OS-9/68000
OPERATING SYSTEM
USER'S MANUAL

The information contained herein is believed to be accurate as of the date of publication, however, Microware will not be liable for any damages, including indirect or consequential, from use of the OS-9 operating system or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Revision E
Publication date: July 1985
Publication Editor: Walden Miller

Microware Systems Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322
(515)224-1929

PMN-OSU68

OS-9/68000 OPERATING SYSTEM USER'S MANUAL

PREFACE

OS-9/68000 is a powerful and versatile operating system that can help you take fullest advantage of your 68000 system's capabilities. OS-9 offers a very wide selection of functions because it was designed to serve the needs of a broad audience. Whether you are a casual user or a professional programmer, you will find many useful features in OS-9.

This manual has been designed for use as a reference and learning guide. It is the basic user reference manual for OS-9. The "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" is a companion manual for advanced programmers who wish to learn about the internal operation and functions of the system.

At first glance, the OS-9 manual set (especially the "Technical Manual"!) may seem overwhelming. Fortunately, you only need to know a fairly small percentage of the material presented in this manual to use OS-9 effectively. In fact, you don't even need to open the "Technical Manual" unless you plan to do some very serious programming or system-level work. You will find that it is easy to learn about OS-9 as you continue to work and experiment with it.

The secret to getting up to speed quickly with OS-9 is to first identify and learn only the basic, everyday functions necessary to run applications programs and programming languages.

This manual contains six chapters:

Chapter 1 is a general introduction to OS-9. It introduces the concept of an operating system and explains some of the basic features of OS-9.

Chapter 2 describes how to get OS-9 up and running. This includes formatting and backup procedures.

Chapter 3 will help you get started using the operating system quickly. The more frequently used system commands are discussed. These are utility commands that every user should be familiar with.

Chapter 4 is a detailed explanation of the tree-structured file and directory system of OS-9. This includes:

A. Using directories.

B. Types of files.

C. File security.

D. Movement around the file/directory system.

Chapter 5 contains a detailed description of the "Shell", the OS-9 user interface.

Chapter 6 consists of detailed explanations of the OS-9 utility commands.

All bold print words (with the exception of chapter headings) are OS-9 commands with detailed descriptions in chapter 6 or are terms defined in appendix B (the Glossary).

OS-9/68000 OPERATING SYSTEM USERS MANUAL

TABLE OF CONTENTS

OS-9/68000 OPERATING SYSTEM USERS MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS

OS-9/68000 OPERATING SYSTEM USER'S MANUAL
CHAPTER 1
AN OVERVIEW OF OS-9

WHAT IS AN OPERATING SYSTEM ?

An operating system's primary function is to be the master supervisor of the resources and functions of a computer system. Computer resources consist of memory, CPU time, and input/output devices (such as terminals, disk drives, printers, etc).

OS-9 is an operating system for microcomputers which utilize 68000 or 6809 microprocessors. OS-9/68000 is the specific version for 68000-based systems. The basic functions of OS-9 are:

1. To provide an interface between the computer and the user.
2. To manage the input/output (I/O) operations of the system.
3. To provide for the loading and execution of programs.
4. To create and manage a system of directories and files.
5. To manage timesharing and multitasking.
6. To allocate memory for various purposes.

The most visible function of the operating system is its role as an interface between the user and the technically complex internal hardware and software functions of the system. Even though OS-9 is a sophisticated operating system, it was specially designed to make its powerful features easy to use (even by persons with limited technical knowledge).

An operating system provides only part of the overall software necessary to make the computer useful. Applications programs such as word processors and accounting packages tend to be the most frequently used programs. They are not part of the operating system but rely heavily on the services provided by the operating system (input/output for example). Most applications programs are written by users or obtained from commercial software suppliers.

Similarly, programming languages are tools used to create applications programs. These rely heavily on and are closely related to the operating system.

OS-9 also includes a set of about 50 programs called utility commands. While technically not part of the basic operating system (they actually are small applications programs), they provide many essential housekeeping, management, customization and maintenance functions. Some of these utility commands such as the text editor are useful general-purpose applications programs.

ACCESSING OS-9'S FUNCTIONS

There are two basic ways you can use OS-9's many capabilities and functions.

The first method uses the utility command set, and in particular the SHELL command interpreter program. These programs let you type in OS-9 commands directly from your keyboard. The utility programs take English-like input and translate it to the more complex internal system calls actually required to carry out the desired operations. All OS-9 utility commands are described in detail in Chapter 6.

The second method uses system calls. These are requests made to OS-9 within programs written in assembler or a high-level language. All programming languages have special statements that cause the program to use OS-9 system calls, often in a hidden manner.

There are system calls to load programs into memory, create new tasks, create or delete files, read, write, open or close files and so on. Programming language user manuals describe how the languages interface to OS-9. System calls are mostly of interest to advanced programmers and are discussed in detail in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL."

MULTITASKING AND MULTIUSER FEATURES

OS-9 is a **multitasking** and **multiuser** operating system.

Multitasking (or multiprocessing) means that the computer can run many different programs at the same time. By rapidly switching from one program to the next, many times per second, it gives the appearance that all programs are running at the same time. Each program running on the system is called a **task** (or **process**).

OS-9's multitasking capabilities make it possible for efficient memory use, CPU time and I/O operations to be shared by all programs without conflict. This is very important because it frees the computer from the limitation of doing only one thing at a time. You can be editing a document while another is printing. Or the computer could allow an industrial control system to manage many processes at the same time. Multitasking is the basis of multiuser operation.

Multiuser (or **timesharing**) operation is a natural extension of the system's basic multitasking functions. It allows several people to interactively use the computer simultaneously. OS-9 provides additional security related timesharing functions. These functions control access to the system and privacy within the system.

The multitasking and multiuser capabilities tremendously increase OS-9's versatility. OS-9 is often used as a single-user/multitasking system on small computers. It is also used as a multiuser/multitasking system on larger computer systems. In either case, there is no difference in OS-9 itself, the application software, or how either works.

THE MEMORY MODULE AND MODULAR SOFTWARE

A unique feature of OS-9 is its support of modular software techniques based on memory modules. The use of memory modules can:

1. Provide more efficient use of available disk and memory storage.
2. Make the system run faster.
3. Simplify programming jobs.
4. Make it easy to customize and adapt OS-9 itself.

All OS-9 programs are kept in the form of one or more program modules that contain pure program code. They do not contain variable storage (OS-9 assigns that in a separate block of memory at run time). Each module has a unique name and can be loaded into memory or stored on a disk or tape. OS-9 automatically keeps track of the names and locations of all modules that are present in memory.

An important characteristic of memory modules is the sharing of one module by several tasks (or users) at the same time. For example, if four users want to run BASIC09 at the same time, only one copy of the BASIC09 program module will be loaded into memory. Other operating systems would typically load four exact copies of Basic into memory, thus requiring 300% more memory. The shared module system is completely automatic and usually transparent to the user.

Another advantage of memory modules is that frequently used functions can share common "library" modules. For example, a standard OS-9 module called "Math" provides basic floating point arithmetic operations for virtually all programming languages and programs. Again, this eliminates the need for each program to include its own math package. It also means that if you add a hardware floating point processor to your system, you only need to replace this one module and all your other software will automatically be converted without modification. In addition, large and complex programs can be split up into smaller, testable modules.

end of chapter 1

BOOTING OS-9

Before you can use OS-9 on your computer, it is necessary to boot the system. Booting (also called a cold start or bootstrapping) involves the computer reading a portion of the system disk into memory.

If your system is a standard disk-based computer, the system disk contains all the modules that make up OS-9. The system disk usually contains other files and directories that are frequently used during normal operations. This includes a directory for each user, a shared commands directory and files that are used by the system.

The files "OS9boot" and "startup" are stored on the system disk. The startup file is a SHELL procedure file that is processed immediately after the system starts running. The startup file may contain any legal OS-9 command or program. The OS9boot file contains the OS-9 system modules that are read into memory.

The boot procedure will vary depending on the requirements of the specific hardware. The manufacturer will supply detailed instructions outlining the specific boot procedure for the specific system involved. You should follow the instructions as specified.

If the system fails to boot, recheck the hardware setup instructions (especially if you had to make any modifications to your computer). Make sure the disk was inserted correctly, and try the boot sequence again. If the boot sequence fails several times, contact your supplier.

When the system boots correctly, a welcoming message will be displayed followed by the prompt:

```
$
```

The "$" prompt means the operating system is active and waiting for you to enter a command line.

BACKING UP THE SYSTEM DISK

Before you try any experimenting with OS-9, it is necessary to make a backup of your master system disk. The BACKUP procedure involves making an exact copy of the disk. If, for some reason, your disk becomes damaged, it is likely that the disk will become unreadable. For this reason, it is important to have another copy stored safely away. The following section describes the steps to be taken to backup a disk on a typical OS-9 system that boots from drive 0 (d0).

FORMATTING A DISK

The first OS-9 utility that you will be introduced to is the **FORMAT** command. FORMAT is used to initialize new disks for reading and writing. You can not read or write a new disk until it has been formatted. BACKUP, the OS-9 command that makes copies of disks, requires the backup disk to be the same size and format as the original.

The format of OS-9 system disks vary by the type of disk drive and by manufacturer. Usually, the format is set to be the maximum capacity of the disk drive. Several parameters can be placed on the command line with the FORMAT command:

-sd	single density
-dd	double density
-ss	single sided
-ds	double sided

Refer to your hardware documentation for the maximum capacity of your drives and the label of your system disk for the proper format of your backup copy. Consult the FORMAT command description in Chapter 6 for other parameters.

Multiple Drive Format

If your system has two disk drives, place the system disk in the first drive and the new disk in the second drive. The second drive is usually called /d1. Now, after the "$" prompt, type "format", the drive name of the new disk, the options specified by the system disk and strike the <return> key to enter the command line:

```
format /d1 -ds -dd
```

This command line specifies that the disk in drive 1 will be formatted as a double-sided, double-density disk. If your disk is different, your options will be different.

Single Drive Format

If your system has only one disk drive, it will be necessary to load the FORMAT command into memory. The LOAD command puts a copy of a program into the memory of the computer. Loading FORMAT will now allow you to remove your system disk from the drive. OS-9 will not have to read the disk to find it now. OS-9 can execute the copy that resides in memory. Any OS-9 command can be loaded and executed in this fashion. To load FORMAT, type the following command after the "$" prompt:

```
load format
```

REMOVE the system disk from the drive.

PLACE the disk to be formatted into the drive.

After the "$" prompt, type:

```
format /d0 -ss -dd
```

This command line specifies that the disk will be formatted as a single sided double density disk. Again, if your disk is different, your options will be different.

In the case of both single and multiple drive systems, the FORMAT command will display a table of variables (see FORMAT listing in Chapter 6) followed by a prompt:

```
Formatting on drive <drive name>
proceed?
```

NOTE: <drive name> will be replaced by the name of the device you are trying to format on.

IF THE DRIVE NAME IN THE PROMPT IS NOT THE NAME OF THE DRIVE WHERE YOU HAVE THE BLANK DISK, type "q" to quit, or your only system disk may be erased.

If the drive name in the prompt is correct, type "y" for yes. If you type "y" at the prompt, FORMAT will prompt for the name that you wish to call the disk.

During the final phase of the process, the hexadecimal number of each track is displayed as each track is verified to see if there are any bad sectors. If any bad sectors are found, an error message will be displayed along with the number of the bad sector.

NEVER BACKUP A SYSTEM DISK TO A DISK THAT HAS ANY BAD SECTORS REPORTED BY FORMAT.

THE BACKUP PROCEDURE

The BACKUP command will make an exact copy of the OS-9 system disk. There are other ways to make a copy of a disk, but this method is the least complicated. After a disk is formatted, you are ready to run BACKUP. The BACKUP process involves copying everything from your system disk to a formatted disk (see figure 1 on the following page). In the BACKUP procedure, the system disk is referred to as the source disk. The new backup disk is called the destination disk.

NOTE: This procedure will make copies of any disk (not just the system disk) of which you need a copy.

The BACKUP command uses two passes (phases). The first pass reads a portion of the source disk into a buffer in memory and writes it to the destination disk. The second pass verifies that everything was copied to the new disk correctly.

If an error occurs on the first pass, there is something wrong with the source disk or the drive that it is in. If an error occurs during the second pass, the problem is with the destination disk. If BACKUP repeatedly fails on the second pass, reformat the disk to make sure there are no bad sectors on the disk. If the disk reformats correctly, try to backup again.

Figure 1 on the following page outlines the step by step procedure for backing up disks for multiple or single drive systems.

NOTE: You may wish to "write protect" your source disk with a "write protect" tab when using the BACKUP procedure on a single drive system. This will prevent any accidental confusion in exchanging the source and destination disks.

TWO DRIVE BACKUP PROCEDURE

STEP 1: PUT DESTINATION DISK IN DRIVE 1	
STEP 2: YOU TYPE: backup	SYSTEM RESPONSE: ready to BACKUP /D0 to /D1?
STEP 3: YOU TYPE: y (yes)	(no response disk is backed up)

SINGLE DRIVE BACKUP PROCEDURE

STEP 1: PUT YOUR SOURCE DISK IN THE DRIVE	
STEP 2: YOU TYPE: backup /d0	SYSTEM RESPONSE: ready to BACKUP /D0 to /D0
STEP 3: YOU TYPE: y (yes)	SYSTEM RESPONSE: ready destination, hit a key
STEP 4: REMOVE THE SOURCE DISK	
STEP 5: INSERT THE DESTINATION DISK	
STEP 6: HIT ANY KEY	
The next prompt will ask you to insert the source disk. Step 3 through Step 6 will be repeated until backup is complete.	

figure 1: The Backup Procedure

When you have backed up the system disk, store the original in a safe place and use the duplicate as your working system disk.

end of chapter 2

USER NOTES

OS-9/68000 OPERATING SYSTEM USER'S MANUAL
CHAPTER 3
BASIC COMMANDS AND FUNCTIONS

LEARNING THE BASICS

Now that you have your system up and running, it's time to learn about some of the basic features and utility commands of OS-9. This chapter along with Chapter 4 provide a "fast-track" introduction to OS-9 designed to get you started quickly.

The secret to getting up to speed quickly with OS-9 is to first identify and learn only the basic, everyday functions necessary to run applications programs and programming languages. It is fairly easy to learn more as you continue to work with the system.

The basic topics covered in this chapter are:

1. Use of the keyboard and display
2. The page pause feature
3. Logging on timesharing systems
4. The "SHELL" and command lines
5. Basic utility commands

USING THE KEYBOARD

Most input to OS-9, programming langauges and applications programs is line oriented. This means that as you type, the characters are collected but not given to the program until you hit the RETURN key. This gives you a chance to correct typing errors before they are given to the program.

OS-9 has several features that make data entry and error correction simple. These are called line editing features. Each of these features use control keys, which are generated by simultaneously pressing the CONTROL and some other character key.

The line editing control keys are:

KEY	FUNCTION
CONTROL A	Repeats the previous input line. The last line entered will be redisplayed but not executed. The cursor will be positioned at the end of the line. You may enter the line as is or you can add more characters to it. You can edit the line by backspacing and typing over old characters.
CONTROL D	Redisplays the current input line. This is mainly used for hardcopy terminals that can not erase deleted characters.
CONTROL H	Backspaces to erase previous characters. Most keyboards have a special BACKSPACE key that can be used directly without using the CONTROL key.
CONTROL Q	Resumes output that was previously halted by CONTROL S. The CONTROL Q function is known as ION.
CONTROL S	Halts output until CONTROL Q is entered. The CONTROL S function is known as IOFF. This is a function used by many serial I/O devices (such as printers) to control output speed.
CONTROL W	Temporarily halts output so you can read the screen before data scrolls off. Output resumes when any other key is hit. See the discussion below of the "screen pause" feature.
CONTROL X	Deletes line: erases the entire current line.
ESCAPE or CONTROL [	Indicates end-of-file: All OS-9 I/O devices (including terminals) are accessed as files. This simulates the effect of reaching the end of a disk file.

There are also two other important control keys called "interrupt" keys. They work differently than the line editing keys in that they can be used at any time, not just when a program has requested input. They are normally used to halt or alter a running program.

CONTROL C — Sends an "interrupt" signal to the most recent program. This functions differently from program to program. If the program does not make specific "interrupt" provisions, it will abort the program. If the program has provisions for interrupts, CONTROL C usually provides a way to stop the current function and return to a master menu or command mode. In the SHELL, it can be used to convert the "foreground" program to a "background" program if the program has not begun I/O to the terminal.

CONTROL E — Sends a "program abort" signal to the program presently running. This key will prematurely ABORT the current program and return you to the SHELL.

The control keys described above are the key assignments commonly used in almost all OS-9 systems. The correspondence between control keys and their functions is changable, so your keys may be different. The TMODE utility command (detailed in chapter 6) can be used to redefine the usage of control keys. This command allows the customization of OS-9 to the specific computer's keyboard layout.

THE PAGE PAUSE FEATURE

The page pause feature is designed to eliminate the annoyance of having output scroll off the screen before you have a chance to read it. To eliminate this, OS-9 counts output lines until the a full screen has been displayed. It then halts output until you hit any key. This is repeated for each screen of output.

Page pause can be fooled by lines that are longer than the physical width of the screen. These long lines wrap around to the next line. The system can not distinguish this and consequently can not count them properly.

The TMODE command (See Chapter 6) may be used to turn this feature on and off, or to change the number of lines per screen.

LOGGING ON A TIMESHARING SYSTEM

If you are using a single user system such as a personal computer or work station, you can skip this section. Otherwise, you need to know how to log on to a multiuser system. This applies to both "hardwire" and "dial-up" terminals.

Idle terminals on multiuser systems do nothing but beep at you until you hit the RETURN key. This starts the log-on program called LOGIN. Its function is to maintain system security and start up each user with a personalized environment.

The system will ask you for the user name and password assigned to you by the system manager. As you enter these the system will echo the user name but the password (for security purposes). You have three chances to enter a valid user name and password.

An example of the LOGIN procedure is given below:

```
OS-9/68000 Level 1 Version 1.2 Timesharing System  04/01/85 14:51:12

User Name: smith
Password: [not echoed]

Process #10 logged on  04/01/85 14:51:20

Shell V1.2

$
```

Depending on how the system is set up, you may also see a system-wide "message of the day". It is also possible for the system to automatically run one or more initial programs for you. Also, you will normally be set up in your own main working directory.

To log off, simply hit the ESCAPE (end-of-file) key any time your main SHELL is active.

For more information see the LOGIN and TSMON command descriptions in Chapter 6.

THE SHELL AND COMMAND LINES

"SHELL" is the name of a program which is OS-9's command interpreter. It is therefore your primary interface to the system. It is automatically run for you following the system startup sequence on a single user system, or after logging on a timesharing system.

Many functions and options are provided by the SHELL. Chapter 5 is devoted to an in-depth discussion of the of features available. This section is intended to give you just enough familiarity with the SHELL for you to run basic OS-9 commands.

The SHELL can be used in two basic ways. The first and most frequent mode is accepting interactive commands from your keyboard. In the second method, the SHELL reads a sequence of commands lines from a file (called a "procedure file") and runs them just as if they had been typed in manually. This is a convenient way to eliminate typing frequently used, identical sequences of commands.

When the SHELL is ready for a command input it displays a "$" prompt. You may then enter a command line followed by a carriage return.

The first word of the command line is the name of a command, which may be in upper or lower case. The command may be:

1. The name of an OS-9 utility command.

2. The name of an application program or programming language.

3. The name of a procedure file.

The command line can have additional optional words which give the program and/or the SHELL some additional information such as file names, options, etc. Almost all options are preceded by a dash "-" character. All optional words are separated by space characters.

The SHELL follows a special searching sequence in order to try to locate the command in memory or on disk. If it can not find the command you specified, it generally will report the error: 000:216, "file not found."

Here is an example of a simple SHELL command line:

```
$ list myfile
```

The name of the command word is "list". The file name "myfile" was also given to be passed to the program.

