow keys instead of an external mouse device. Press [CTRL][CLEAR] to toggle the keyboard mouse on and off.

keyword. A command name.

kill. Terminate the execution of a process.

kilobyte. 1024 bytes.

link. To make a module available to a process.

link count. The number of processes using a module. When a module's link count reaches 0, OS-9 deallocates the module.

load. To transfer data from an external device into your computer's memory.

local variable. A variable that can be used by only the procedure or routine in which it resides.

locked. *See* file locking.

lockout. *See* file locking.

log in. To initiate the necessary procedure to operate OS-9 from a separate terminal (type in a user name and password).

logical address. An offset address. An address that is numbered from the beginning of a block rather than from the beginning of memory, a module, or other storage area.

logical sectors. Sectors that OS-9 or a program references in numeric order, regardless of their actual physical location on a disk.

loop. A sequence of BASIC09 commands that execute repeatedly a specified number of times or until a specific condition occurs to terminate the execution.

macro. A series of commands you can execute with only a single command name.

map. *See* memory map.

mask. A pattern of bits that you use in combination with a logical operator to change specified data selectively — reversing certain bits without affecting the others.

megabyte. One million bytes.

memory. The portion of your computer that stores data and values.

memory management. Assigning and mapping memory to keep track of modules (processes and data) and their uses.

memory map. A chart depicting the use of your computer's memory by the operating system.

menu. A screen display from which you select an action for your computer.

microprocessor. An integrated circuit (chip) that controls the basic operation of your computer.

mode. A particular function of a program or system.

modem. Modulator/demodulator. A device to prepare signals for transmission through telephone lines and to reverse the process after transmission.

modifier. *See* command modifier.

module. An OS-9 program or block of data residing in your computer's memory.

module body. A module's code (program or data), including the module name.

module directory. A table in your computer's memory that lists all the modules residing in memory.

module header. Code that resides at the beginning of all modules and that contains information about the module, including size, type, attributes, storage requirements, and its execution starting address.

monitor. The video display device connected to your computer.

mouse. A device you use to control a pointer on the display screen. In application programs, you can often use a mouse to indicate functions you want to initiate.

multi-programming. A method of computer operation in which the system allocates *slices* of execution time to more than one process in order to execute them concurrently.

multi-tasking. Executing more than one process at the same time.

multi-user. A system that lets more than one person access its functions at the same time.

nesting. Incorporating one structure into another structure of the same type. Both procedures then retain their individual identities.

non-shareable file. A file or module that can be used by only one procedure or user at one time.

null string. A string variable that does not contain a value (has a length of 0 characters).

object code. Machine language instructions.

offset. The difference between a location and a beginning location. For instance, you can tell some BASIC09 graphics functions to begin operation at a location that is offset from the current draw pointer position.

operand. A value that is used or manipulated during an operation or during the execution of an instruction.

operating system. A set of associated programs that carry out your commands.

operator. A symbol or word that signifies some action to be performed on specified data.

options. *See* command options.

output path. The route through which the system sends data from one device to another.

overflow. A condition in which a storage space is not large enough to contain the data sent to it.

overlay. A condition in which programs or modules in a computer's memory are replaced with other data.

overlay window. A window opened or placed on top of a device window.

overwrite. To replace data with other data.

owner. An entity that has control over a file, module, or process.

pack. To compile a BASIC09 procedure. *See* compile.

padding. Adding spaces to a string or unit of data to make it a specific length.

page. In your computer's memory, a block of 256 bytes.

paint. To fill all or a portion of the screen with a color.

palette. A register that contains a numeric code representing a color or shade.

parameters. *See* command parameters.

parent or parent process. A process that forks (starts) another process (a child process).

parity. A system in which all binary numbers of a code are converted to either even-bit numbers (an even number of 1s) or odd-bit numbers (an odd number of 1s).

parse. To search through a list or sequence of data.

Pascal. A high-level computer language.

pass by value. When BASIC09 passes a value from one procedure to another by evaluating a constant or expression and placing the result in temporary storage to be accessed by the second procedure.

pass by reference. When BASIC09 passes a variable from one procedure to another by providing the second procedure with the address of the variable's storage.

passive window. Any window that is not receiving input from the keyboard. A *process* can be running on a passive window provided that the process is getting its input from a source other than the keyboard.

pathlist. The route from one position in a disk's directory to another directory or file.

peripherals. Devices connected to your computer, such as printers, disk drives, and so on.

permission. The attributes of a file or module that determine who can use the file or module and in what manner.

physical sectors. The actual arrangement of sectors on a disk's surface, regardless of any internal organization by OS-9.

pipe. A function in which the output of one process becomes the input of another process.

pipeline. A series of commands, each of which passes the results of its operations to the next command in the series.