BASIC COMMANDS

Chapter 6 of this manual includes over fifty standard utility commands that are provided with OS-9. The majority of them are used rarely, if ever, by casual users. You will frequently use less than a dozen of them, and less frequently use about a dozen more.

We have broken down the complete utility command set into four groups to give you an idea of what you should and should not bother learning immediately. It is recommended that you get acquainted with the first group now, and the second group as time permits. If you plan to do advanced programming or systems-level work you can study the other two groups at your convenience.

Group 1: The Very Basic Commands

BACKUP	BUILD	COPY	DATE
DEL	DIR	FORMAT	FREE
LIST	MFREE	RENAME	SETIME

Group 2: Other Useful Basic Commands

ATTR	CHD	CHX	DELDIR
ECHO	EDT	KILL	MAKDIR
MERGE	PD	PR	PROCS
SET	SHELL	TMODE	

Group 3: Advanced Programming Commands

BINEX	CFP	CMP	CODE
COMPRESS	COUNT	DEBUG	DSAVE
DUMP	EX	EXBIN	EXPAND
GREP	LOAD	MAKE	PRINTENV
QSORT	SAVE	SETENV	TEE
TOUCH	TR	UNSETENV	

Group 4: System Management Commands

DCHECK	DEINIZ	FIXMOD	IDENT
INIZ	LINK	LOGIN	MDIR
OS9GEN	SETPR	SLEEP	TSMON
UNLINK	XMODE		

All Group 1 commands are introduced in chapters 2 and 4 except for the ones discussed in the following pages.

SETIME AND DATE

The first command that should be executed after the system comes up is SETIME. Most systems automatically run SETIME during the system startup sequence. SETIME starts the system clock and also allows OS-9 to keep track of the date and time of creation of new files. The clock must be running for multitasking to take place. SETIME is executed by typing:

```
$ setime
```

SETIME will prompt with:

```
yy/mm/dd hh:mm:ss [am/pm]
Time ?
```

At the prompt, enter the year, month, day, hour, minutes, seconds and optionally am or pm. Unless am or pm is specified, SETIME uses the 24 hour clock (1520 is 3:20 pm). The input is one or two digit numbers with a space, colon, semicolon, comma, or slash used as field delimiters (if a semicolon is used, the entire date string must be within quotes). For example, to set the time on July 31, 1984 at 1:24 pm, you would type:

```
84/7/31/1/24/pm     or     84 07 31 1 24 pm    or

84,7,31,13,24       or     84:7:31:13:24       or

84/7/31/13/24       or     "84;7;31;13;24"
```

To find out if the system clock is running or to see if the date and time was set correctly, use the DATE command. Example:

```
$ date
July 31, 1984 Tuesday 1:25:26pm
```

FREE AND MFREE

You might be interested in knowing how much space you have available on your disk and in your computer's memory.

During the FORMAT procedure, a disk is divided into data sectors of 256 bytes each. These sectors, in turn, are allocated into groups called clusters. The number of sectors per cluster is dependent on the storage capacity and physical characteristics of the given device.

FREE will display the amount of unused disk space in number of sectors. It will also display the disk name, its creation date and the cluster size of the device. For example:

```
$ free /d1
68000 Documentation Disk created on:  85/06/01
Capacity:  1,232 sectors (1-sector clusters)
935 Free sectors,  largest block 568 sectors
```

MFREE will display the address and size of unused memory available for allocation. For even more information concerning the unused memory, the "-e" option may be used with MFREE. For example:

```
$ mfree

Current total free RAM:      #181504     $2C118


$ mfree -e

Minimum allocation size:          0.25 K-bytes
Number of memory segments         6
Total RAM at startup              256 K-bytes
Current total free RAM            171.75 K-bytes

Free memory map:

     Segment Address          Size of Segment
     ----------------     -------------------------
        $29100               $F00          3.75 K-bytes
        $2D100               $26E00      155.50 K-bytes
        $57800               $1600         5.50 K-bytes
        $59400               $100          0.25 K-bytes
        $59600               $1800         6.00 K-bytes
        $59B00               $300          0.75 K-bytes
```

end of chapter 3

OS-9 FILES

One of the major functions of any operating system is the storage of information that the user creates. Without some way to store and organize programs, data and text, all work would be lost when the computer's power is turned off. OS-9 stores information in files and **directories** that are located on mass-storage devices such as floppy disks and provides easy access methods for update, storage and retrieval.

Files are the heart of the OS-9 system. When you write programs, edit text or collect data, you are using files.

Files consist of an ordered sequence of **bytes**. A file may contain a program, characters of text, or almost anything else. Within a text file each byte contains one character. Data is written to and read from the file exactly as given. The file's size can be up to the maximum capacity of the disk. A file can be expanded or shortened as desired.

Bytes within the file are addressed like memory. When a file is created or opened, a **file pointer** is also created and maintained for it. The file pointer holds the address of the next byte to be written to or read from (see figure 1). As data in the file is read or written, the file pointer is automatically moved. Therefore, successive read or write operations will transfer data sequentially (see figure 2).

Any part of a file may be directly accessed by positioning the file pointer to any location in the file using an OS-9 **system call**: "seek." You can access the "seek" system call through the various languages available for OS-9 or directly with the macro assembler command: I$SEEK (described in the "OS-9 OPERATING SYSTEM TECHNICAL MANUAL").

Reading up to the last byte of the file will cause the next read operation to return an end-of-file status (see figure 3). Trying to read past the end-of-file mark will cause an error. To expand a file, you simply write past the previous end of the file (see figure 4).

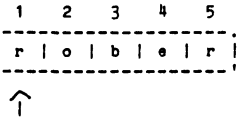

figure 1: When creating or opening a file, the file pointer is positioned to read from or write to the first component.

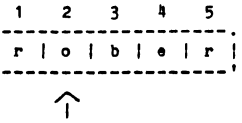

figure 2: After reading or writing the first component of a file, the file pointer will be pointed to the second component.

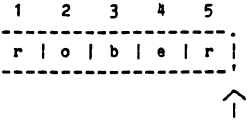

figure 3: The file pointer is pointed at the current end-of-file. Attempting another read operation will cause an error. Another write operation will increase the size of the file.

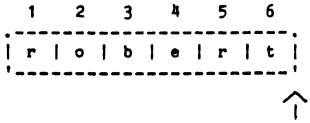

figure 4: The next write operation will add a new component to the file and moves the file pointer to the new end-of-file.

Because all OS-9 files have the same physical organization, file manipulation utilities can generally be used on any file (except directory files which are discussed separately) regardless of its logical usage. The main logical types of files used by OS-9 are:

1. text files
2. executable program module files
3. data files
4. directories

Text Files

Text files contain variable length lines of ASCII characters. Each line is terminated by a carriage return. Text files typically contain documentation, procedure files, program source code, etc. Text files can be built by any text editor or the BUILD utility.

Executable Program Module Files

Executable program modules are used to store programs generated by assemblers and compilers. Each file may contain one or more modules. Each module that is resident in memory has a standard OS-9 module format. See the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" for more information on modules.

Random Access Data Files

Random access data files are created and used primarily by high level languages such as Pascal and Basic09. The file is organized as an ordered sequence of records. Records do not have to be the same size. If each record has exactly the same length, it's beginning address within the file can be computed to allow records to be accessed in any order. OS-9 does not directly deal with records other than providing the basic file manipulation functions required by the high level languages that support random access records.

THE OS-9 FILE SYSTEM

OS-9 uses a tree-structured (or hierarchical) organization for its file system on mass storage devices such as disk systems (see figure 5). Each mass storage device has a master directory. This is called the **root directory.**

The root directory is created automatically when a new disk is formatted. The root directory contains the names of the files and the sub-directories on the disk. Every file is listed in a directory by name. Each file has a unique name within a directory.

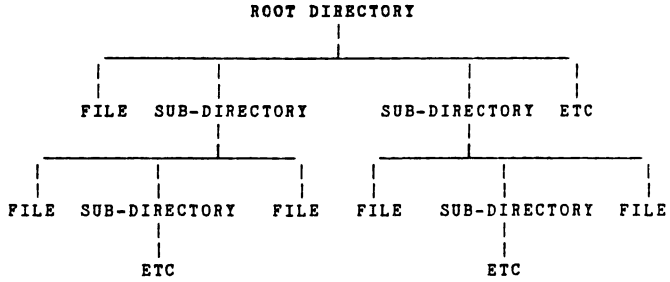

figure 5: the file system

An OS-9 directory can contain both files and sub-directories. Each sub-directory can contain more sub-directories and files. The only limit to this division is the amount of available space on a disk.

CURRENT DIRECTORIES

Two working directories are associated at all times with each user or process. These directories are called the current data directory and the current execution directory. A data directory is the directory in which text files are stored and created. An execution directory is where executable files are located, such as utilities and programs that you have created.

The current directory concept allows you to keep your files separate from the other users on the system. The word "current" is used because it is possible for you to move through the tree structure of the OS-9 file system to a different directory. This new directory would then become your "current" data or execution directory.

On a single user system, OS-9 will choose the root directory of your system disk as your initial current data directory. A directory called CMDS on the system disk will be chosen to be your initial current execution directory.

On a multiuser system, your current data and execution directories are established for you. As part of the login sequence, your initial directories will be determined by your password entry in the password file (see the Chapter 6 LOGIN command description for the contents of a password file entry). Usually, your data directory will have the name of your password entry, and your execution directory will be the CMDS directory.

On typical multiuser systems, all users have their own data directory, but share an execution directory.

The private data directory allows users to conveniently organize their own files (by project, function, user, etc.) without affecting other user's files.

The shared execution directory (CMDS) contains OS-9 utilities and other executable files. If all users had their own copy of all OS-9 commands, there would be a great deal of wasted disk space. It is possible for each user to have a private execution directory (apart from the CMDS directory). This will be described later.

ACCESSING FILES AND DIRECTORIES: THE PATHLIST

All files or directories in your current data directory can be accessed by simply specifying the name of the file or directory after the proper command. When only a file or directory name is given, OS-9 will not look outside your current data directory to find it.

You may access a file that is not in your current data directory, or run a program that is not in your current execution directory by changing your current directory or specifying a "pathlist" through the tree system for OS-9 to follow. Often it is very inconvenient to change directories because you may not want to leave your current directory, so a path is specified.

There are two types of pathlists:

1. full pathlists
2. relative pathlists

A full pathlist starts at the root directory and follows the pathlist down the tree structure to a specific file or directory (following the directory names in the list). A full pathlist must begin with a slash character (/). Each name in the pathlist is separated by a slash character.

A relative path starts at the current directory and follows down the tree structure to the specified file or directory. A relative pathlist does not begin with the slash character, but each name within the pathlist must be separated by a slash character.

Here are some examples:

/d1/Pascal/tests/futureval - is a full pathlist. It specifies a path from the root directory (/d1) through two subdirectories (pascal and tests) to the file futureval.

doc/letters/jim - is a relative pathlist. The path begins in the user's current directory through two sub-directories ("doc" and "letters") to the file "jim".

accounts - is a relative pathlist. No directories are searched other than the current directory.

On a command line, if a pathlist is not preceded by a utility name, OS-9 assumes you want to run a program. In the case of a single name on the command line, OS-9 searches the current execution directory for the specified name. If a pathlist is specified, OS-9 searches the location given. In either case, if the name that is found is not an executable program, OS-9 will try to process it as a procedure file (procedure files are discussed in Chapter 5).

BASIC FILE SYSTEM ORIENTED COMMANDS: A TUTORIAL

This section explains some of the OS-9 utility commands that manipulate the file system. The utilities include DIR, CHD, CHX, PD, BUILD, MAKDIR, LIST, COPY, DEL, DELDIR, and ATTR. This section has been designed to be a tutorial. All the examples given refer to the file system diagram in figure 6.

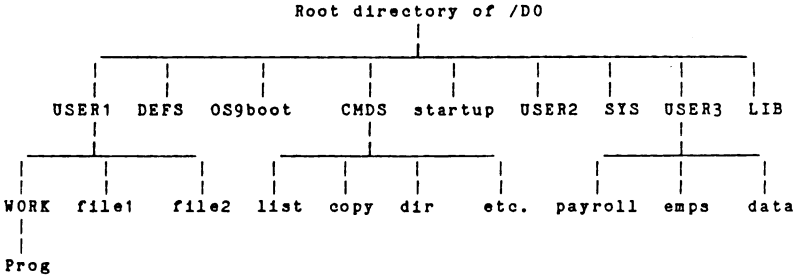

figure 6: Diagram of a Typical File System

DIR: Displaying the Contents of Directories

The DIR command displays the contents of directories. Typing DIR by itself will display the contents of your current data directory. Example:

```
$ dir

   directory of .  13:56:58
CMDS            DEFS            LIB             OS9boot         startup
SYS             USER1           USER2           USER3
```

The contents of the root directory are displayed (in this case, it is the directory of your system disk).

The file named "OS9boot" contains the system modules that are loaded into memory at system startup.

The file called "startup" is a procedure file that displays the welcoming message.

The "SYS" directory contains two files: a file named "Errmsg" that contains text for descriptions of error messages, and a file named "password" that contains a sample password file for timesharing systems.

The "DEFS" directory contains several files that contain symbolic definitions that are useful when writing assembly language routines.

The "LIB" directory contains relocatable library files.

"CMDS" is a directory that contains all the system commands (utilities) such DIR, LIST, SETIME, etc.

To display specific directories other than the current data directory, you must type in the pathlist (full or relative) and the name of the directory. To see the contents of the CMDS directory type:

```
dir cmds
```

DIR will now display the names of the files in the CMDS directory. The name "cmds" is the relative pathlist. You could type the full pathlist, "dir /d0/cmds", and get the same result. You can type "dir cmds" because CMDS is in your current directory.

DIR Options

The DIR utility has many options. These are fully listed in chapter 6. The "-e" option stands for "extended directory listing". Typing "dir -e" displays all files within the specified directory with their attributes. It also displays the size of the file and the sector in which it is stored.

The DIR options may be used in conjunction with each other. The "-r" option displays the contents of the specified directory and any files contained within its sub-directories. Typing "dir -er" displays all files within the current data directory, all files within its sub-directories, and gives an extended listing of their attributes, sizes, etc. To fully understand the DIR options, we recommend reading the DIR listing in chapter 6. Try each of the options and see what information is displayed.

CHD AND CHX: Moving Around in the File System

The CHD (change current data directory) and CHX (change current execution directory) commands allow the user to "travel" around the file system.

If you wish to move from your current data directory (the root directory) into the USER1 directory in the file system, you would type "chd" followed by the pathlist of USER1. Since your current data directory is the root directory (/d0), USER1 is the relative pathlist (a full pathlist would be /d0/USER1). You can type:

```
chd USER1
```

You are now in the directory called USER1. If you type DIR, the following information will be displayed:

```
   directory of . 14:04:32
file1          file2          WORK
```

If you want to see the files in the WORK directory inside USER1, you can type "dir WORK." Or you can change directories again by typing "chd WORK" and then after the new prompt typing "dir."

The CHX command allows a user to redefine an existing directory as a "personal execution directory". This may be important if you have programs that you do not want other people to execute. To use this command, type "chx," the pathlist (relative or full), and the name of the directory.

If your current data directory is USER1, and you want to make the WORK directory be your current execution directory, you would type "chx work". Now, when you type a command, WORK will be searched instead of the CMDS directory (which was your execution directory).

By typing "dir -x", you will display the contents of your current execution directory. Since this directory is now WORK, the following would be displayed:

```
   directory of . 14:04:32
Prog
```

If the file system becomes complex, it is possible to become "lost" and not know where the directory you are currently working in is located (in regards to the overall file system). The PD command will display the complete pathlist from the root to your current data directory.

For Example, (if your current data directory is USER1):

```
pd
/d0/USER1
```

If you forget which directory is your current execution directory, you can type "pd -x" to display the pathlist to the current execution directory.

Climbing Directory Trees

Sometimes it is useful to refer to the current directory (or a directory located higher in the tree-file structure) without typing in its name. In some cases the name may not be known. Special name substitutes are available:

.	refers to the current directory
..	refers to the next higher directory

NOTE: Using these name substitutes does not change the actual directory's name.

Any number of periods (.) may be used to access higher directories (one period per level).

Name substitutes can be used in conjunction with paths. They can be used as the first name in a path. Examples (Note: A parent directory is the directory that contains the current directory):

dir .	displays the contents of the current directory.
dir ..	displays the contents of the current directory's parent directory.
chd ..	changes the current directory to the parent directory.
chd ../user1/work	changes the current directory to "work" by giving a path from the parent directory.

MAKDIR AND BUILD: Creating New Directories and Files

New directories are created by using the MAKDIR command. For example, to create a directory called "NEWWORK," you would type:

```
makdir NEWWORK
```

There will now be a new entry in your current directory: NEWWORK. If you want the new directory to be created somewhere other than your current directory, you must specify a full pathlist. For example, "makdir /d0/user3/NEWWORK" creates the new directory in "USER3".

NOTE: By OS-9 convention, directory names are in upper case and file names are in lower case. This allows the user to easily distinguish directories from files. This is only a recommendation for easy use, so you may develop your own style if you like.

The BUILD command is used to create short text files. Scred, the screen editor for OS-9/68000 (See the "OS-9/68000 SCRED USER'S MANUAL") or the line editor, EDT, should be used for editing larger files (EDT is fully detailed in Chapter 5).

To use the build command, type "BUILD", followed by the file you want to create. BUILD then responds with the prompt: "?". This tells you that BUILD is waiting for input. To terminate BUILD, type a carriage return as the only thing on the line. For example:

```
$ build test
? Some programmer's have been known to
? howl at full moons.
?

$
```

If you execute the DIR command, you will see that your directory has a new entry called "test".

Directory Guidelines

Some important characteristics relating to the use of directory files are as follows:

1. Directories have the same ownership and attributes as regular files. These attributes are explained in detail later in this chapter.

2. Each file name within a directory must be unique.

3. Files can have identical names, as long as they are stored in different directories.

4. All files are stored on the same device (i.e. disks, tape,) as the directory in which they are listed.

5. The only limit to the number of files that can be stored in a directory is the amount of free disk space.

Rules for Constructing File Names

Any file name can contain from 1 to 28 upper or lower case letters, numbers or special characters (as listed below). While the file name may begin with any of the following characters or digits, the file name must contain at least one letter or number. Within these limitations, it can contain any combination of the following:

-upper case letter: A - Z

-lower case letter: a - z

-decimal digits: 0 - 9

-underscore: _

-period: .

-dollar sign: $

File names may not contain spaces. OS-9 does not distinguish upper case letters from lower case letters. The names "FRED" and "fred" would be considered the same name (just a reminder: using uppercase directory names is a simple convention that allows you to easily determine files from directories).

Here are some examples of legal names:

raw.data.2	project_review_backup
X6809	$Ship.Dir
...c	12345

Here are some examples of illegal names:

Max*min (* is not a legal character)

open orders (Name can not contain a space)

this.name.obviously.has.more.than.28.characters (too long)

NOTE: File names that start with a period are not displayed by the DIR command unless the "-a" option is used. This allows a user to "hide" files within a directory.

LIST and COPY: Listing and Copying Files

LIST is used to examine the contents of files. It displays the lines of text on your terminal screen. To examine a file, type "list" followed by the name of the file you want to list. For example:

```
$ list test
Some programmer's have been known to
howl at full moons.
```

An important fact to remember: you can not list a directory. If you type "list USER2", OS-9 will respond with an error message and an error number: "list: can not open "USER2". Error# 000:214". This means you did not have the correct permission to access "USER2". Since "USER2" is a directory, you can not list the contents of it.

If you try to list the contents of a file in the CMDS directory, you will see random data displayed to your screen. LIST displays text files, and all the files in CMDS are executable object code files. Try it and see.

The COPY command is used to make a duplicate of a file. To copy a file, type "copy", followed by the name of the file to be copied, followed by the name of the duplicate file (the copy command actually creates the duplicate file). Example:

```
$ copy test newtest
```

Now, LIST "newtest". It should be an exact copy of test.

DEL AND DELDIR: Deleting Files and Directories

DEL and DELDIR are used to eliminate unwanted files and directories. If a file is no longer needed, deleting the file will free disk space.

To delete a file, type "del", followed by the name of the file that you want deleted. For example, let's delete the file ("test") you created with BUILD.

```
del test
```

If you execute the DIR command you will see that test is no longer displayed.

Deleting a directory is a little different.

DELDIR is the command used to delete directories. DELDIR first deletes all the files and directories in the given directory, and then, if no errors have occurred, will finally delete the directory name. For example:

```
$ deldir user2

Deleting directory: user2
List directory, delete directory, or quit ? (l/d/q)
```

At the prompt, type "l" to list the contents of the directory, "d" to delete the directory, or "q" to quit (and not delete anything).

Just a reminder: never delete a file or directory unless you are sure you don't need it.

FILES: THE OWNER AND THE PUBLIC

Before discussing the file attribute system, two terms must be understood: the **owner** and the **public**. The OS-9 user's community is divided into two classes: the owner and the public (other users).

OS-9 automatically stores a group and user number (the "group.user" ID) with each file or directory. This "group.user" ID is determined by the password file entry. You receive a group and user ID when you gain access to the system through the LOGIN command.

Any user that has the same user ID as the file is considered the owner. A person with the same group number as the owner of a file can access the file in the same way as the owner. This allows people that work on the same project or work in the same department, to be given a common group identification: the group number.

The File Security System

File use and security are based on files **attributes**. Each file can have up to seven possible attributes (directories can have eight). They are displayed in an eight character line listing.

The term **permission** is used when one of the eight possible attribute characters is set. Permission determines who can access a file or directory and how it can be used. If a permission is not valid for the file or directory being examined there will be a "-" in its position.

Here is an attribute listing for a directory in which all permissions are valid:

```
dsewrewr
```

By convention, attributes are read from right to left. They are:

Owner Read: the owner can read the file. When off, this attribute can be used to prevent executable files from being read as text files.

Owner Write: the owner can write to the file. When off, this attribute can be used to protect files from accidentally being deleted or modified.

Owner Execute: the owner can execute the file.

Public Read: the public can read the file.

Public Write: the public can write the file.

Public Execute: the public can execute the file.

Single user: when set indicates only one user at a time can access the file.

Directory: when set indicates a directory.

NOTE: Prior editions of this manual have referred to the single user attribute as the sharable attribute. When set this attribute is not sharable, hence the confusion. We have found "single user" a more descriptive term.

The attributes are examined and changed by using the **ATTR** command. Type "attr", followed by the name of a file. For example:

```
$ attr newtest
------wr
```

The file called "newtest" has the permissions set for owner reading and owner writing. Access to this file by anyone other than the owner will be denied.

When a directory is created it will initially have all the permissions set except the single user permission. For example:

```
$ attr user1
d-ewrewr
```

When a file is created using the BUILD command or the line editor (EDT), only the owner read, owner write, and public read permissions are set.

When using Scred to create files, only the owner read and write permissions are set.

If ATTR is used with a list of one or more attribute abbreviations, the file's attributes will be changed accordingly (if you have the proper write permission to access the file). The attribute abbreviations do not have to be listed in any particular order. The attribute abbreviations are:

r	Owner read
w	Owner write
e	Owner execute
pr	Public read
pw	Public write
pe	Public execute
s	Single user
d	Directory

The following command enables public read and write permission and removes execution permission for both the owner and the public (the "n" preceding the attribute means no permission granted):

```
$ attr newtest -pw -pr -ne -npe
```

If you are the owner of a file, you can change the access permissions no matter what the permissions indicate. Thus, the owner always has the right to delete a file, change the user privileges, etc. Users in the same group have the same permissions as the owner.

The directory attribute behaves somewhat differently than any of the other attributes. It could be quite dangerous to be able to change directory files to normal files or a normal file to a directory.

The ATTR utility can not be used to turn the directory (d) attribute on (only MAKDIR can). Furthermore, it can only be used to turn the directory attribute off if the directory is empty.

All files in a directory can have their attributes examined by using the DIR command with the "-e" option.

end of chapter 4

USER NOTES

THE FUNCTION OF THE SHELL

The SHELL is the OS-9 command interpreter program that provides the interface between the user and the internal functions of the operating system. The SHELL is automatically entered following the system startup, or after logging on to a timesharing terminal. You will know when the SHELL is waiting for input because it displays the prompt:

```
$
```

The prompt indicates that the SHELL is active and waiting for a command from your keyboard. You can now respond by typing a command line followed by a carriage return.

Some SHELL options are automatically (default) turned on following startup or logging on. These and other options can be turned off or on by one of two methods.

The first method is to type the option on the command line or after the command, "shell". For example:

`$ -np`	turns off the SHELL prompt.
`$ shell -np`	creates a new shell that will not prompt. When the new SHELL is exited, the original SHELL will prompt.

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

`$ set np`	turns off the SHELL prompt.
`$ shell set np`	creates a new shell that will not prompt. When the new SHELL is exited, the original SHELL will prompt.

As you can see, the two methods accomplish the exact same function. They are both provided for your convenience. Use the method that is clearer for you.

The SHELL options available are:

-e=<file>	Prints error messages from <file>. If no file is specified, the default file used is /dd/sys/errors. Without the "-e" option, SHELL will print only error numbers with a brief message description.
-ne	Prints no error messages (default).
-t	Echoes input lines.
-nt	Does not echo input lines (default).
-p	Displays prompt (default prompt: "$ ").
-p=<string>	Sets current SHELL prompt equal to <string>.
-np	Does not display prompt.
-x	Aborts process upon error (default).
-nx	Does not abort on error.

A MORE DETAILED DESCRIPTION OF COMMAND LINE PROCESSING

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

Keyword:	A program, procedure file or built-in command name.
Object:	A file or directory name.
Parameters:	Values, variables, constants, etc. to be passed to the program.
Execution modifiers:	These modify a program's execution by redirecting I/O or changing the priority or memory allocation of a process.
Separators:	These specify (to SHELL) how programs are to be executed when placed on the same command line: sequentially or concurrently.

Only the keyword needs to be present for SHELL to process a command line. The other parts are optional. After the keyword has been identified, the SHELL processes execution modifiers and separators (if any). Any other text not yet processed is assumed to be objects or parameters and is passed to the program called.

The keyword must be the first word in the command line. After it has been scanned, the SHELL first checks if it is a "built-in" command. If it is, it is immediately executed. Commands that are part or the SHELL (built-in commands) will be described later in this chapter.

If the keyword is not a built-in command, the SHELL assumes it is a program name and attempts to locate and execute it. It then waits until the program eventually terminates, at which time it reports any errors returned. The SHELL will get another input line (if any), and the process is repeated. This sequence is repeated until an end-of-file condition is detected by the SHELL, which causes the SHELL to terminate its own execution.

When the SHELL processes the command line, it searches for the command in the following sequence:

1. The SHELL first chacks memory to see if the program has already been loaded. If it has, there is no need to load another copy. SHELL then calls that program to be executed.

2. The user's current execution directory is searched. An execution directory is where executable programs are kept. If the program is found, it is loaded and executed.

3. The user's current data directory is searched. If the specified file is found, it is processed as a procedure file. Procedure files are assumed to contain one or more command lines that are processed by the SHELL in the same manner as if they had been typed in manually.

Here is a sample command line which calls a program:

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

In this example:

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

ADVANCED FEATURES OF THE SHELL

In addition to basic command line processing, the SHELL has functions that facilitate:

- Memory Allocation
- I/O redirection (including filters)
- Process Priority
- Wildcard Pattern Matching
- Multitasking (concurrent execution)
- Procedure File Execution (background processing)
- Execution Control (built-in commands)

These functions are accessed through the use of execution modifiers, separators and wild cards. There are virtually no limits to the combination of ways these capabilities can be used.

Characters which comprise execution modifiers, separators, and wild cards are stripped from the part(s) of the command line passed to the program as parameters. These characters can not be used inside parameters to the program unless they are contained in quotes:

modifiers:	#	(alternate memory size)
	^	(process priority)
	>	(redirect output)
	<	(redirect input)
	>>	(redirect error output)
separators:	;	(sequential execution)
	&	(concurrent execution)
	!	(pipe construction)
wild cards:	*	(matches any character)
	?	(matches a single character)

EXECUTION MODIFIERS

Execution modifiers are processed by the SHELL before the program is run. If an error is detected in any of the modifiers, the run will be aborted and the error reported.

Alternate Memory Size Modifier

Every executable program is converted to 68000 machine language for storage. As part of the conversion process, a module header is created for a program. A module header is part of all executable programs and holds the program's name, size, memory requirements, etc. (a complete explanation of module headers is available in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL").

When an executable program is processed by the SHELL, it is allocated the minimum amount of working memory specified in the program's module header. It is sometimes desirable to increase the default memory size. Memory can be assigned in 1024 byte (1k) increments using the modifier "#", followed by a number of allocated kilobytes: "#10k" or #10".

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

I/O Redirection Modifiers

Redirection modifiers are used to redirect the program's "standard I/O paths" to alternate files or devices. Programs do not normally use specific file or device names. It is therefore fairly simple to "redirect" the I/O to any file or device without altering the program itself.

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

STANDARD INPUT PATH: This path normally passes data from a terminal's keyboard to a program.

STANDARD OUTPUT PATH: This path normally passes output data from a program to a terminal's display.

STANDARD ERROR OUTPUT PATH: This path is used to output routine status messages such as prompts and errors to the terminal's display (defaults to the same device as the standard output path). NOTE: The name "error output" is somewhat misleading. Many other types of messages (besides errors) are sent on this path.

When a new process is created by using an existing process (a "parent" process), the new process inherits the parent process' standard I/O paths. When the SHELL creates a new process, it inherits the SHELL's standard I/O paths. Upon startup or logging in, the SHELL's standard input is the terminal keyboard; the standard output and error output is the terminal's display.

When a redirection modifier is used on a SHELL command line, the SHELL will open the corresponding paths and pass them to the new process as its standard I/O paths. There are three redirection modifiers:

<	Redirect the standard input path
>	Redirect the standard output path
>>	Redirect the standard error output path

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

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

TERM	- Primary system terminal
t1, t2, etc.	- Other serial terminals
p	- Parallel printer
p1	- Serial printer
dd	- default disk drive (duplicates physical device name such as d0, h0, etc.)
d0	- Floppy disk drive unit 0
d1, d2, etc.	- Other floppy disk drives
h0, h1, etc.	- Hard disk drives

Device names may only be used as the first name of a pathlist, and must be preceded by a slash character, "/", to indicate that the name is an I/O device. If the device is not a mass storage multifile device like a disk drive, the device name must be the only name in the path. This restriction is true for devices like terminals and printers.

For example, the standard output of LIST can be redirected to write to the system printer instead of the terminal:

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

Files referenced by I/O redirection modifiers are automatically opened or created, and closed (as appropriate) by the SHELL. Here is another example, the output of the DIR command is redirected to the path "/D1/savelisting":

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

If the LIST command is used on the path "/d1/savelisting", output from the DIR command will be displayed as shown below:

```
$ List /d1/savelisting

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

Redirection modifiers can be used before and/or after the program's parameters, but each modifier can only be used once. Redirection modifiers can be used together to cause more than one of the standard paths to be redirected. For example, "shell <>>>/t1" causes all three standard paths to be redirected to /t1.

The "+" and "-" characters can be used with redirection modifiers. The operation ">-" means redirect output to a file. If the file already exists, the operation will rewrite it. The ">+" operation means add the output to the end of the file. Examples:

```
dir -e >-dirfile              rewrites "dirfile" with output
                              from the execution directory.

list newnames >+oldnames      adds the listing of "newnames"
                              to the end of "oldnames."
```

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

Process Priority Modifier

On multiuser systems, or when multitasking, many processes seem to be executed at the same time. Actually, OS-9 uses a scheduling algorithm to allocate execution time to active processes.

All active processes are sorted into a queue based on the "age" of the process. The "age" is a number between 0 and 65536 based on how long a process has waited for execution and its initial priority. Every time a new active process is submitted for execution, all earlier processes' ages are incremented. Each process is given an initial priority that is specified in the password file. This priority is set by the system manager.

If you have a program that you want the system to give higher priority, the "^" modifier is used. By specifying a higher priority, a process will be placed higher in the execution queue. For example:

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

In this example, the process format is given the assigned priority of 255, regardless of the initial priority listed in the password file. You may also specify lower priority by assigning a lower number than is specified.

Wild Card Matching

The SHELL uses some alternate ways to identify file and directory names. The SHELL accepts wild cards in the command line. The two wild card characters are "*" and "?".

"*" matches any group of zero or more characters. "?" matches any single character. SHELL searches the current data directory or the directory given in a path for matching file names.

For the following example commands, we will use a directory that contains the following files:

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

$ list log*
```

This command will list the contents of "login.names", "logistics" and "logs". The pattern "log*" will match all file names beginning with "log" followed by zero or more characters.

More examples of pattern matching using the "*" character are:

del s* - deletes all files beginning with "s":
"Shellfacts", "setime.c" and "sizes".

del * - deletes every file in the directory "FILES".

The "?" will match any single character in the position where the wild card character is located. The following commands demonstrate the function of "?".

del form? - will delete the file "forms" but not "form".

list s???? - will list the contents of "sizes" but not "setime.c" or "Shellfacts".

In both examples, the SHELL will search for five character names only.

SHELL will only attempt to expand a character string (that includes a wild card) that could be a pathlist. NOTE: double quotes around a character string causes SHELL to disregard any wildcards that may be contained within the quotes:

grep "*p" myfile - this command will cause SHELL to search the file "myfile" for the string "*p". The double quotes around "*p" will cause SHELL to overlook the wild card character.

WARNING: Care must be practiced with the use of wild cards with utilities such as DEL. Wild cards SHOULD NOT be used with the "-x" or "-z" options of most commands.

COMMAND SEPARATORS

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

Sequential execution of programs may be caused by separating the programs (and parameters) by a ";". Concurrent execution can be caused by separating the programs with a "&".

Sequential Execution

Programs are executed sequentially (one after another) when each is entered on a separate line. All programs executed sequentially are individual processes created by the SHELL. After initiating execution of a program to be executed sequentially, the SHELL enters the "wait" state until the program it created terminates.

More than one program can be specified on a single SHELL command line (for sequential execution) by separating each <program name> <parameters> from the next one with a ";" character. For example:

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

This command line will first execute the COPY command and then the DIR command. The command line will execute exactly as the following command lines (unless an error occurs):

```
$ copy myfile /D1/newfile

$ dir >/p
```

If an error is returned by any program, subsequent commands on the same line are not executed (regardless of the state of the "x" option). In all other cases, ";" and "carriage return" are identical separators.

Here are some more examples:

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

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

Concurrent Execution (Multitasking)

Programs may be executed concurrently using the "&" separator. This allows programs to run simultaneously with other programs, including the SHELL. The SHELL does not wait to complete a process before processing the next command. Concurrent execution is how a "background" program is started.

Multitasking is accomplished by means of the concurrent execution separator. The number of programs that can run at the same time is not fixed: it depends upon the amount of free memory in the system and the memory requirements of the specific programs.

Here is an example:

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

The DIR, LIST, and COPY programs will run concurrently, because they were separated by "&". The DEL program will not run until the COPY program has terminated because sequential execution (";") was specified.

By adding the "&" character to the end of a command line (regardless of the type of execution specified), SHELL will immediately return command to the keyboard, display the "$" prompt and wait for a new command. This frees the user from waiting for a process or sequence of processes to terminate.

You can display a "status summary" of all processes you have created by using the **PROCS** command.

Pipes and Filters

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

A pipe is simply a way to connect the output of a process to the input of another process, so the two run as a sequence of processes: a pipeline. Pipes are one of the primary means by which data is transferred from process to process (interprocess communications). Pipes are first-in, first-out buffers that behave like mass-storage files.

Pipelines are created by the SHELL when an input line having one or more "!" separators is processed. For each "!", the standard output or the program named to the left of the "!" is redirected by a pipe to the standard input of the program named to the right of the "!".

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

For example:

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

This example will cause the standard output of DIR to be piped to the standard input of PR (it would normally be displayed on the terminal screen). PR will read the output of DIR (PR reads standard input by default) and display the result.

Individual pipes are created for each "!" present. For example:

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

In this example, the program "update" has its input redirected from a path called "master_file". Update's standard output becomes the standard input for the program "sort". Sort's output, in turn, becomes the standard input for the program "write_report", which has its standard output redirected to the printer.

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

Some of the most useful applications of pipelines are character set conversion, data compression/decompression and text file formatting. Programs which are designed to process data as components of a pipeline are often called "**filters**". The TEE command is a useful filter. It uses pipes to allow data to be simultaneously "broadcast" from a single input path to several output paths.

COMMAND GROUPING

Sections of SHELL input lines can be enclosed in parentheses which permits modifiers and separators to be applied to an entire set of programs. The SHELL processes them by calling itself recursively (as a new process) to execute the enclosed program list. For example the following commands give the same result:

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

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

There is one subtle difference however: the printer is "kept" continuously in the first example. In the second case, another user could "steal" the printer in between the DIR commands.

Command grouping can be used to cause a group of programs to be executed sequentially, but also concurrently with respect to the SHELL that initiated them. For example:

```
$ (del file1; del file2; del file3)&
```

This command will begin to sequentially delete file1, file2, and file3. At the same time, a "$" prompt will appear, indicating that SHELL is waiting for a new command.

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

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

This command line will output the "dir" listings of CMDS and SYS (in that order) through a pipeline to the program "makeuppercase". The total output from "makeuppercase" will then be piped to "transmit".

BUILT-IN SHELL COMMANDS AND OPTIONS

When processing input lines, the SHELL looks for several special names of commands or option switches that are built-in to the SHELL. These commands are executed without loading a program and creating a new process. They can be used at the beginning of a line, or following the command separator (";" or "&").

The built-in commands and their functions are:

chd <path>	changes the current data directory to the directory specified by the path.
chx <path>	changes the current execution directory to the directory specified by the path.
ex <name>	directly executes the program named. This replaces the SHELL process with a new execution module in its place.
w	waits for any process to terminate.
* text	indicates a comment: the "text" is not processed.
kill <proc ID>	aborts the process specified.
set <options>	sets options for SHELL.
setenv <eparam> <evalue>	sets a value for the specified environment parameter.
setpr <proc ID> <priority>	changes process' priority.
unsetenv <eparam>	clears the value of the specified environment parameter.

NOTE: Refer to the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" for a description of process ids and process priorities.

SHELL PROCEDURE FILES

A procedure file is a text file that contains one or more command lines that are identical to command lines that are manually entered from the keyboard.

A simple procedure file could consist of the DIR command on one line and the DATE command on another. When the name of this procedure file is typed on the command line, DIR would be run followed by DATE.

Procedure files have a number of valuable applications. It can eliminate repetitive manual entry of commonly used sequences of commands. It can allow the computer to execute a lengthy series of programs "in the background" while the computer is unattended or while the user is running other programs "in the foreground".

MULTIPLE SHELLS

The SHELL can be simultaneously executed by more than one process at a time. It does this by creating other SHELLs.

The SHELL can create another SHELL by various means. Here are two simple ways to use the SHELL to create another SHELL:

```
$ SHELL <procfile

$ procfile
```

The first example redirects the standard input path to a procedure file, by means of the "SHELL" command. The new "incarnation" of the SHELL can automatically accept and execute command lines from a procedure file instead of a terminal keyboard. This technique is sometimes called "batch" processing.

In the second example, SHELL will start the new SHELL by simply processing the procedure file. Both examples do basically the same thing: execute the commands of the file "procfile".

If the <program name> specified on a SHELL command line can not be found in memory or in the execution directory, SHELL will search the current data directory for a file with the desired name. If one is found, SHELL will automatically execute it as a procedure file.

In addition to redirecting the SHELL's standard input to a procedure file, the standard output and standard error output can be redirected to a printer or to another file which can record output for later review. This can also eliminate the sometimes annoying output of SHELL messages to your terminal at random times.