PIPEMAN. The pipe file manager. Pipes are memory buffers acting as files to transfer data between processes.

pixel. The smallest area of a display screen that can be manipulated (turned off or on).

pointer. An indicator that determines a location in memory, in a file, or on the screen.

port. A junction between devices through which data flows. An electrical connection between your computer and a peripheral.

position independent module. A module that need not be loaded at any certain location in memory.

procedure. A program or routine your computer can execute.

procedure file. A file containing one or more OS-9 commands. You can execute a procedure file in the same manner as you execute OS-9 commands or programs.

process. A computer program or a routine that performs a specific task as part of a computer program.

process descriptor. A block of data that includes information about a process, its state, memory allocations, priority, queue pointers, and so on.

process ID. A unique number the system gives each process it executes.

process priority. A value you or the system gives to a process that determines the amount of CPU (execution) time it is to receive in a multi-tasking environment.

process state. The condition of a process in regard to its execution. A process can be active (executing), waiting (awaiting its turn for processing), or sleeping (inactive until it receives a signal to awaken).

program. Code that causes your computer to perform some function or a series of functions.

program modules. Executable code. Modules you can run to perform a function or series of functions.

public. Any user of a program or module other than the owner. *See* owner.

purge. Delete. Usually refers to removing all, or a selected group, of files from a directory.

RAM. Random access memory. Computer memory you can write to (change) and read from.

RAM disk. A portion of your computer's memory that OS-9 can use for data storage and retrieval in the same manner as it uses an external disk drive. However, be certain you copy RAM disk data to a floppy diskette or hard disk before you exit OS-9 or turn off your computer. If you do not, the data is lost.

random access. Reading (accessing) information in a block of data without first having to read any preceding data.

raw data. Unformatted information that is passed to a device exactly as it exists.

RBF. The random block file manager that processes all disk input/output.

re-entrant programs. Programs or modules that can be used by more than one process at the same time.

read. The process of transferring data from a device into the computer's memory.

read permission. System permission to read (withdraw data from) a file.

real data type. A type of variable that can store floating point numbers in the range $\pm 1 \times 10^{\pm 38}$

record. A collection of related data items that a program or process considers to be a unit for the purpose of processing. A subdivision of a file, such as all information about a single item in an inventory file.

record locking. Protecting a portion of a file to ensure that one process does not change it while another process is using it.

recursive procedure or routine. A procedure or routine that repeatedly executes itself (that contains a statement causing it to run itself one or more times).

register. A location within a computer's memory (often in the CPU) for storing values during arithmetic, logic, or transfer operations.

remarks. Text contained in a program that describes the program itself and that is not to be executed.

ROM. Read only memory. Computer memory containing constant values that the computer can read but cannot change.

ROOT directory. The parent directory of all files and directories on a disk. The ROOT directory is created by FORMAT.

run. To execute, or to cause a program or procedure to start.

runtime. The duration of a program's execution.

SCF. The sequential character file manager that handles non-disk input/output operations to devices such as printers and terminals.

scratched. Destroyed. When you copy one file over another file, or the contents of one disk onto another disk, any data existing in the second file or on the second disk is scratched.

sector. A division of a disk track. Disk tracks are organized into several sectors.

seek. To position a file pointer at a specific byte location in a file.

semigraphics. Graphics (designs on the display screen) using ASCII graphic characters.

sequential access. The process of reading data in order, one character at a time.

sequential execution. Executing a series of commands or processes, one after the other.

sequential file. A file consisting of records of various lengths that must be accessed one after the other, starting at the first record.

serial. Refers to transmissions in which data leaves or arrives at a location or device, with data units following one after the other in space or time.