To run the procedure file in a "background" mode you simply add the ampersand operator:

```
$ procfile&
```

OS-9 does not have any limit on the number of procedure files that can be simultaneously executed as long as there is memory available. Also, the procedure files themselves can cause sequential or concurrent execution of additional procedure files.

Perhaps the most useful application of multiple SHELLs is OS-9's ability to move around the file system by creating new SHELLs. To demonstrate this application we will use the directory system in figure 7.

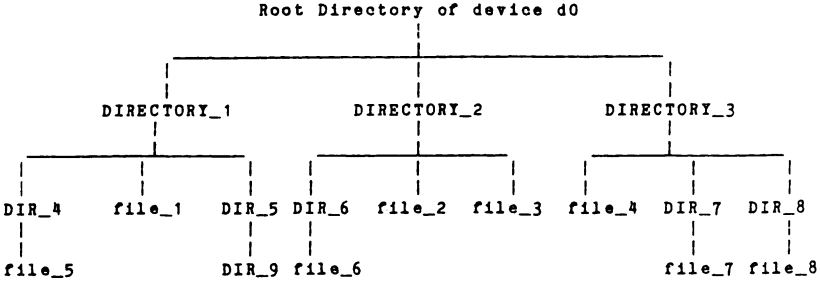

figure 7: an example directory

If your current data directory is "DIR_9", and you needed to work on "file_8", you would normally change directories to "DIR_8" and access the file. To do this, you would type:

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

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

Instead, you can create a SHELL and change directories:

```
$ (chd /d0/DIRECTORY_3/DIR_8)
```

Now you are in DIR_8, but you can return to DIR_9 by simply hitting the "escape" (Esc) key. By this method, you may use any directory as a base directory and "fork" a SHELL out to any other directory.

You may continue to imbed as many SHELLs as you like. Each time you "escape" you will be taken to the previous SHELL. In this fashion you could conceivably "escape" from "DIRECTORY_2" to "DIR_8" to "DIR_6" to "DIR_9".

REMINDER: Because of the nature of jumping from SHELL to SHELL, it is easy to get "lost". The "Pd" command will display a complete pathlist from the root directory to your current data directory.

The "PROCS" command is also useful in determining the number of SHELLs that are imbedded. It will show you a list of active processes currently running on the system (for a full description, see chapter 6). Try it.

We encourage experimentation with the multiple SHELL aspects of OS-9 to fully utilize the system.

SETTING UP A TIMESHARING SYSTEM STARTUP PROCEDURE FILE

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

Usually one TSMON command program is started up concurrently for each terminal in the system. TSMON is a special program which monitors a terminal for activity.

Here is a sample procedure file for a 4-terminal timesharing system having terminals named "TERM", "T1", "T2" and "T3":

```
* system startup procedure file
echo Please Enter the Date and Time
setime </term
tsmon /t1&
tsmon /t2&
tsmon /t3&
```

NOTE: This LOGIN procedure will not work until a password file called "/d0/SYS/Password" has been created.

The previous example deserves special attention. Note that the SETIME command has its input redirected from the system console "term". This is necessary because it would otherwise attempt to read the time information from its current standard input path (which is the procedure file and not the keyboard).

ERROR REPORTING

Many programs (including the SHELL) use OS-9's standard error reporting function. This displays a brief description of the error and an error number on the error output path. The standard error codes are listed in Appendix A.

If a longer description of errors is desired, set the "-e" SHELL option. This will print error massages from "/dd/sys/errors" on standard output.

RUNNING COMPILED INTERMEDIATE CODE PROGRAMS

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

For example, if you wanted to run a BASIC09 I-code module called "adventure", you could type either of the two commands given below (they accomplish exactly the same thing):

```
$ BASIC09 adventure

$ adventure
```

end of chapter 5

USER NOTES

SYSTEM COMMAND DESCRIPTIONS

Chapter 6 contains descriptions and examples of each of the OS-9 command programs. While these programs are generally called from the SHELL, they can also be called from most other OS-9 programs (BASIC09, the Debugger, Scred, etc.).

At the time of this edition, OS-9 supplies 59 utility command programs. The range they encompass is wide: from commonly used functions (DIR, CHD, COPY, etc.) to advanced system management tools (DCHECK, INIZ, etc.).

For quick reference purposes, the format of the information on the utilities has been standardized. Each command has a section concerning syntax, function, execution, options (if any), examples and any special usages. Many have been cross referenced to other utility commands.

The utilities have been broken down into three catagories. This should aid the user for quick reference. The three catagories are as follows:

1. **BASIC UTILITY COMMANDS:** Every user should become familiar with these utilities. Many of them have been discussed in the earlier chapters because of their importance.

ATTR	BACKUP	BUILD	CHD	CHX	COPY
DATE	DEL	DELDIR	DIR	ECHO	EDT
FORMAT	FREE	KILL	LIST	MAKDIR	MERGE
MFREE	PD	PR	PROCS	RENAME	SET
SETIME	SHELL	TMODE			

2. **PROGRAMMER UTILITY COMMANDS:** These utility programs are of extreme help to a programmer of intermediate or advanced skills. They allow greater exploration of the timesharing environment of OS-9 and more dynamic file manipulation:

BINEX	CFP	CMP	CODE	COMPRESS	COUNT
DEBUG	DSAVE	DUMP	EX	EXBIN	EXPAND
GREP	LOAD	MAKE	PRINTENV	QSORT	SAVE
SETENV	TEE	TOUCH	TR	UNSETENV	

3. **SYSTEM MANAGEMENT UTILITY COMMANDS:** These commands will be used primarily by system managers and advanced assembly language programmers. Beginning level programmers will almost never have the need to use these commands:

DCHECK	DEINIZ	FIXMOD	IDENT	INIZ	LINK
LOGIN	MDIR	OS9GEN	SETPR	SLEEP	TSMON
UNLINK	XMODE				

FORMAL SINTAX NOTATION

Each command section includes a syntactical description of the command line. These are symbolic descriptions that use the following notation:

[ ]	=	Enclosed items are optional.
{ }	=	Enclosed items may be used 0, 1, or many times.
<path>	=	A legal pathlist.
<devname>	=	A legal device name.
<modname>	=	A legal memory module name.
<procID>	=	A process number.
<opts>	=	One or more of the options specified in the command description.
<arglist>	=	A list of arguments.
<text>	=	A character string ended by end-of-line.
<expr>	=	An expression or set of characters. This may include metacharacters as specified in the GREP and TR utility command descriptions.
<num>	=	A decimal number (unless otherwise specified).
<file>	=	An existing file.
<addr>	=	A hexadecimal number signifying an address in memory.

IMPORTANT NOTES

The utility command syntax specified in the command sections does not include the SHELL's built-in options (alternate memory size, I/O redirection, piping, etc.). The SHELL will filter out its options from the command line before processing the program being called.

The "=" used in many utility options is generally optional. This is true for the "k" used in the alternate memory size option also. For example, "-b=256k" may be written as "-b256", "-b256k", or "-b=256".

Any utilities that use the "-z" option expect one file name to be input per line.

CIO, the utility trap handler, must be in the execution directory or pre-loaded into memory. By using special I/O techniques, CIO allows the utilities to be much smaller. Unpredictable results could occur if CIO is missing. CIO is typically loaded at startup.

ATTR BASIC UTILITY COMMAND ATTR

Change file security attributes

SYNTAX: ATTR [<opts>] {<path> [<opts>] <permission abbreviations>}

FUNCTION: ATTR is used to examine or change the security permissions of a file.

EXECUTION: Type ATTR, followed by the pathlist for the file(s) whose security permissions are to be changed or examined. Then type a list of permissions which are to be turned on or off.

A permission is turned on by giving its abbreviation preceded by a minus sign: "-". It is turned off by preceding its abbreviation with a minus sign followed by an "n": "-n". Permissions not explicitly named are not affected. If no permissions are given, the current file attributes will be printed.

You can not change or examine the attributes of a file which you do not own (except for user zero, who can change the attributes of any file in the system).

The file permission abbreviations are:

d = Directory file

s = Single user file ("s" denotes a nonsharable file)

r = Read permission to owner

w = Write permission to owner

e = Execute permission to owner

pr = Read permission to public

pw = Write permission to public

pe = Execute permission to public

SPECIAL USAGE: The ATTR command may be used to change a directory file to a non-directory file if all entries have been deleted from it. DELDIR may also be used to delete directory files. You can not change a non-directory file to a directory file with this command (see MAKDIR).

OPTIONS:

-?	Displays the usage of ATTR.
-a	Suppresses the printing of attributes.
-x	Searches for the specified file in the execution directory; the file must have execute permission to be found using -x.
-z	Reads the file names from standard input.
-z=<file>	Reads the file names from <file>.

EXAMPLES:

$ attr myfile -npr -npw	Turns off public read and public write.
$ attr myfile -rweprpwpe	Turns on (both public and owner) read, write, and execute permissions.
$ dir -u ! attr -npwpr -z	Pipes an unformatted directory listing into ATTR and turns off public write and turns on public read for all files with correct ownership in the directory.
$ attr datalog	Displays current attributes.
$ attr -z=file1	Displays the attributes of those file names read from "file1".
$ attr -z	Displays the attributes of those file names read from standard input.

BACKUP BASIC UTILITY COMMAND BACKUP

Make a backup copy of a disk

SYNTAX: BACKUP [<opts>] [<devname1> [<devname2>]] [<opts>]

FUNCTION: BACKUP is used to physically copy all data from one device to another.

EXECUTION: A physical copy is performed sector by sector without regard to file structures. In almost all cases, the devices specified must have the exact same format (size, density, etc.) and must not have defective sectors.

SINGLE DRIVE BACKUP: A single drive BACKUP requires exchanging disks in and out of the disk drive.

NOTE: Before backing up a disk, it is suggested to "write protect" the source disk (with a "write protect" tab) to prevent accidently confusing the source disk and the destination disk during exchanges.

To begin the BACKUP procedure, put the source disk in the drive and type "backup". The system will ask if you are ready to BACKUP. Type "y" if you are ready.

Initially, BACKUP will read a portion of the source disk (the disk to be backed up) into memory. BACKUP will then ask you to exchange disks. Remove the source disk from the drive, and insert the destination disk (the disk being copied to) into the drive. BACKUP will write the previous data that it had stored onto this disk. When it is finished, it will request an exchange again, placing the source disk back in the drive. This will continue until all of the data on the disk has been copied.

The "-b" option increases the amount of memory that BACKUP uses. This will decrease the number of disk exchanges required.

TWO DRIVE BACKUP: On a two drive system, if both device names are omitted, the names "/d0" and "/d1" are assumed. If the second device name is omitted, a single unit backup will be performed on the drive specified.

To begin the BACKUP procedure, put the source disk in the source drive (/d0 by default) and the destination disk in the destination drive (/d1 by default). Type "backup", the name of the source drive, and the name of the destination drive. The system will ask if you are ready to BACKUP. You type "y" if you are ready. If no error occurs, the BACKUP procedure is complete.

(continued)

ERRORS: The BACKUP procedure includes two passes. The first reads a portion of the source disk into a buffer in memory and then writes it to the destination disk. The second pass verifies that the data was copied correctly.

If an error occurs on the first pass, there is something wrong with the source disk or the its drive. If an error occurs in the second pass, the problem is with the destination disk. If BACKUP fails repeatedly on the second pass, reformat the destination disk and try to backup again.

OPTIONS:

-?	Displays the usage of BACKUP.
-b=<num>k	Allocates <num>k of memory for the BACKUP buffer to use. BACKUP uses a 4k buffer (by default). BACKUP will run faster if more memory is used.
-r	Causes BACKUP to continue if a read error occurs.
-v	Will stop BACKUP from making a verification pass.

EXAMPLES:

```
$ backup /D2 /D3        Backup /d2 to /d3

$ backup -v             Backup /d0 to /d1 with no verify
```

BINEX / EXBIN PROGRAMMER UTILITY COMMAND BINEX / EXBIN

Convert Binary To S-Record File
Convert S-Record To Binary File

SYNTAX: BINEX [<opts>] [<path1>] [<path2>] [<opts>]
EXBIN [<opts>] [<path1>] [<path2>] [<opts>]

FUNCTION: BINEX converts binary files to an S-record file. EXBIN reverses this process.

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

EXECUTION: BINEX converts <path1>, an OS-9 binary file, to a new file named <path2> in S-Record format. S-Records have a header record to store the program name (for informational purposes) and each data record has an absolute memory address which is meaningless to OS-9 (because OS-9 uses position-independent-code).

BINEX will currently generate the following S-record types:

S1 records	use a two byte address field.
S2 records	use a three byte address field.
S3 records	use a four byte address field.
S7 records	terminate blocks of S3 records.
S8 records	terminate blocks of S2 records.
S9 records	terminate blocks of S1 records.

To specify which type of S-record file is to be generated, use the "-s=<num>" option. <num> = 1, 2, etc., corresponding to S1, S2, etc.

EXBIN is the inverse operation. <path1> is assumed to be an S-Record format text file which EXBIN converts to pure binary form in a new file (<path2>). The load addresses of each data record must describe contiguous data in ascending order. EXBIN does not generate or check for the proper OS-9 module headers or CRC check value required to actually load the binary file. The IDENT command can be used to check the validity of the modules if they are to be loaded or run. EXBIN will convert any of the S-record types mentioned above.

(continued)

Using either command, if both paths are omitted, standard input and output are assumed. If the second path is omitted, standard output is assumed.

OPTIONS:

-?	Displays the usage of BINEX/EXBIN.
-a=<num>	Specifies the load address (in hex) - BINEX only.
-s=<num>	Specifies which type of S-record format is to be generated - BINEX only.
-x	BINEX searches for <path1> in the execution directory - BINEX only.

EXAMPLES:

$ binex scanner.S1 >/T1	downloads a program to T1. This type of command will download programs to devices such as PROM programmers.
$ binex -s1 prog prog.S1	generates "prog.S1" in S1 format from the binary file, "prog".
$ exbin prog.S1 cmds/prog	generates "cmds/program" in OS-9 binary format from the S1 type file, "program.S1".

BUILD BASIC UTILITY COMMAND BUILD

Build a text file from standard input

SYNTAX: BUILD [<opts>] <path> [<opts>]

FUNCTION: BUILD is used to create short text files by copying the standard input into the file specified by <path>.

EXECUTION: Type "build" and a pathlist. BUILD creates a file according to the pathlist parameter, then displays a "?" prompt to request an input line. Each line entered is written to the output file. Entering a line consisting of only a carriage return causes BUILD to terminate. BUILD will also terminate when an end-of-file character is entered (typically ESCAPE) at the beginning of an input line.

OPTIONS: -? Displays the usage of BUILD

EXAMPLES:

```
$ build newfile
? Build should only be used
? in creating short text files.
? [RETURN]

$ list newfile
Build should only be used
in creating short text files.
```

CFP PROGRAMMER UTILITY COMMAND CFP

Command File Processor

SYNTAX: CFP [<opts>] [<path1>] [<path2> [<opts>]]

FUNCTION: CFP creates a temporary procedure file on the default system drive and then invokes the shell to execute it.

EXECUTION: Type "cfp", the name of the procedure file (<path1>), and the file(s) (<path2>) to be executed by the procedure file. The name of the procedure file may be omitted if the "-s=<string>" option is used.

All occurrences of "@" in the procedure file (<path1>) are replaced by the given pathlists (<path2>) unless preceded by "~" (e.g., "~@" will translate to "@"). The command procedure will not be executed until all input files have been read.

OPTIONS:

-? Displays the usage of CFP.

-d Deletes the temporary file (default).

-nd Will not delete the temporary file.

-e Executes the procedure file (default).

-ne Will not execute the procedure file. Instead, it will dump to standard output. This option causes "-d" and "-nd" to have no effect because the temporary procedure file is not created.

-s=<str> Reads <str> instead of a procedure file. If the string contains characters that are interpreted by the shell, the entire option needs to be enclosed in quotes. It does not make sense to specify both a procedure file and this option.

-z Reads file names from standard input instead of <path2>.

-z=<file> Reads file names from <file> instead of <path2>.

(continued)

EXAMPLES:

In this example, "test.p" is a procedure file that contains "list * >/p2". The following command will produce the following procedure file:

```
$ cfp test.p file1 file2 file3

procedure file:  list file1 >/p2
                 list file2 >/p2
                 list file3 >/p2
```

The following command accomplishes the same thing:

```
$ cfp "-s=list * >/p2" file1 file2 file3
```

Note that double quotes are required to force the shell to send the string (-s=list * >/p2) as a single parameter to CFP. The quotes also prevent the "*" from being expanded by the shell to be all pathlists in the current data directory.

Change current data directory
Change current execution directory

SYNTAX: CHD <path>
CHX <path>

FUNCTION: CHD and CHX are "built-in" SHELL commands used to change OS-9's working data directory or working execution directory.

EXECUTION: Type "chd" and the pathlist to the data directory desired. Type "chx" and the pathlist to the new execution directory. Either a full or relative pathlist may be used.

NOTE: These commands do not appear in the CMDS directory as they are built-in to the SHELL.

EXAMPLES:

```
$ chd /d1/PROGRAMS

$ chx ..

$ chx binary_files/test_programs

$ chx /D0/CMDS; chd /D1
```

CMP BASIC UTILITY COMMAND CMP

Compare Binary Files

SYNTAX: CMP [<opts>] <path1> <path2> [<opts>]

FUNCTION: CMP opens two files and performs a comparison of the binary values of the corresponding data bytes of the files. If any differences are encountered, the file offset (address), the hexadecimal value, and the ASCII character for each byte is displayed.

The comparison ends when end-of-file is encountered on either file. A summary of the number of bytes compared and the number of differences found is then displayed.

OPTIONS:

-? Displays usage of CMP.

-b=<num>k Assigns <num>k of memory for CMP use. CMP uses a 4k memory size (by default).

-s Silent mode. Stops the comparison when first mismatch occurs and will print error message.

-x Searches the current execution directory for both of the specified files.

EXAMPLES:

```
$ cmp file1 file2 -b=8k

Differences

           (hex)  (ascii)
byte       #1  #2  #1  #2
========   ==  ==  ==  ==
00000019   72  6e   r   n
0000001a   73  61   s   a
0000001b   74  6c   t   l

Bytes compared:  0000002d
Bytes different: 00000003

file1 is longer

$ cmp file1 file1

Bytes compared:  0000002f
Bytes different: 00000000
```

CODE PROGRAMMER UTILITY COMMAND CODE

Prints hex value of input character

SYNTAX: CODE [<opts>]

FUNCTION: CODE prints the input character followed by the hex value of the input character. Unprintable characters will print as ".". The abort key (normally control-C) or the quit key (normally control-E) will terminate CODE.

The most common usage of CODE is to discover the value of an unknown key on the keyboard or the hex value of an ASCII character.

OPTIONS: -? Display usage of CODE

EXAMPLES:

```
$ code
ABORT or QUIT characters will terminate CODE
a -> 61
e -> 65
A -> 41
. -> 10
. -> 04

$
```

COMPRESS PROGRAMMER UTILITY COMMAND COMPRESS

Compress ASCII files

SYNTAX: COMPRESS [<opts>] {<path> [<opts>]}

FUNCTION: COMPRESS reads the specified text file(s), converts it to compressed form and writes the compressed text file to standard output or to an optional output file.

EXECUTION: Type "compress" and the path of the text file to be compressed. If no files are given on the command line, standard input is used.

COMPRESS replaces multiple occurrences of a character with a three character coded sequence:

"aaaaabbbbbccccccccccc" would be replaced with "~Ea~Eb~Jc".

Each compressed input file name is appended with "_comp". If a file with this name already exists, the old file is overwritten with the new file. Typical files will compress about 30% smaller than the original file.

COMPRESS reduces the size of a file to save disk space. See EXPAND for details on how to expand a compressed file.

WARNING: Only use COMPRESS and EXPAND on text files.

OPTIONS:

-?	Displays the usage of COMPRESS.
-d	Deletes the original file (inappropriate when no pathlist is specified on the command line and standard input is used).
-n	Creates an output file.
-z	Reads file names from standard input.
-z=<file>	Reads file names from <file>.

(continued)

EXAMPLES:

`$ compress file1 -dn`	Compresses "file1". Creates "file1_comp" and deletes "file1".
`$ compress file2 -d >file3`	Compresses "file2". Creates "file3" (from redirected standard output) and deletes "file2".

COPY BASIC UTILITY COMMAND COPY

Copy data from one path to another

SYNTAX: COPY [<opts>] <path1> [<path2>] [<opts>]

FUNCTION: COPY copies data from <path1> to <path2>.

EXECUTION: Type "copy", <path1>, and then <path2>. <path1> must already exist, and <path2> is automatically created if it is a file on a mass storage device. Data may be of any type and is NOT modified in any way as it is copied. The attributes of <path1> are also copied exactly.

If <path2> is omitted, the same name as the source file will be used for the destination file. It will be copied into the current data directory. Consequently, the two following commands accomplish the same process:

```
$ copy /h0/cmds/file1 file1

$ copy /h0/cmds/file1
```

Data is transferred using large block reads and writes until end-of-file occurs on the input path. Because block transfers are used, normal output processing of data does not occur on character-oriented devices such as terminals, printers, etc. Therefore, the LIST command is preferred over COPY when a file consisting of text is to be sent to a terminal or printer.

OPTIONS:

-? Displays the usage of COPY

-a Aborts COPY routine if an error occurs. This option effectively cancels the "continue (y/n) ?" prompt of the "-w" option.

-b=<num>k Allocates <num>k memory to be used by COPY. COPY uses a 4k memory (by default).

-p Does not print a list of files copied (for multiple file copy only).

-r Rewrites over existing file.

-v Verifies the integrity of the new file.

(continued)

OPTIONS (continued)

-w=<dir> Copies one or more files to <dir>. This option prints the file name after each successful copy. If an error occurs (no permission to copy, for example), the prompt "continue (y/n) ?" will be displayed.

-x Uses the current execution directory for <path1>.

EXAMPLES:

```
$ copy file1 file2 -b=15k          Copies file1 to file2 using
                                   a 15k buffer.

$ copy * -w=MYFILE                 Copies all files in current
                                   data directory to MYFILE.

$ copy /d1/joe /d0/jim -w=FILE     Copies /d1/joe and /d0/jim
                                   to FILE.

$ copy file3 file4 -r              Writes file3 over file4.
```

COUNT BASIC UTILITY COMMAND COUNT

Count characters, words, and lines in a file

SYNTAX: COUNT [<opts>] [<path> [<opts>]]

FUNCTION: COUNT will count the number of characters in a file and optionally print a breakdown consisting of each unique character found and the number of times it occurred.

COUNT will also count the number of words in a file. A word is defined as a sequence of nonblank, non-carriage-return characters.

Finally, COUNT will count the number of lines in a file by counting the number of carriage-returns.

OPTIONS:

-?	Displays the usage of COUNT.
-b	Counts characters and gives a breakdown of their occurrence.
-c	Counts characters.
-l	Counts lines.
-w	Counts words.
-z	Reads file names from standard input.
-z=<file>	Reads file names from <file>.

EXAMPLES:

```
$ list file1
first line
second line
third line

$ count -clw file1
"file1" contains 28 characters
"file1" contains 6 words
"file1" contains 3 lines
```

DATE BASIC UTILITY COMMAND DATE

Display system date and time

SYNTAX: DATE [<opts>]

FUNCTION: DATE will display the current system date and system time.

OPTIONS:

-?	Displays the usage of DATE.
-j	Displays the Julian date and time.
-m	Displays the military time (24 hour clock) after the date.

EXAMPLES:

```
$ date

February 7, 1984 Tuesday 2:20:20pm

$ date -m

February 7, 1984 Tuesday 14:20:24

$ date >/p          Redirects the current date and time to
                    the printer.
```

DCHECK SYSTEM MANAGEMENT UTILITY COMMAND DCHECK

Check Disk File Structure

SYNTAX: DCHECK [<opts>] <devname>

FUNCTION: DCHECK is a diagnostic that can be used to detect the condition, as well as the general integrity of the directory/file linkages.

EXECUTION: Type "dcheck", the option(s) desired, and the name of the disk device to be checked.

DCHECK first verifies and prints some of the vital file structure parameters. It moves down the tree file system to all directories and files on the disk. As it does so, it verifies the integrity of the file descriptor sectors (FDs). It reports any discrepancies in the directory/file linkages.

From the segment list associated with each file, it builds a sector allocation map. This map is created in memory, not on disk (unless specified).

If any FDs describe a segment with a cluster not within the file structure of the disk, a message is reported:

*** Bad FD segment (xxxxxx-yyyyyy)

This indicates that a segment starting at sector xxxxxx (hexadecimal) and ending at sector yyyyyy can not be on this disk. There is a good chance the entire FD is bad if any of it's segment descriptors are bad, therefore the allocation map is <u>not</u> updated for bad FDs.

While building the allocation map, DCHECK also makes sure that each disk cluster appears only once in the file structure. If a cluster appears more than once, DCHECK will display a message:

xxxxxx previously allocated

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

(continued)

Occasionally, sectors on a disk are marked as being allocated but actually they are not associated with a file or the disk's free space. This can happen if a disk is removed from a drive while files are still open, or if a directory which still contains files is deleted by a means other than DELDIR.

If not all of the sectors of a cluster are used in the file system, DCHECK prints a message:

```
xxxxxx cluster only partially used
```

If there is more than one sector per cluster, DCHECK must take the allocation map and convert it into a true bitmap (the allocation map is actually a sector by sector bitmap of the file structure). The newly created allocation map is then compared to the allocation map stored on the disk, and any differences are reported in messages:

```
xxxxxx not in file structure

xxxxxx not in bit map
```

The first message indicates sector number xxxxxx was found not to be part of the file system (but is marked as allocated in the disk's allocation map). In addition to the causes previously mentioned, some sectors may have been excluded from the allocation map by the FORMAT program because they were defective. They could be the last sectors of the disk, whose sum is too small to comprise a cluster.

The second message indicates that the cluster starting at sector xxxxxx is part of the file structure (but is <u>not</u> marked as allocated in the disk's allocation map). This type of disk error could cause later problems. It is possible that this cluster may (later) be allocated to another file. This would overwrite the current contents of the cluster with data from the newly allocated file. All current data located in this cluster would be lost. Any clusters that have been reported as "previously allocated" by DCHECK no doubt have this problem.

(continued)

OPTIONS:

-?	displays the usage of DCHECK.
-d=<num>	prints the path to the directory <num> deep.
-m	saves the allocation map work files.
-w=<path>	tells DCHECK on which pathlist to create its directory for work files.

The "-w=" option must specify a full pathlist to a directory. It is highly recommended that this pathlist be located on a separate disk than that being DCHECKed (especially if the disk's file structure integrity is in doubt).

When using the "-w=" option, DCHECK builds a disk allocation map in the file: <pathlist>/DCHECKpp0. <pathlist> is specified by the "-w=" option. "pp" is the process number in hexadecimal. Each bit in this bitmap file corresponds to a cluster of sectors on the disk.

The bitmap work files may be saved by specifying the "-m" option.

RESTRICTIONS:

DCHECK should have exclusive access to the disk being checked. DCHECK can be fooled if the disk allocation map changes while it is building its bitmap file from the changing file structure.

DCHECK can not process disks with a directory depth greater than 39 levels.

Only the Super user (user 0.0) may use this utility.

(continued)

EXAMPLES:

```
$ dcheck /d2

Volume - 'My system disk' on device /d2
$009A bytes in allocation map
1 sector per cluster
$0004D0 total sectors on media
Sector $000002 is start of root directory FD
$0010 sectors used for id, alloc map and root FD
Building allocation map ...
Checking allocation map ...

'My system disk' file structure is intact
1 directory
2 files
111360 of 262144 bytes (0.10 of 0.25 meg) used on media


$ dcheck -mw=/d2 /d0
Volume - 'System disk' on device /d0
$0046 bytes in allocation map
1 sector per cluster
$00022A total sectors on media
Sector $000002 is start of root directory FD
$0010 sectors used for id, alloc map and root FD
Building allocation map...
00040 previously allocated
*** Bad FD segment (111111-23A6F0)
Building allocation map...
000038 not in bit map
00003B not in bit map
0001B9 not in file structure
0001BB not in file structure

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 disk' file structure is not intact
5 directories
25 files
111360 of 262144 bytes (0.10 of 0.25 meg) used on media
```

DEBUG PROGRAMMER UTILITY COMMAND DEBUG

User Symbolic Debugger

SYNTAX: DEBUG [<opts> <program> <program opts>]

FUNCTION: DEBUG is a tool for debugging and testing 68000 machine language programs.

EXECUTION: Type "debug" and optionally the program to be debugged. The debugger operates in response to command lines from the keyboard. DEBUG will wait for a command with the prompt: "dbg:".

Each command line begins with a one or two character command, optionally followed by text or numeric expressions. Uppercase and lowercase letters can be used interchangeably. Each line is terminated by a carriage return.

To start a process to be examined by DEBUG, a program may be specified on the initial DEBUG command line, or once in the debugger the "f" command can be used, followed by the program name and any parameters that are needed. While this should not be a problem, it should be noted that a maximum of 64 arguments can be passed in a "f" command.

Many SHELL parameters will be recognized by the debugger in a "f" command. These include redirecting standard input and output, priority and memory modification (<, >, ^, #). Pipes (!), concurrent execution (&), sequential exectution and redirection of standard error output are not supported.

When the process to be debugged is forked ("f" command), it will inherit the same number of paths as its parent process. This is usually just standard I/O, however this is quite helpful with a process that has multiple I/O paths.

Extra memory may be allocated to the process to be debugged by using the "-m=<num>" option on the SHELL command line. This option must be given before the specified process, or it will be processed as a program option.

The symbolic debugging capabilities of DEBUG are provided by the "-g" option in the OS-9 linker ("l68"). This option will cause the global data and code symbolic names (and addresses) to be placed in an OS-9 data module. The data module is then loaded into memory by DEBUG. It is used to determine actual addresses from symbolic names.

NOTE: DEBUG can not debug anything that runs in "supervisor mode", so it is useless for debugging the kernel, file managers and device drivers.

(continued)

OPTIONS: -? Displays the usage of DEBUG.

-m=<num> Allocates <num>k of memory to the process to be debugged (specified on the initial SHELL command line). This option must appear before the specified process, otherwise it will be processed as an option of the specified process (not the debugger). This does not affect the memory allocation for subsequent programs forked within the debugger.

COMMAND SYNTAX: In command lines <addr>, <len> and <num> are allowed to be expressions:

<addr> is any symbol, number, or expression.

<num> is by default a hexadecimal value; the value is decimal if preceded with a "#"; the value is ASCII if preceded by a single quote ("'").

<len> is a decimal number; specifying "*" will give infinity.

Traditional precedence is not observed in these expressions and evaluation is simply applied from left to right. However, the use of parentheses is legal. It will cause the contained expression to be evaluated and the result used in the overall expression.

".<reg>" is a legal operand. For example, the command "b .a0" will set a breakpoint at the address in register a0. A register operand may be any of the following (NOTE: The period (".") is used to differentiate between hex numbers and the register designations):

```
.d0 thru .d7    data registers
.a0 thru .a7    address registers
.r0 thru .r7    relative registers (not 68000 regs!)
.cc             condition codes    (status register)
.sp             user stack pointer (ssp is not available)
.pc             program counter
```

(continued)

An expression may contain any of the operators or modifiers in the following three charts:

BINARY OPERATORS (operate on the left and right operand)	
+	adds the operands together.
-	subtracts the right operand from the left operand.
*	multiplies the operands.
/	divides the left operand by the right operand.
>	shifts the left operand to the right by the number of bits specified by the right operand. Example: A > B shifts A right by B count of bits.
<	shifts the left operand to the left by the number of bits specified by the right operand. Example: A < B shifts A left by B count of bits.

BINARY OPERATORS (operate on the left and right operand)	
&	connects the operands by the bitwise "and".
|	connects the operands by the bitwise "or".
^	connects the operands by the bitwise "exclusive or".

Unary operators (operates on the immediate right operand)	
-	negates the operand.
~	converts the operand to one's complement form.

(continued)

Modifiers	
()	evaluates the expression within as a single operand, making it possible to affect the order of operation.
[]l	evaluates the expression within to be the address of a long operand (which is retrieved from memory).
[]w	evaluates the expression within to be the address of a word operand (which is retrieved from memory).
[]b	evaluates the expression within to be the address of a byte operand (which is retrieved from memory).
[]	evaluates the expression within to be the address of a operand (defaults to long operand).

The following list of expressions were evaluated using the register values shown below. Their results are shown following the expression:

```
.d1 = $12345

.d3 = $4A

.d5 = $FA08

.a0 = $8164

.a2 = $45

.a6 = $20768

values starting at address $20768 are $42 $69 $6C $6C
```

Expression	Result
.d1	12345
.d5 > 3	1F41
11+59	6A

(continued)

Expression	Result
FE61*2	1FCC2
.a0-24	8140
.a0&FFF	164
.d5-5*.a2	%362CF
.d5-(5*.a2)	F8AF
.a2^.d3	F
[.a6]l	%2696C6C
[.a6]w	%269
[.a6+3]b	6C

DEBUG COMMANDS: In the following command descriptions, <path> specifies the file to store the output of a command.

COMMAND	DESCRIPTION
b	Lists breakpoints
b <addr>	Sets a breakpoint at <addr>
c <addr>	Switches the debugger to change mode at <addr>. Byte values are changed with this command. Change mode commands are: + move to next location - move to previous location <CR> move to next location and display <num> store new value and move to next . exit change mode

(continued)

COMMAND	DESCRIPTION
c <addr> (continued)	NOTE: Commands may be strung together. For example, "c fff0e0 03 + ff ." would change the locations FFF0E0 and FFF0E2 but display nothing.
cw <addr>	Switches the debugger to change mode at <addr>. Word values are changed with this command. See the "c <addr>" command for the change mode commands.
cl <addr>	Switches the debugger to change mode at <addr>. Long word values are changed with this command. See the "c <addr>" command for the change mode commands.
d[n] <addr> [<len>] [><path>]	Switches the debugger to display mode and displays the memory at <addr> for <len> bytes in hex and ASCII. The default for <len> is 256 bytes. Display commands are: <CR> displays the next <len> bytes [n] specifies how many lines (0 to 9) to display if <len> is missing Any other command exits display mode.
di <addr> [<len>] [><path>]	Switches the debugger to display mode and displays <len> of instructions at <addr>. The default for <len> is 20 instructions.

DEBUG (continued)

COMMAND	DESCRIPTION
f <program> [<params>]	Begins the debugging process. It prepares the program for execution. It does not execute any code in the program. In addition to program parameters, <params> may include these SHELL parameters: memory allocation (#) process priority (^) and I/O redirection (<>). Pipes (!), concurrent execution (&), sequential exectution and redirection of error output are not supported.
g	Starts execution at current program counter. The debugger will stop at the next breakpoint.
g <addr>	Starts execution at <addr> and stops at the next breakpoint.
gs	Executes a branch subroutine (without displaying subroutine instructions).
gs <addr>	Executes until the program counter = <addr>. If the debugger finds a breakpoint before <addr>, it will stop. It will not remember <addr> on next "g" or "gs".
i	Prints the number of instructions executed.
k <addr>	Kills the breakpoint at <addr>.
k*	Kills all breakpoints.
l <module>	Links the debugger to the named module. The address is placed in .r7.

DEBUG (continued)

COMMAND	DESCRIPTION
q	Quits the debugger mode.
s <symbol> [><path>]	Displays symbols (wild cards "*" and "?" are allowed). If no <symbol> is specified, the entire symbol table is printed.
t [<num>]	Traces one or <num> instructions and switches to trace mode. Trace displays registers after each <num> instructions has been traced. Trace mode commands: <CR> trace another <num> instructions any other command exits trace mode
v <expr>	Prints the value of <expr> in hexadecimal and decimal.
.	Displays registers.
.r	Displays relocation registers.
.<reg> <num>	Sets register <reg> to <num>.
$	Calls the system shell.
@	Prints the current relative register.
@<0-7>	Changes register <0-7> to the current relative register.

(continued)

ERROR MESSAGES

ERROR MESSAGE	DESCRIPTION
can't get device name	The debugger could not determine the terminal name.
can't open <path>	The specified pathlist could not be opened.
not a valid address	The expression parser has received an error when parsing the command line.
not a valid register	The register reference is invalid, (not in range of .d0-.d7, .a0-.a7, .cc, .pc, .r0-.r7, .sp.)
breakpoint table full	No more breakpoints can be set.
odd breakpoint address	Instruction addresses must be even.
disasm address must be even	Instruction addresses must be even.
breakpoint not found	An attempt to kill a nonexisting breakpoint has occurred.
invalid data	An invalid number was used in an attempt to change memory.
no module name	An "f" command was given without specifying a module to fork.
unknown cmd	An unrecognized command was used.
forking error	The module specified by the "f" command (fork) can not be found.
not at "bsr"	The "gs" command was given with the pc not at a bsr instruction.
execution error	An address trap, bus error, etc. has occurred while running user program.

(continued)

ERROR MESSAGE	DESCRIPTION
received signal	The debugger has received a signal.
parenthesis missing	A closing parentheses was expected but not found.
bracket missing	A closing bracket was expected but not found.
symbol '<name>' not found	The specified symbol (<name>) could not be found in the symbol module.
symbol module is obsolete	The CRC that is stored in the symbol module does not match the CRC for the program. This is usually caused by the .stb module already being in the module directory.
symbol module not found	The symbol module for the program (pathlist should be "program.stb") was not found. The program should be relinked with the -g option to make the .stb file.
bus error	Bus error occurred in the debugger.
address error	An attempt to use an odd address as the address of a longword was made in the debugger.
can't allocate space for parameters	A memory request to set up parameters for a forked process failed.

EXAMPLE DEBUG SESSION

The following Debugging session is provided to not only show the format, but the use of certain Debug commands. Careful tracing of the following program and the debugging session example (with the command list) will allow fuller understanding of the debugger's use.