Setstat. An OS-9 routine that sets (changes) the state or status of a specific system operation.

shell. The command interpreter.

sibling. One of two or more processes executed by the same parent process.

sign bit. The leftmost bit of a binary number that serves as an indicator to show whether the number is positive or negative. Normally, a value of 0 indicates positive, and a 1 indicates negative.

signal. An interrupt from the system or another process that changes a procedure's or a device's state. For example, signals set an active process to a waiting state, awaken an inactive or sleeping process, or change the display window.

single step. A procedure in the Debug mode that lets you execute one procedure statement and (optionally) view the results.

single-user file. A file that only one person can access at a time.

single-user module. A program that only one person can use at a time.

sleeping state. A situation where you or the system suspends a process for a specified time or until you or the system sends it a wakeup signal.

source code. Program code produced using a computer language. Before it can control a computer, source code must be translated into machine language, either by a compiler or a translator program. *See also* compiler.

stack. A storage area in your computer's memory in which data can be placed or recovered in sequence, from one end only.

standard error path. The route through which your computer sends error codes and other messages to the screen.

standard input path. The route through which you can send data to your computer (usually the keyboard).

standard output path. The route your computer uses to send data to the screen.

start up. To turn on your computer and initialize OS-9.

stop bits. One or two bits that a terminal program sends after each unit of data to indicate that the transmission of the unit is complete.

string. A group of alphanumeric characters.

string data type. A type of variable that can contain one or more ASCII values (values representing alphanumeric characters or other symbols). String data types can be any length, up to the capacity of your computer's memory.

structured programming. Building a program out of a series of procedures, each of which performs a specific task but combines with its associated procedures as one program.

subdirectory. A directory that resides within another (parent) directory.

subroutine. An operation that performs a specific task as part of a larger operation.

super user. The system user who has control of the entire system and access to all system files and modules. User Number 0.

symbolic debugging. An error correcting system that lets you pause program execution and view the current values of variables, using their program names.

syntax. The rules for forming legal instructions for your computer.

system. (1) A group of files and programs that provide you with control over your computer. (2) Your computer with all its attached devices.

system call. Built-in OS-9 routines that perform particular functions, such as accessing disk files, printing data on the screen, and so on.

table. A storage area in memory or on disk containing ordered data to be used by a process or function.

task. A unit of work performed by your computer.

terminal. A computer or an electronic device, with a screen and keyboard, connected to your Color Computer 3. You can access OS-9 functions from a terminal in the same manner as you can access them from your Color Computer 3 keyboard.

text files. Files containing printable characters, or the code representing such characters.

text screen. A Type 1 or 2 screen. Text screens use hardware generation of characters (fonts are not definable) and are often referred to as hardware screens or windows. Text screens cannot display graphics. Text operations occur faster in text windows/screens than on graphic windows/screens.

text window. Any window created on a hardware text screen.

time slice. The period of time between system clock ticks. A tick occurs every 1/60th of a second.

timesharing. A situation in which more than one person uses the same operating system.

token. In the BASIC language, a numeric value that represents a keyword.

trace. To display each procedure statement as it executes and view its results.

tracks. Magnetically created concentric circles created on a disk for the storage of data. Tracks are established when you format a disk.

transparent characters. Characters that display over screen images without erasing any of the area surrounding the characters.

unlink. To remove a module (program) from your computer's memory.

update mode. The condition of a file when it is open for both reading and writing.

user ID. A number that identifies the operator to which a process belongs.

user number. *See* user ID.

utility. A short program that performs a frequently required task, usually for the maintenance of your computer system or files.

variable. A unit of storage with no fixed value. You define a variable and locate it in your computer's memory using a variable name.

VDG. Video Display Graphics.

vector. A graphics line or portion of a line.

verify. To check data for accuracy.

wait state. A situation in which a process remains suspended until one of its child processes terminates or until it receives a wakeup signal from the system.

wake up. To continue the execution of a process that has been suspended.

wild card. A symbol that represents or takes the place of one or more other characters or symbols.

WINDINT. Window interface.

window. All or a portion of your video screen with specific formats (columns, lines, size, colors, and so on) and type (graphics, text, or both). An area of a screen in which you can run a process or which can receive input.

word length. The number of bits to transmit as one unit.

workspace. A portion of your computer's memory that BASIC09 establishes for the storage and manipulation of procedures.

write. To transfer data from the computer's memory to a device.

write permit. System permission to change the data in a file.

write protect. A method of protecting a diskette so that your computer cannot change the data on it.

RADIO SHACK
A Division of Tandy Corporation
Fort Worth, Texas 76102

6/87-SWCG 874-8010 Printed in U.S.A.