The following C program reads input and writes it to output:

```
main()
{
register int i;
char buff[100];

while ((i=read(0,buff,100)) > 0)
write(1,buff,i);
}
```

EXPLANATION	DEBUG SESSION
Calls program "em" and prepares it for execution.	debug dbg: f em
	Dn: 0000000F 000C0004 00000080 00000003 00000000 00000001 00001000 00000000 An: 00000000 00010700 00000000 00133C00 00000000 000106FE 00017700 000106FE
Directs debugger to execute until PC = main.	PC: _ostart >2D468010 move.l d6,_totmem(a6) CC: ----- dbg: gs main
	Dn: 00000001 000106FA 0000F718 00000003 00000000 FFFFFFFF 00001000 00000000 An: 0000FAA0 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 000106F6
Switches debugger to display mode and displays 8 instructions at the current PC address: main.	PC: main >48E7C880 movem.l d0-d1/d4/a0,-(a7) CC: ----- dbg: di .pc 8
	main >48E7C880 movem.l d0-d1/d4/a0,-(a7)
	main+4 >203CFFFFFF54 move.l #-172,d0
	main+A >6100FED0 bsr _stkchec
	main+E >4FEFFF9C lea.l -100(a7),a7
	main+12 >6010 bra.s main+24
A <CR> repeats the previous command and displays 8 more instructions at the new PC address: main +1C.	main+14 >2F04 move.l d4,-(a7)
	main+16 >41EF0004 lea.l 4(a7),a0
	main+1A >2208 move.l a0,d1
	dis:
	main+1C >7001 moveq.l #1,d0
	main+1E >610000BA bsr write
	main+22 >588F addq.l #4,a7
This command not only exits the display mode, but sets a breakpoint at the specified address: main +30.	main+24 >48780064 pea.l 0064
	main+28 >41EF0004 lea.l 4(a7),a0
	main+2C >2208 move.l a0,d1
	main+2E >7000 moveq.l #0,d0
	main+30 >610000A0 bsr read
	dis: b main+30
	ebug
These three "s" commands display all the symbols from the symbol table that begin with an "r", an "f" and an "e" respectively.	dbg: s r*
	rewind C 00133E16 read C 00133E8E readln C 00133E92
	dbg: s f*
	freemem C 00133CDA fopen C 00133E02 fdopen C 00133E06
	freopen C 00133E0A fseek C 00133E12 fread C 00133E1A
	fgets C 00133E22 fprintf C 00133E36 fwrite C 00133E3E
	fputs C 00133E46 fflush C 00133E52 fclose C 00133E56
	ftell C 00133E5A fscanf C 00133E66
	dbg: s e*
	errno D 0000F70C end D 0000FAA0 etext C 00133F94
	exit C 00133FC0

```
EXPLANATION          DEBUG SESSION
-------------------  ------------------------------------------------------------------------
Execution starts at
the current PC and>  dbg: g
continues until      Dn: 00000000 00010682  0000F718 00000003   00000000 FFFFFFFF  00001000 00000000
a breakpoint occurs  An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 0001067E
                     PC: main+30       >610000A0             bsr read                      CC: --Z--
Execute bsr at the>  dbg: gs
read, which waits    this is a line i am typing
for a <CR>:"this.."  Dn: 0000001B 00010682  0000F718 00000003   00000000 FFFFFFFF  00001000 00000000
                     An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 0001067E
Displays 32 bytes    PC: main+34       >588F                 addq.l #4,a7                  CC: --Z--
starting at .d1   >  dbg: d .d1 20
("this is ...")      end+BE2        - 7468 6973 2069 7320 6120 6C69 6E65 2069   this is a line i
                     end+BF2        - 2061 6D20 7479 7069 6E67 0DF6 0000 000D    am typing.v....
Switches debugger >  dis: t
to trace mode for    Dn: 0000001B 00010682  0000F718 00000003   00000000 FFFFFFFF  00001000 00000000
next instruction.    An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 00010682
                     PC: main+36       >2800               move.l d0,d4                  CC: --Z--
As this and the   >  tra:
following 6 "t"      Dn: 0000001B 00010682  0000F718 00000003   0000001B FFFFFFFF  00001000 00000000
commands do not      An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 00010682
specify a number     PC: main+38       >6EDA               bgt.s main+14->               CC: -----
of instructions   >  tra:
to trace, they       Dn: 0000001B 00010682  0000F718 00000003   0000001B FFFFFFFF  00001000 00000000
trace only one.      An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 00010682
                     PC: main+14       >2F04               move.l d4,-(a7)               CC: -----
Note: conditional >  tra:
branch at main+38    Dn: 0000001B 00010682  0000F718 00000003   0000001B FFFFFFFF  00001000 00000000
indicates the        An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 0001067E
while loop. Since    PC: main+16       >41EF0004           lea.l 4(a7),a0                CC: -----
a line has been   >  tra:
read, the branch     Dn: 0000001B 00010682  0000F718 00000003   0000001B FFFFFFFF  00001000 00000000
will execute         An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 0001067E
                     PC: main+1A       >2208               move.l a0,d1                  CC: -----
                  >  tra:
                     Dn: 0000001B 00010682  0000F718 00000003   0000001B FFFFFFFF  00001000 00000000
                     An: 00010682 000C857C  000C8500 0000F70C   00133C00 00133C66  00017700 0001067E
                     PC: main+1C       >7001               moveq.l #1,d0                 CC: -----
```

EXPLANATION	DEBUG SESSION
A write bar is encountered. >	tra:
	Dn: 00000001 00010682 0000F718 00000003 0000001B FFFFFFFF 00001000 00000000
This will trace	An: 00010682 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 0001067E
the bar and display	PC: main+1E >610000BA bar write CC: -----
the line read. >	tra: gs
This leaves trace >	this is a line i am typing
mode.	Dn: 0000001B 00010682 0000F718 00000003 0000001B FFFFFFFF 00001000 00000000
	An: 00010682 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 0001067E
Executes until the	PC: main+22 >588F addq.l #4,a7 CC: --Z--
read bar is again >	dbg: g
encountered.	Dn: 00000000 00010682 0000F718 00000003 0000001B FFFFFFFF 00001000 00000000
	An: 00010682 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 0001067E
Executes read bar	PC: main+30 >610000A0 bar read CC: --Z--
(which waits for a>	dbg: g
<CR> and gets one)>	this is the second line
The write bar >	this is the second line
executes, displays	Dn: 00000000 00010682 0000F718 00000003 00000018 FFFFFFFF 00001000 00000000
the new line.	An: 00010682 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 0001067E
	PC: main+30 >610000A0 bar read CC: --Z--
A new "g" command >	dbg: g
continues this type	Dn: 0000FA98 00000000 0000F718 00000003 00000000 FFFFFFFF 00001000 00000000
of execution.	An: 0000FAA0 000C857C 000C8500 0000F70C 00133C00 00133C66 00017700 000106F6
	PC: _exit+6 >DEADDEAD add.l -8531(a5),d7 CC: --Z--
The "q" command >	dbg: q
quits the debugger.	

DEINIZ SYSTEM MANAGEMENT UTILITY COMMAND DEINIZ

Detach device

SYNTAX: DEINIZ [<opts>] {<modname> [<opts>]}

FUNCTION: DEINIZ performs an I$Detach call on the specified modules. When a device is no longer needed, DEINIZ may be used to remove the device from the system device table. See the description of I$Detach in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

EXECUTION: Type "deiniz", followed by the name of the module(s) to be detached. <modname> can begin with a "/" if desired. The module names may be read from standard input or from a specified pathlist if the "-z" option is used.

OPTIONS:

-?	Displays usage of DEINIZ.
-z	Reads the module names from standard input.
-z=<file>	Reads the module names from <file>.

EXAMPLE:

```
$ deiniz t1 t2 t3
```

DEL BASIC UTILITY COMMAND DEL

Delete a file

SYNTAX: DEL [<opts>] [<path> [<opts>]]

FUNCTION: DEL is used to delete the file(s) specified by the pathlist(s). The user must have write permission for the file(s) to be deleted. Directory files can not be deleted unless their attribute is changed to non-directory: see the ATTR and DELDIR command descriptions.

OPTIONS:

`-?` Displays the usage of DEL.

`-p` Prompts for each file to be deleted with the following prompt:

```
delete <filename> ? (y,n,a,q)
```

Y = yes. N = no. A = delete all specified files without further prompts. Q = quit the deleting process.

`-x` Looks for the file in the current execution directory (NOTE: the file must have write permission).

`-z` Reads the file names from standard input.

`-z=<file>` Reads the file names from <file>.

EXAMPLES:

```
$ dir

   Directory of /D1  14:29:46
junk              myfile           newfile
number_five       old_test_program test_program

$ del newfile

$ del test_program old_test_program /D1/number_five

$ dir

   Directory of /D1  14:30:37
junk              myfile

$ dir -u ! del -z     Delete all files in current directory
```

DELDIR BASIC UTILITY COMMAND DELDIR

Delete All Files In a Directory

SYNTAX: DELDIR [<opts>] [<dir> [<opts>]]

FUNCTION: DELDIR is a convenient alternative to manually deleting directories and files they contain. It is only used when all files in the directory are to be deleted.

EXECUTION: When DELDIR is run, it prints a prompt message like this:

```
$ deldir OLDFILES

Deleting directory: OLDFILES
Delete directory, List directory, or Quit (d,l,q) ?
```

A "d" response will initiate the process of deleting files. A "q" response will abort the command before action is taken. An "l" response will cause a "dir -e" command to be run so you can have an opportunity to see the files in the directory before they are deleted. After listing the files, DELDIR will prompt with:

```
delete ? (y,n)
```

The directory to be deleted may include directory files, which may themselves include directory files, etc. In this case, DELDIR operates recursively (it calls itself), so all lower-level directories are deleted as well. The lower-level directories are processed first.

You must have correct access permission to delete all files and directories encountered. If not, DELDIR will abort upon encountering the first file for which you do not have write permission.

The DELDIR command automatically calls the DIR and ATTR commands, so they must reside in the current execution directory.

NOTE: You should never delete the current data directory (".").

OPTIONS:

-?	Displays the usage of DELDIR.
-q	Quiet mode; no questions will be asked.
-z	Reads the file names from standard input.
-z=<file>	Reads the file names from <file>.

DIR BASIC UTILITY COMMAND DIR

Display the names of files contained in a directory

SYNTAX: DIR [<opts>] [<dir> [<opts>]]

FUNCTION: DIR displays a formatted list of file names in a directory file on standard output.

EXECUTION: Type "dir" (and the directory pathlist if desired). If no parameters are given, the current data directory is shown. If the "-x" option is given, the current execution directory is shown. If a pathlist of a directory file is given, the indicated directory is shown.

If the "-e" option is included, each file's entire description is displayed: size, address, owner, permissions, date and time of last modification. Because SHELL does not interpret the "-x" option, wild cards will not work as expected when this option is used.

File names that begin with "." will not be displayed, unless the "-a" option is used.

OPTIONS:

-?	Displays the usage of DIR.
-a	Displays the directory with all file names (including file names beginning with a ".").
-d	Appends "/" to all directory names listed. This does not affect the actual name of the directory.
-e	Displays an extended directory listing (excluding file names beginning with a ".").
-n	Displays directory names without displaying the file names they contain. This option is especially useful using wildcards. See example.
-r	Displays the directories recursively (excluding file names beginning with a ".").
-r=<num>	Displays the directories recursively up to the <num> level below the current directory (excluding file names beginning with a ".").
-s	Displays an unsorted listing (excluding filenames beginning with a ".").

(continued)

OPTIONS:

-u	Displays an unformatted listing (excluding filenames beginning with a ".").
-x	Displays current execution directory (excluding file names beginning with a ".").
-z	Reads the directory names from standard input.

UNFORMATTED DIRECTORY LISTING

It is possible to print an unformatted directory listing using the "-u" option. This allows only the names of the entries of a directory to be displayed. No directory header will be displayed, and entries will be printed as follows:

```
file1
file2
file3
DIR1
```

The output of DIR can then be sent through a pipe to another utility command or program that can utilize a pipe. For example:

```
$ dir -u ! attr -z
```

This displays the attributes of every entry in the current directory.

The "-e" option can be used to display an extended directory listing without the header by adding the "-u" option:

```
$ dir -ue
```

(continued)

EXAMPLES:

```
$ dir             Display directory

   Directory of . 12:12:54
BK              BKII            RELS
ed10.c          ed11.c          ed2.c
```

```
$ dir ..          Displays parent of working data directory
```

```
$ dir NEWSTUFF    Displays NEWSTUFF directory
```

```
$  dir -e         Displays the entire description of current
                  data directory
                      Directory of . 12:13:00
 Owner    Last Modified Attributes Sector Bytecount Name
-------   ------------- ---------- ------ --------- ----
  0.0     85/02/28 0906  d-ewrewr    26E8       640 BK
  0.0     85/02/27 1846  d-ewrewr    CE00       512 BKII
 12.4     85/03/11 1255  ------wr    D048       480 RELS
 12.4     85/03/10 1417  ------wr    D0D0     11113 ed10.c
 12.4     85/03/10 1601  ------wr    3458      5744 ed11.c
 12.4     85/03/11 1148  ------wr   104A0     15944 ed2.c
```

```
$ dir -x          Displays execution directory
```

```
$  dir  -xe       Displays the entire description of the
                  execution directory
```

```
$  dir  -r=1      Displays contents of the current directory
                  and all directories within this directory
```

```
$  dir  -er       Displays entire description of all files
                  within the current directory, including
                  files within all subdirectories of the
                  current directory
```

```
Dir  -n B*        Displays all directory and file names that
                  begin with "B".
```

DSAVE BASIC UTILITY COMMAND DSAVE

Generate procedure file to copy files

SYNTAX: DSAVE [<opts>] [<path>] [<opts>]

FUNCTION: DSAVE is used to backup or copy all files in one or more directories. It generates a procedure file, which is executed later to actually do the work or (using the "-e" option) executed immediately.

EXECUTION: Type "dsave" and the path of the new directory. When DSAVE is executed, it writes commands (on standard output) to copy files from the current data directory to the directory specified by <path>. If no <path> is specified, the copies are directed to the current data directory at the time the procedure file is executed.

To utilize DSAVE, standard output should be redirected to a procedure file (that can be executed at a later time) or the "-e" option should be used (so the execution would be immediate).

If DSAVE encounters a directory file, it will automatically include MAKDIR and CHD commands in the output before generating copy commands for files in the subdirectory. DSAVE is recursive in operation. Therefore, the procedure file will duplicate all levels of the file system connected downward from the current data directory (such a section of the file system is called a "subtree").

If the current working directory happens to be the root directory of the disk, DSAVE will create a procedure file that will backup the entire disk, file by file. This is useful when it is necessary to copy many files from different format disks, or from a floppy disk to a hard disk.

If an error occurs (e.g. no permission), the following prompt will be displayed:

```
continue (y,n,a,q)?
```

"a" indicates copy all possible files and do not display prompt on error. "q" indicates quit DSAVE procedure. If for any reason you do not wish to be bothered by this prompt, the "-s" option is available. This will skip any file which can not be copied and continue the DSAVE routine (with no prompt).

(continued)

DSAVE is especially helpful for keeping up to date directory backups. Using the "-d" or "-d=<date>" option, DSAVE will compare the date of the file to be copied with a file of the same name in the directory it is to be copied to. The "-d" option will copy any file with a more recent date. The "-d=<date>" option will copy any file with a date more recent than that specified.

A common error occurs when using DSAVE: the "destination" directory has files with the same name as the "source" directory. Because of file naming conventions (a file name must be unique within a directory), this produces an error.

OPTIONS:

-?	Displays the usage of DSAVE.
-a	Will not copy any file that has a name beginning with a ".".
-b[=]<n>k	Allocates <n> k bytes of memory for COPY (and CMP if needed).
-d	Compares dates with files of the same name and copies files with more recent dates.
-d=<date>	Compares the specified date with the date of files with the same name and copies any file with a more recent date (than that specified).
-e	Executes the output immediately.
-i	Indents for directory levels.
-l	Does not save directories below current level.
-m	Does not include MAKDIR commands in the procedure file.
-n	Does not load COPY (or CMP if "-v" is specified).
-o	Uses OS9GEN to create a bootfile on the specified destination device, if a bootfile exists on the source device. The default name used for the bootfile is OS9Boot. This option is used to create a bootable disk. Merely copying OS9Boot to a new disk will not make it bootable.

(continued)

OPTIONS:

-o=<n> Uses OS9GEN to create a bootfile on a new device, using the specified name. This is used to create a bootable disk. Merely copying OS(Boot to a new disk will not make it bootable.

-r Writes any source file over a file with the same name in the destination directory (effectively, this uses the COPY command with the "-r" option).

-s Skips files on error. This effectively turns off the prompt to continue the DSAVE routine when an error occurs.

-v Verifies files with CMP.

EXAMPLES:

In the first example, all files are copied from /d2 to /d1. No path is specified in the DSAVE command. Therefore it is necessary to change data directories before executing the procedure file. If the directory is not changed, an error message will occur (#218--file already exists in this directory under the same name).

```
$ chd /d2                 Selects "/d2" as the directory to be
                          copied.

$ dsave >/d0/makecopy     Makes the procedure file "makecopy".

$ chd /d1                 Select the destination directory for
                          "makecopy".

$ /d0/makecopy            Run "makecopy".
```

The second example copies all files from "/d0/MYFILES/STUFF" to "/d1/BACKUP/STUFF". It uses the path "/d1/BACKUP/STUFF" in the DSAVE command. There is consequently no need to change directories before executing the procedure file. This example also allocates 32k of memory for the copy procedure. This usually saves time.

```
$ chd /d0/MYFILES/STUFF
$ dsave -ib=32  /d1/BACKUP/STUFF >saver
$ saver
```

(continued)

EXAMPLES (continued):

The third example will effectively accomplish the same thing, but without the use of a procedure file.

```
$ chd /d0/MYFILES/STUFF
$ dsave -ieb32 /d1/BACKUP/STUFF
```

In the following example the dates shown in the directory are for the purpose of displaying the "-d" option only. They do not appear in a "dir" listing.

```
$ chd /d0/BACKUP

$ dir

   directory of . 14:14:32
file1 (June 1, 1985)      file2 (Jan. 1, 1985)

$ chd /d0/WORKFILES

$ dir

   directory of . 14:14:40
file1 (May 1, 1985)       file2 (June 1, 1985)

$ dsave -deb32 /d0/BACKUP

$ chd /d0/BACKUP

$ dir

   directory of . 14:15:45
file1(June 1, 1985)  file2(June1, 1985)
```

In this example only file2 was copied, because it had a more recent date. In this way you may keep an up to date backup file.

DUMP PROGRAMMER UTILITY COMMAND DUMP

Formatted File Data Dump in Hexadecimal and ASCII

SYNTAX: DUMP [<opts>] [<path>] [<addr>] [<opts>]

FUNCTION: DUMP produces a formatted display of the physical data contents of <path> which may be a mass storage file or any other I/O device. The DUMP command is commonly used to examine the contents of non-text files.

EXECUTION: Type "dump" and the pathlist (and address within the file if desired) of the file to be displayed. If <path> is omitted, standard input is used. The output is written to standard output. When <addr> is specified, the contents of the file will be displayed starting with the appropriate address. <addr> is presumed to be a hexadecimal number.

The data is displayed 16 bytes per line in both hexadecimal and ASCII character format. Data bytes that have non-displayable values are represented by periods in the character area.

The addresses displayed on the dump are relative to the beginning of the file. Because memory modules are position-independent and stored in files exactly as they exist in memory, the addresses shown on the dump are relative to the load addresses of the memory modules.

OPTIONS:

-?	Displays the usage of DUMP.
-c	Does not compress duplicate lines.
-x	Indicates that <path> is an execution directory; the user must have execute permission for the pathlist.

EXAMPLES:

```
$ dump                  Displays keyboard input in hex.

$ dump myfile >/P       Dumps "myfile" to printer.

$ dump shortfile        Dumps "shortfile".
```

(continued)

EXAMPLES (continued):

Sample Output:

(starting address) (data bytes in hexadecimal format) (data bytes in ASCII format)

```
Addr   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
----   ----------------------------------------  ----------------
0000   87CD 0038 002A F181 2800 2E00 3103 FFE0  .M.8.*q.(...1..`
0010   0418 0000 0100 0101 0001 1808 180D 1B04  ................
0020   0117 0311 0807 1500 002A 5445 52CD 5343  .........*TERMSC
0030   C641 4349 C10E 529E                      FACIA.R.
```

ECHO BASIC UTILITY COMMAND ECHO

Echo text to output path

SYNTAX: ECHO [<opts>] [<text> [<opts>]]

FUNCTION: ECHO echoes its argument to the standard output path. It is typically used to generate messages in SHELL procedure files or to send an initialization character sequence to a terminal.

EXECUTION: Type "echo" and the text to be output. ECHO will read the text until a carriage return. It will then echo the input on the output path.

A hex number representing a character may be imbedded in a character string but it must be preceded by a "\". Shell removes all but one imbedded spaces from character strings passed to ECHO. Therefore, to allow for more than one blank between characters, the string must be enclosed by double quotes. A single "\" is echoed by entering "\\".

NOTE: The text should not include any of the punctuation characters used by the SHELL unless the string is enclosed by double quotes.

OPTIONS:

-? Displays the usage of ECHO.

-n Separates the text with carriage returns.

-r Does not send a carriage return after <text>.

-z Reads the text from standard input.

-z=<file> Reads the text from <file>.

EXAMPLES:

```
$ echo "Here is an important message!"
Here is an important message!

$ echo \41\42\43abcd efg\44           cccccc
ABCabcd efgD cccccc

$ echo "aaaaa            aaaa\41\41\41\\      bbbbb"
aaaaa            aaaaAAA\      bbbbb
```

EDT BASIC UTILITY COMMAND EDT

Line Editor

SYNTAX: EDT [<opts>] <path> [<opts>]

FUNCTION: EDT is a line oriented text editor that allows the user to create and edit source files.

EXECUTION: Type "edt" and the pathlist desired. If the file is new or can not be found, EDT will create and open it. EDT will then display a "?" prompt and wait for a command. If the file is found, EDT will open it, display the last line and then display the "?" prompt.

The first character of a line must be a space if text is to be inserted. If any other character is typed in the first character position, EDT will try to process the character as an EDT command. EDT command format is very similar to BASIC09's edit mode.

EDT determines the size of the file to be edited and uses the returned size plus 2k as the edit buffer. If the file does not already exist, the edit buffer will be initialized to 2k. When the end of the edit buffer is reached, a message will be displayed.

OPTIONS: -? Displays the usage for EDT.

-b=<num>k Allocates a buffer area equal to the size of the file plus <num> k bytes. If the file does not exist, a buffer of the indicated size will be assigned for the new file.

EDIT COMMANDS: All EDT commands begin in the first character position of a line.

<num> moves curser to line number <num>.

<esc> closes file and exits ("q" also does this).

<cr> moves curser down one line (carriage return).

+<num> moves curser down <num> lines (default is one).

-<num> moves curser up <num> lines (default is one).

<space> inserts lines.

d[<num>] deletes <num> lines (if <num> is not specified, the default value of <num> is 1).

EDIT COMMANDS (continued)

l<num> lists <num> lines. <num> may be positive or negative (default value of <num> is 1).

q quits the editing session. Command will return to the OS-9 Shell or the program that called the editor.

NOTE: For the following search and replace commands, <delim> may be any character. The "*" option indicates that all occurrences of the pattern will be searched for (and replaced if specified).

s[*]<delim><search string><delim>

search command: searches for the occurrences of a pattern. Examples:

s/and/ - find the first occurrence of "and".

s*,Bob, - find all occurrences of "Bob".

c[*]<delim><search string><delim><replace string><delim>

replace command: find and replace a given string. Examples:

c/Tuesday/Wednesday/

replaces the first occurrence of "Tuesday" with "Wednesday".

c*"employee"employees"

replaces all occurrences of "employee" with "employees".

EX PROGRAMMER UTILITY COMMAND EX

Execute program as overlay

SYNTAX: EX <path> [<arglist>]

FUNCTION: EX is a SHELL built-in command that causes the process executing the command to start execution of another program. It permits a transition from the SHELL to another program without creating another process, thus conserving system memory.

EX is often used when the SHELL is called from another program to execute a specific program, after which the SHELL is not needed. For instance, applications which use only BASIC09 need not waste memory space on SHELL .

EX should always be the last command on a SHELL input line because any command line following will never be processed.

NOTE: Because this is a built-in SHELL command, it does not appear in the CMDS directory.

EXAMPLES:

```
$ ex BASIC09

$ tsmon /t1&; tsmon /t2&; ex tsmon /term
```

EXPAND PROGRAMMER UTILITY COMMAND EXPAND

Expand a compressed file

SYNTAX: EXPAND [<opts>] [<path> [<opts>]]

FUNCTION: EXPAND will restore compressed files to their original form. It is the complement command of COMPRESS.

EXECUTION: Type "expand" and the name of the file to be expanded. If no file names are given on the command line, standard input is assumed.

OPTIONS:

-?	Displays the usage of EXPAND.
-d	Deletes the old version of the file (this is not appropriate when no pathlist is specified on the command line and standard input is used).
-n	Sends output to a file instead of standard output. The file will have "_exp" appended to it, unless the file name already has a "_comp" suffix. In this case, the "_comp" is removed.
-z	Reads the file names from standard input.
-z=<file>	Reads the file names from <file>.

If the file already has an "_exp" appended to it, when EXPAND is executed with the "-n" option, the old "<path>_exp" will be destroyed and replaced with the new "<path>_exp".

EXAMPLES:

$ expand data.a -nd	Expands and then deletes "data.a", creating "data.a_exp".
$ expand file1_comp	Expands "file1_comp" and displays output on standard output.
$ expand -nd file2_comp	"file2_comp" is expanded and then deleted, creating "file2" with the expanded output.

FIXMOD SYSTEM MANAGEMENT UTILITY COMMAND FIXMOD

Fix module CRC, parity

SYNTAX: FIXMOD [<opts>] [<modname> [<opts>]]

FUNCTION: FIXMOD is used to verify and update module CRC (cyclic redundancy check) and module parity.

EXECUTION: Type "fixmod" followed by the name of the module to be updated or verified.

The "-u" option is used to recalculate and update CRC and parity. Without the -u option, FIXMOD will only verify the CRC and parity of the module. It will display a message if the module is good or not.

FIXMOD should be used to update the CRC and parity of a module every time a module is "patched" or modified in any way. OS-9 will not recognize a module with an incorrect CRC.

For more information concerning CRC and parity, refer to the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

OPTIONS:

-?	Displays the usage of FIXMOD.
-u	Updates an invalid module CRC or parity.
-x	Looks for the module in the execution directory.
-z	Reads the module names from standard input.
-z=<file>	Reads the module names from <file>.

EXAMPLES:

$ fixmod dt	Checks parity and CRC for module "dt".
$ fixmod dt -u	Checks parity and CRC for module "dt" and updates them if necessary.

FORMAT BASIC UTILITY COMMAND FORMAT

Initialize disk media

SYNTAX: FORMAT [<opts>] <devname> [<opts>]

FUNCTION: FORMAT is used to physically initialize, verify, and establish an initial file structure on a disk. All disks must be formatted before they can be used on an OS-9 system.

EXECUTION: Type "format" followed by the device name and any options wanted. A description of the disk is automatically read from the device descriptor module. Default values for the number of sides, number of tracks, sector size, and density are determined by the values in the descriptor. At this time, cluster size is set at one. The <opts> may be used to override these default values. FORMAT will ask for any required options not given on the command line. FORMAT can be used to format almost any type of disk, including hard disks.

OPTIONS: For use with floppy disks:

-sd Single density disk

-dd Double density disk

-ss Single sided disk

-ds Double sided disk

For use with floppy or hard disks,

-c=<num> Number of sectors per cluster. <num> must be decimal and must be a power of 2. The default=1.

-i=<num> Number for sector interleave value (decimal).

-np No physical format.

-nv No physical verify.

-r Inhibit ready prompt.

-t=<num> Number of tracks (decimal).

-v=<name> Disk name (32 characters maximum). NOTE: If the name contains blanks, the option and name must be enclosed by quotation marks: "-v=Name of disk"

(continued)

The formatting process works as follows:

1. The disk surface is physically initialized and sectored.

2. Each sector is read back and verified. If the sector fails to verify after several attempts, the offending sector is excluded from the initial free space on the disk. As the verification is performed, track numbers are displayed on the standard output device.

3. The disk allocation map, root directory and identification sector are written to the first few sectors of track zero. These sectors can not be defective.

NOTE: If the disk is to be used as a system disk, OS9GEN must be run to create the bootstrap after the disk has been formatted.

EXAMPLES:

```
$ format /D1 -dsdd -v="database" -t=77

$ format /D1 -sssd -r
```

```
$ format /D1

                DISK FORMATTER  1.1
 ------------------- Format Data --------------------

Fixed values:
               Disk type: 8" floppy  <- disk size
           Sectors/track: 16         <- set by manufacturer
     Track zero sect/trk: 16         <- set by manufacturer
 Minimum sect allocation: 32

Variables :
         Recording format: FM         <- density; FM=single,MFM=double
     Track density in TPI: 48         <- some 5 inch disks use 96 TPI
      Number of cylinders: 77         <- 77 for 8"; 35/40/80 for 5"
       Number of surfaces: 2
Sector interleave offset: 3

Formatting device /D1
Proceed?
```

FORMAT BASIC UTILITY COMMAND FORMAT

(continued)

The values in the variables section can be changed by command line parameters or by answering "n" to the prompt. The values in the "Fixed values" section can only be changed by altering the device descriptor module of the specific unit.

FREE BASIC UTILITY COMMAND FREE

Display free space remaining on mass-storage device

SYNTAX: FREE [<opts>] [<devname> [<opts>]]

FUNCTION: FREE displays the number of unused 256-byte sectors on a device which are available for new files or for expanding existing files. FREE also displays the disk's name, creation date and cluster size.

EXECUTION: Type "free" followed by the name of the device to be examined. The device name must be the name of a mass-storage, multifile device.

Data sectors are allocated in groups called "clusters". The number of sectors per cluster depends on the storage capacity and physical characteristics of the specific device. This means that small amounts of free space (given in sectors) may not be divisible into the same number of files.

For example, a given disk system uses 8 sectors per cluster. A FREE command shows that it has 32 sectors free. Because memory is allocated in clusters, a maximum of four new files could be created even if each had only one sector.

OPTIONS: -? Displays the usage of FREE.

-b=<num> Uses the specified buffer size

EXAMPLES:

```
$ free
BACKUP DATA DISK created on: 80/06/12
Capacity: 1,232 sectors (1-sector clusters)
1,020 free sectors, largest block 935 sectors

$ free /D1
OS-9 Documentation Disk created on: 81/04/13
Capacity: 1,232 sectors (1-sector clusters)
568 Free sectors, largest block 440 sectors
```

GREP PROGRAMMER UTILITY COMMAND GREP

Search a file for a pattern

SYNTAX: GREP [<opts>] [<expression>] [[<path>] [<opts>]]

FUNCTION: GREP searches the input pathlist(s) for lines matching <expression>.

EXECUTION: Type "grep", the expression to be searched for, and the pathlist of the file to be searched. If no <path> is specified, GREP searches standard input.

If GREP finds a line that matches <expression>, the line is written to the standard output with an optional line number of where it is located within the file. When multiple files are searched, the output will have the name of the file preceding the occurrence of the matched expression.

EXPRESSIONS:

An <expression> is used to specify a set of characters. A string which is a member of this set is said to match the expression. To facilitate the creation of expressions, some metacharacters are defined to enable the creation of complex sets of characters. These special characters are:

CHAR	NAME:	DESCRIPTION
.	ANY.	"." is defined to match any ASCII character except new line.
*	CLOSURE.	"*" is defined to modify the preceding single character expression, so that it will match ZERO or more occurrences of the single character. If a choice is available the longest such group is chosen.
[]	CHARACTER CLASS.	"[]" defines a group of characters which will match any single character in the compare string. GREP recognizes certain abbreviations to aid the entry of ranges of strings: [a-z] is equivalent to the character string "abcdefghijklmnopqrstuvwxyz". [m-pa-f] is equivalent to the character string "mnopabcdef". [0-7] is equivalent to the character string "01234567".

(continued)

CHAR	NAME:	DESCRIPTION
-		BOL or NEGATE. "-" is defined to modify a character class (as described above) when between []. At the beginning of an entire expression, it requires the expression to compare and match the string at only the beginning of the line. The NEGATE character modifies the character class so that it will match any ASCII character NOT in the given class or newline.
$		EOL. "$" is defined to require the expression to compare and match the string only when at the end of line.
\		ESCAPE. "\" is defined to remove special significance from special characters. It is followed by a base and a numeric value or a special character. If no base is specified, the base for the numeric value will default to hexadecimal. An explicit base of decimal or hexadecimal can be specified by preceding the numeric value with a qualifier of "d" or "x", respectively. It also allows entry of some non-printing characters such as: \t = tab char \n = new-line char \l = line feed char \b = backspace char \f = form feed char

EXAMPLE EXPRESSIONS

Any combination of metacharacters and normal characters can be combined to create an expression:

Expression	Same as
abcd	abcd
ab.d	abcd, abxd, ab?d, etc.
"ab "d"	"abd", "ab d", "ab d", "ab d", etc.

(continued)

Expression	Same as
^abcd	abcd (only if very first chars on a line)
abcd$	abcd (only if very last chars on a line)
^abcd$	abcd (only if abcd is the complete line)
[a-z]bcd	abcd, bbcd, cbcd, ... zbcd
[Aa]bcd	abcd, Abcd
abcd[0-9a-zA-z]	abcd followed by any alpha-numeric character
abcd[a-d]	abcd followed by a, b, c, or d
bcd[^a-d]	bcd followed by any ASCII char except a, b, c, d, or new line

OPTIONS:

-? Displays the usage of GREP.

-c Counts the number of matching lines.

-e=<expr> searches for <expr> (Same as <expression> in the command line).

-f=<path> Reads the list of expressions from <path>.

-l Prints only the names of the files with matching lines.

-n Prints the relative line number within the file followed by the matched expression.

-s Does not display matching lines (Silent mode).

-v Prints all lines except for those that match.

-z Reads the file names from standard input.

(continued)

NOTES: Options "-l" and "-n" can not be used at the same time.
Options "-n" and "-s" can not be used at the same time.

EXAMPLES:

$ grep xyz myfile	Writes all lines of "myfile" that contain occurrences of "xyz" to standard output.
$ grep -f=words myfile -nc	Searches "myfile" for expressions input from "words", counts the number of matches, and gives the line number found with each occurrence.

IDENT SYSTEM MANAGEMENT UTILITY COMMAND IDENT

Print OS-9 module identification

SYNTAX: IDENT [<opts>] [<modname> [<opts>]]

FUNCTION: IDENT is used to display module header information, and additional information that follows the header, from OS-9 memory modules.

EXECUTION: Type "ident", followed by the module name(s) to be examined. IDENT displays the following information (in this order):

-module size.

-CRC bytes (with verification).

-owner.

-header parity (with verification).

-edition.

-type/language, and attributes/revision.

(for program modules it also includes):

-execution offset.

-data size.

-stack size.

-initialized data offset.

-offset to the data reference lists.

IDENT will also print the interpretation of the type/language and attribute/revision bytes.

OPTIONS:
-? Displays the usage of IDENT.

-m Searches for module in memory.

-x Searches for module in the execution directory.

-z Reads the module names from standard input.

-z=<file> Reads the module names from <file>.

(continued)

EXAMPLE:

```
$ ident -m ident
Header for:       ident
Module size:      $250C       #9692
Module CRC:       $5DC5F6     Good CRC
Owner:            0.0
Header parity:    $EFF8       Good parity
Edition:          $1          #1
Ty/La At/Rev:     $101        $8001
Exec. off:        $30         #48
Data size:        $7C6        #1990
Stack size:       $C00        #3072
Init. data off:   $203A       #8250
Data ref. off:    $25B0       #9648
Prog mod, 68000 obj, Sharable
```

INIZ SYSTEM MANAGEMENT UTILITY COMMAND INIZ

Attach modules

SYNTAX: INIZ [<opts>] [<devname> [<opts>]]

FUNCTION: INIZ performs the system call I$Attach on each device name passed to it.

EXECUTION: Type "iniz" followed by the name(s) of the device(s) to be "attached" to the system. OS-9 searches the system module directory (using the device's name) to see if the device is already attached. If not, an initialization routine is called that links the device to the system. If the device is already attached, it will not be re-initialized, but the link count will be incremented.

The device names may be listed on the command line, read from standard input, or read from a specified pathlist.

OPTIONS:

-?	Displays the usage of INIZ.
-z	Reads the device names from standard input.
-z=<file>	Reads the device names from <file>.

EXAMPLES:

$ iniz h0 term	Increments the link counts of modules h0 and term.
$ iniz -z	Increments the link count of any modules with names which are read from standard input.
$ iniz -z=/h0/file	Increments the link count of all modules whose names are supplied in "/h0/file1".

KILL BASIC UTILITY COMMAND KILL

Abort a process

SYNTAX: KILL [<procID>]

FUNCTION: KILL is a SHELL built-in command that sends an abort signal to the process having the process ID number specified.

EXECUTION: Type "kill", followed by the ID number(s) of the process(es) to be aborted. The process must have the same user ID as the user that executes the command. The PROCS command can be used to obtain the process ID numbers.

NOTE: If a process is waiting for I/O, it may not die until it completes the current I/O operation. Therefore, if you KILL a process and the PROCS command shows it still exists, it is probably waiting to receive a line of input data before it can die.

NOTE: This is a built-in SHELL command. It does not appear in the CMDS directory.

EXAMPLES:

```
$ kill 5

$ kill 22 27
```

LINK SYSTEM MANAGEMENT UTILITY COMMAND LINK

Link module into memory

SYNTAX: LINK [<opts>] [<modname> [<opts>]]

FUNCTION: LINK is used to "lock" a previously loaded module into memory.

EXECUTION: Type "link" followed by the name(s) of the module(s) to be locked into memory. The link count of the module specified is incremented by one each time it is LINKed. The UNLINK command is used to "unlock" the module when it is no longer needed.

OPTIONS: -? Displays the usage of LINK.

-z Reads the file names from standard input.

-z=<file> Reads the file names from <file>.

EXAMPLES:

```
$ link prog1 prog2 prog3
$ link -z=linkfile          Link modules from linkfile
$ link -z                   Link modules from standard input
```

LIST BASIC UTILITY COMMAND LIST

List the contents of a text file

SYNTAX: LIST [<opts>] [<path> [<opts>]]

FUNCTION: LIST copies text lines from the path(s) given to standard output.

EXECUTION: Type "list" and the pathlist. The program terminates upon reaching the end-of-file of the last input path. If more than one path is specified, the first path will be copied to standard output, the second path will be copied next, etc.

LIST is most commonly used to examine or print text files.

OPTIONS: -? Displays the usage of LIST.

-z Reads the file names from standard input.

-z=<file> Reads the file names from <file>.

EXAMPLES:

```
$ list /d0/startup >/P&          The "startup" listing is redirected
                                 to the printer in the background.

$ list /D1/user5/document /d0/myfile /d0/Bob/text

                                 Copies text from files to standard
                                 output in the same order as the
                                 command line.

$ dir -u ! list -z               Lists all files in the current data
                                 directory.

$ list -z=namefile               Reads the name(s) of the file(s) to
                                 be listed from "namefile" and lists
                                 their contents.
```

LOAD PROGRAMMER UTILITY COMMAND LOAD

Load module(s) from file into memory

SYNTAX: LOAD [<opts>] {<path> [<opts>]}

FUNCTION: This command loads one or more modules (specified by <path>) into memory.

EXECUTION: Type "load" followed by the pathlist(s) of the module(s) to be loaded into memory. The names of the modules are added to the module directory. If a module is loaded that has the same name and type as a module already in memory, the module having the highest revision level is kept. The modules are loaded from the current execution directory.

OPTIONS: -? Displays the usage of LOAD.

-d Loads the file from your current data directory, instead of your current execution directory.

-z Reads the file names from standard input.

-z=<file> Reads the file names from <file>.

EXAMPLE:

```
$ mdir
   Module Directory at 14:44:35
kernel      init        p32clk      rbf         p32hd
h0          p32fd       d0          d1          ram
r0          dd

$ load edit

$ mdir
   Module Directory at 14:44:35
kernel      init        p32clk      rbf         p32hd
h0          p32fd       d0          d1          ram
r0          dd          edit
```

LOGIN SYSTEM MANAGEMENT UTILITY COMMAND LOGIN

Timesharing System Log-In

SYNTAX: LOGIN [<name>] [,] [<password>]

FUNCTION: LOGIN is used in timesharing systems to provide login security. It is automatically called by the timesharing monitor TSMON or can be used after initial login to change a terminal's user.

EXECUTION: After the system has been booted, hit <carriage return>, or (if LOGIN is to be used after the initial user login) type "login".

LOGIN requests a user name and password, which is checked against a validation (password) file. If the information is correct, the user's system priority, user ID, and working directories are set up according to information stored in the file, and the initial program specified in the password file is executed (usually SHELL). The date, time, and process number (which is not the same as the user ID) are also displayed. If the user can not supply a correct user name and password after three attempts, the process is aborted.

NOTE: If the SHELL from which LOGIN was called will not be needed again, it may be discarded by using the EX command to start the LOGIN command: ex login.

LOGGING OFF THE SYSTEM

To log off the system, the initial program specified in the password file must be terminated. For most programs (including SHELL) this may be done by typing an end-of-file character (escape) as the first character on a line.

DISPLAYING A "MESSAGE-OF-THE-DAY"

If desired, the file "motd" (located in the SYS directory) will cause LOGIN to display its contents on the user's terminal after successful login. This file is not required for LOGIN to operate.

THE PASSWORD FILE

The password file must be present in the directory "/d0/SYS". The file contains one or more variable-length text records, one for each user name. Each record has the following fields, which are delimited by commas:

(continued)

1. User name (up to 32 characters, including spaces). If this field is empty, any name will match.

2. Password (up to 32 characters, may include spaces). If this field is omitted, no password is required for the specified user.

3. Group.User (ID) number (allows 0 to 65535 groups and 0 to 65535 users, 0.0 is superuser). This number is used by the file security system as the system-wide user ID to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user.

4. Initial process (CPU time) priority: 1 - 255.

5. Pathlist of initial execution directory (usually /d0/CMDS).

6. Initial data directory pathlist (specific user directory).

7. Name and parameters of initial program to execute (usually SHELL).

NOTE: This is not a SHELL command line.

SAMPLE PASSWORD FILE:

```
superuser,secret,0,255,.,.,shell
steve,open sesame,3,128,.,/d1/STEVE,shell
mary,jo,10,100,/d0/BUSINESS,/d1/LETTERS,wordprocessor
robert,,4,128,.,/d1/ROBERT,Basic09
```

OPTIONS: -? Displays the usage of LOGIN.

MAKDIR BASIC UTILITY COMMAND MAKDIR

Create directory file

SYNTAX: MAKDIR [<opts>] [<dir> [<opts>]]

FUNCTION: MAKDIR creates a new directory file specified by the given pathlist.

EXECUTION: Type "makdir", followed by the pathlist specifying the new directory. The user must have write permission for the new directory's parent directory. The new directory is initialized and initially does not contain files except for the "." and ".." pointers to itself and its parent directory, respectively. All access permissions are enabled except single user (non sharable).

It is an OS-9 convention (not required) to capitalize directory names to distinguish them from lower case (also a convention) file names.

OPTIONS:

-?	Displays the usage of MAKDIR.
-z	Reads the directory names from standard input.
-z=<file>	Reads the directory names from <file>.

EXAMPLES:

```
$ makdir /d1/STEVE/PROJECT

$ makdir DATAFILES

$ makdir ../SAVEFILES

$ makdir RED GREEN BLUE ../PURPLE
```

MAKE PROGRAMMER UTILITY COMMAND MAKE

Maintain, update, and regenerate groups of programs

SYNTAX: MAKE [<opts>] [<target1>] {[<target2>] [<macros>]}

FUNCTION: The purpose of MAKE is to ease the burden of creating executable modules that consist of several parts.

EXECUTION: Type "make", followed by the name of the module to be created. MAKE executes commands from a special type of procedure file called a makefile. The makefile describes the "dependencies" between files that make up a target program.

For example, an executable program could be made up of one or more relocatable object files. These files could be made up of one or more source files. The makefile for this program resolves the dependent relationships between itself and the files that make it.

A makefile contains a sequence of entries that specify dependencies and commands to resolve the dependencies. A "dependency entry" begins with the target name of the file or module followed by a ':'. This is followed by a list of files that are prerequisites to building the target file (this is called a dependency list).

In addition to the dependency entry, the makefile may contain commands on how to update a particular target file (if it needs to be updated). MAKE updates a target file only if it depends on files that are newer than itself. If no instructions for update are provided, MAKE will attempt to create a command line to perform the operation. MAKE recognizes a command line by a line beginning with one or more spaces or tabs.

EXAMPLE ONE: A SAMPLE MAKEFILE

```
program: xxx.r yyy.r
    cc xxx.r yyy.r -xf=program
xxx.r: xxx.c /d0/defs/oskdefs.h
    cc xxx.c -r
yyy.r: yyy.c /d0/defs/oskdefs.h
    cc yyy.c -r
```

This makefile specifies that "program" is made up of two ".r" files: "xxx.r" and "yyy.r". They are dependent upon "xxx.c" and "yyy.c" respectively and both are dependent on the "oskdefs.h" file.

If either "xxx.c" or "/d0/defs/oskdefs.h" has a more recent date than "xxx.r", the command "cc xxx.c -r" is executed. If "yyy.c" or "/d0/defs/oskdefs.h" is newer than "yyy.r", then "cc yyy.c -r" is executed. If either of the former commands are executed, the command "cc xxx.r yyy.r -xf=program" will also be executed.

IMPLICIT RULES AND DEFINITIONS

The following definitions and rules allow greater versatility in creating a makefile:

object files:	Files that have no suffixes. Object files are made from a relocatable file and is linked when it needs to be made.
source files:	Files that have one of the following suffixes: ".a", ".c", ".f" or ".p"
relocatable files:	Files appended by the suffix: ".r". Relocatable files are made from source files and are assembled or compiled if they need to be made.
default compiler:	"cc"
default assembler:	"r68"
default linker:	"cc" NOTE: The default linker should only be used with programs that want to use Cstart.

default directory for all files: "."

MACROS

Macros may be placed in the makefile for convenience or on the command line for flexibility. Macros are allowed in the form of <macro name> = <expansion>. Everywhere that the macro name appears, the expansion will be substituted for it.

MAKE recognizes a macro by the "$" character in front of the name. If a macro name is longer than a single character, the entire name must be surrounded by parentheses. For example, $R refers to the macro "R", $(PFLAGS) refers to the macro "PFLAGS", $(B) and $B refer to the macro "B", and $BR is interpreted as the value for the macro "B" followed by the character "R".

NOTE: If you define a macro in your makefile and then redefine it on the command line, the command line definition will override the definition in the makefile. This feature can be useful for compiling with special options.

SPECIAL MACROS

In order for MAKE to be more flexible and not so verbose, there are some special macros that can be defined in the makefile that are used by MAKE to determine what to do. They are:

MACRO	DEFINITION
ODIR = <path>	MAKE searches this directory (<path>) for all files that have no suffix or relative pathlist.
SDIR = <path>	MAKE searches this directory (<path>) for all implicitly defined source files. If SDIR is not defined in the makefile, MAKE searches the current directory (default).
RDIR = <path>	MAKE searches this directory (<path>) for all implicitly defined relocatable files. If RDIR is not defined, MAKE searches the current directory (by default).
CFLAGS = <opts>	MAKE uses these compiler options to generate its own command line.
RFLAGS = <opts>	MAKE uses these assembler options to generate its own command line.
LFLAGS = <opts>	MAKE uses these linker options to generate its own command line.

(continued)

RESERVED MACROS

There are some reserved macros that are expanded when a command line associated with a particular file dependency is forked. These macros may only be used on a command line. In practice, they are wildcards that have the following meanings:

MACROS	DEFINITION
$@	Expands to the name of the file to be made by the command
$*	Expands to the prefix of the file to be made.
$?	Expands to the list of files that were found to be newer than the target on a given dependency line.

These macros are very useful when you need to be explicit about a command line but have a target program with several dependencies.

HOW MAKE WORKS

MAKE starts by reading the makefile and setting up a table of dependencies exactly as it is listed in the makefile. When the parser encounters a name on the left side of a colon, it first checks to see if it has encountered the name before. If it has, it connects the lists and continues. It treats every list on the right side of the colon as a unique structure.

After reading the entire makefile, MAKE determines the target file (the main file to be made) on the list. It then makes a second pass through the subtable. It looks for object files that have no relocatable files in their dependency lists, and for relocatable files that have no source files in their dependency lists.

If MAKE needs to find any source files or relocatable files to complete the dependency lists, it looks for them in the directory specified by the macros: "SDIR" and "RDIR" (or "." - default). It looks in these directories for files with the same name as their dependent file. For example, if no source file is found for program.r, MAKE will search the specified directory (RDIR or ".") for program.a (or ".c", ".p", ".f").

MAKE does a third pass through the list to get the file dates and compare them. When MAKE finds a file that is newer than its dependent file, it will generate the necessary command or execute the command given. Due to the fact that OS-9 only stores the time down to the closest minute, MAKE will remake a file if its date matches one of its dependents.

WARNING:

When MAKE is generating a command line for the linker, it looks at its list and uses the first relocatable file that it finds, but only the first one. For example:

```
prog: x.r y.r z.r
```

would generate

"cc x.r", not "cc x.r y.r z.r" or "cc prog.r"

COMMANDS

More than one command can be given for any dependency. Each command will be forked separately unless it is continued from the previous command with a "\".

If your system runs out of memory while executing a command, you can redirect the output of MAKE into a procedure file and execute the procedure file.

Comments should not be interspersed with commands.

COMMENTS AND LONG LINES

All characters following a "#" character are ignored as comments. An asterisk ("*") may be used as a comment line if it is the first character in the line.

Blank lines are ignored.

If you use lines that are longer than 256 characters or wider than your screen, you need to place a space followed by a "\" at the end of each line to be continued. If a command line is continued, a space or tab must be the first character in the continued line. With non-command lines, leading spaces and tabs are ignored on continuation lines. For example:

```
FILES = aaa.r bbb.r ccc.r ddd.r eee.r fff.r ggg.r \
        hhh.r iii.r jjj.r
```

EXAMPLE TWO: REFINING THE SAMPLE MAKEFILE

Knowing how MAKE works and understanding the implicit rules can simplify coding immensely:

```
program: xxx.r yyy.r
   cc xxx.r yyy.r -xf=program
xxx.r yyy.r: /d0/defs/oskdefs
```

Example Two makes use of MAKES awareness of file dependencies. No mention is made of the C language files, therefore, MAKE looks in the directory specified by the macro definition "SDIR = <path>" and adjusts the dependency list accordingly (in this case, by default, the current directory). MAKE will also generate a command line to compile "xxx.r" and "yyy.r" if one or both needs to be updated.

Further simplification would be possible, if "program" was made up of only one source file:

```
program:
```

MAKE will make the following assumptions from this simple command:

1. "program" has no suffix. It is an object file and therefore needs to rely on relocatable files to be made.

2. No dependency list is given, therefore MAKE creates an entry in the table for "program.r".

3. After creating an entry for "program.r", it creates the entry for a source file connected to the relocatable file.

Assuming it found "program.a", it will check the dates on the various files and generate one or both of the following commands if required:

```
r68 program.a -o=program.r  ( + RFLAGS if used)

cc program.r  ( + LFLAGS if used)
```

EXAMPLE THREE: SAMPLE MAKEFILE USING MACROS

Using these inherent features of MAKE can be especially helpful if you have several object files that you want MAKE to check:

```
* beginning
ODIR = /d0/cmds
RDIR = rels
UTILS = attr copy load dir backup dsave
SDIR = ../utils/sources

utils.files: $(UTILS)
     touch utils.files

* end
```

MAKE will look in "rels" for "attr.r", "copy.r", "load.r", etc. and look in "../utils/sources" for "attr.c", "copy.c", etc. MAKE will then generate the proper commands to compile and/or link any of the programs that need to be made. If one of the files in UTILS is made, then the command "touch utils.files" will be forked to maintain a current overall date.

EXAMPLE FOUR: PUTTING IT ALL TOGETHER

The following example is a makefile to create MAKE:

```
* beginning
ODIR = /h0/cmds
RDIR = rels
CFILES = domake.c doname.c dodate.c domac.c
RFILES = domake.r doname.r dodate.r
PFLAGS = -p64 -nh1
R2 = ../test/domac.r
RFLAGS = -q
make: $(RFILES) $(R2) getfd.r
    linker
$(RFILES): defs.h
$(R2): defs.h
    cc $*.c -r=../test
print.file: $(CFILES)
    pr $? $(PFLAGS) >-/p1
    touch print.file
*end
```

The makefile in Example Four looks for the ".r" files (listed in RFILES) in the directory specified by RDIR: "rels". The only exception is "../test/domac.r", which has a complete pathlist specified (MAKE will only look there).

Even though "getfd.r" does not have any explicit dependents, it's dependency on getfd.a is still checked. All of the source files are found in the current directory.

Notice that this makefile can be used to make listings, also. By typing "make print.file" on the command line, MAKE will expand the macro "$?" to mean all of the files that were updated since the last time "print.file" was updated. If you keep a dummy file called "print.file" in your directory, it will only print out the newly made ones. If no "print.file exists", it will print all of the files.

OPTIONS:

-?	Displays the usage of MAKE.
-b	Does not use built in rules.
-d	Prints the dates of the files in makefile (Debug mode).
-f=<path>	Specifies <path> as the makefile. If <path> is specified as "-", MAKE commands are read from standard input.
-i	Ignores errors.
-n	Does not execute commands, but does display them.
-s	Executes commands without echo--Silent mode.
-t	Updates the dates without executing commands.
-u	Does the MAKE regardless of the dates on files.
-x	Uses the cross-compiler/assembler.
-z	Reads a list of MAKE targets from standard input.
-z=<path>	Reads a list of MAKE targets from <path>.

Options may be included on the command line when you run MAKE, or they may be included in the makefile for convenience.

EXECUTING THE COMMAND

When a command is executed it is echoed to standard output, unless the "-s" (silent) option is used or the command line starts with a "@". When the "-n" option is used, the command is echoed to standard output but not actually executed.

MAKE will normally stop if an error code is returned when a command line is executed. Errors are ignored if the "-i" option is used or a command line begins with a hyphen.

DEBUG MODE

Sometimes, it is helpful to see the file dependencies and the dates associated with each of the files in the list. The "-d" option turns on the MAKE debugger and gives a complete listing of the macro definitions, a listing of the files as it checks the dependency list, and all the file modification dates. If it can't find a file to examine its date, it assumes a date of -1/00/00 00:00, indicating the necessity to update the file.

TOUCHING FILES

If you want to update the date on a file, but don't want to remake it, you can use the "-t" option. MAKE merely opens the file for update and then closes it, thus making the date current.

TURNING OFF THE RULES

If you are quite explicit about your makefile dependencies, and don't want MAKE to assume anything, you may use the "-b" option to turn off the built-in rules governing implicit file dependencies.

CAVEATS

If an object has more than one dependency, the dependency lists will be linked together. If the first dependency lists multiple objects, then all the objects on that dependency line will share the same set of dependencies. This may or may not be correct, depending on the situation. For example (the first makefile is correct, and the second creates some extra dependencies):

```
First makefile:      x.r: defs.h
                     x.r y.r z.r: defs2.h

Second makefile:     x.r y.r z.r: defs2.h
                     x.r: defs.h
```

The first makefile specifies that "x.r" is dependent on "defs.h" and "defs2.h". It specifies "y.r" and "z.r" as dependent on "defs2.h". The second makefile specifies that all three ".r" files are dependent on "defs2.h", and seems to specify only "x.r" as dependent on "defs.h".

In the second makefile, "x.r", "y.r" and "z.r" are all listed together on the same dependency line. They will therefore also implicitly share in any future dependencies for any of the individual files. Thus, "x.r", "y.r" and "z.r" are all implicitly dependent on "defs.h".

NOTE: The MAKE language is very specific, therefore you need to be careful when you are using dummy files with names like "print".

Unless a file is specifically an object file or you use the "-b" option to turn off the implicit rules, it is recommended to use a suffix for your dummy files (i.e. "print.file" and "xxx.h" for your header files).

MDIR SYSTEM MANAGEMENT UTILITY COMMAND MDIR

Display Module Directory

SYNTAX: MDIR [<opts>] [<name>]

FUNCTION: MDIR displays the present module names in the system module directory (all modules currently resident in memory). By specifying individual module names, only those modules specified will be displayed (if resident in memory).

If the "-e" option is given, a extended listing of the physical address, size, owner, revision level, user count and the type of each module is displayed.

The module type will be listed using the following mnemonics:

MNEMONIC	TYPE OF MODULE
Prog	Program module
Subr	Subroutine module
Mult	Multi module
Data	Data module
Trap	Trap handler module
Sys	System module
FMan	File manager
Driv	Device driver module
Desc	Device descriptor module

NOTE: User defined modules that do not correspond with this list will be displayed by their number.

By using the "-a" option, the language of each module will be displayed instead of the type (in an extended listing). The language field will use the following mnemonics:

MNEMONICS	MODULE LANGUAGE
Obj	68000 machine code
Bas	Basic09 I code
Pasc	Pascal I code
C	C I code
Cobl	Cobol I code
Fort	Fortran I code

NOTE: If the language field is inappropriate for the module (for example, d0, t1 or init) a blank field will be displayed.

MDIR SYSTEM MANAGEMENT UTILITY COMMAND MDIR

(continued)

WARNING: Not all modules listed by MDIR are executable as processes: always check the module type code to make sure it is executable, before executing a module if you are not familiar with it.

OPTIONS:
- -? Displays the usage of MDIR.
- -a Displays language field instead of type field in an extended listing.
- -e Displays the extended module directory.
- -t=<type> Displays only the modules of the specified type.
- -u Displays an unformatted listing (used for piping the output etc.).

EXAMPLES:

NOTE: To save space, the following examples are fairly incomplete. Module directories are generally much larger.

```
$ mdir
   Module Directory at 14:44:35
kernel      init        p32clk      rbf         p32hd
h0          p32fd       d0          d1          ram
r0          dd

$ mdir -e

 Addr     Size      Owner     Perm Type Revs  Ed #  Lnk  Module name
------ -------- ----------- ---- ---- ---- ----- ----- -------------
002600    12136     12.3     ffff Sys  8000     9     0 kernel
005568      218     12.3     ffff Sys  8000     2     2 init
005642      172     12.3     ffff Sys  8000     4     3 p32clk
0056ee     5548     12.3     ffff Fman 8000    14     7 rbf
006c9a     1806     12.3     ffff Driv 8004     6     3 p32hd
0073a8      110     12.3     ffff Desc 8000     1     3 h0
007416     1608     12.3     ffff Driv 8001     8     2 p32fd
007a5e      110     12.3     ffff Desc 8000     1     1 d0
007acc      110     12.3     ffff Desc 8000     1     1 d1
007b3a      462     12.3     ffff Driv 8000     3     2 ram
007d08      106     12.3     ffff Desc 8000     1     1 r0
007d72      106     12.3     ffff Desc 8000     1     2 dd
```

(continued)

EXAMPLES (continued):

```
$ mdir -ea
```

Addr	Size	Owner	Perm	Lang	Revs	Ed #	Lnk	Module name
002600	12136	12.3	ffff	Obj	8000	9	0	kernel
005568	218	12.3	ffff		8000	2	2	init
005642	172	12.3	ffff	Obj	8000	4	3	p32clk
0056ee	5548	12.3	ffff	Obj	8000	14	7	rbf
006c9a	1806	12.3	ffff	Obj	8004	6	3	p32hd
0073a8	110	12.3	ffff		8000	1	3	h0
007416	1608	12.3	ffff	Obj	8001	8	2	p32fd
007a5e	110	12.3	ffff		8000	1	1	d0
007acc	110	12.3	ffff		8000	1	1	d1
007b3a	462	12.3	ffff	Obj	8000	3	2	ram
007d08	106	12.3	ffff		8000	1	1	r0
007d72	106	12.3	ffff		8000	1	2	dd

MERGE BASIC UTILITY COMMAND MERGE

Copy and Combine Files to Standard Output

SYNTAX: MERGE [<opts>] [<path> [<opts>]]

FUNCTION: MERGE copies multiple input files specified by <path> to the standard output. It is commonly used to combine several files into a single output file.

EXECUTION: Type "merge" and the pathlist of the input file(s) to be copied. Data is copied in the order the pathlists are given. MERGE does no output line editing (such as automatic line feed). The standard output is generally redirected to a file or device.

OPTIONS:

-? Displays the usage of MERGE.

-b=<num> Allocates <num> k buffer size for use by MERGE (default memory size: 4k).

-x Searches current execution directory for files to be merged.

-z Reads the file names from standard input.

-z=<file> Reads the file names from <file>.

EXAMPLES:

```
$ merge compile.list asm.list >/p

$ merge file1 file2 file3 file4 >combined.file -b=32k

$ merge -x load link copy >Utils1

$ merge -z=/h0/PROGS/file1 >merged_files
```

MFREE BASIC UTILITY COMMAND MFREE

Display Free System RAM

SYNTAX: MFREE [<opts>]

FUNCTION: MFREE displays a list of areas in memory that are not presently in use and are available for assignment. The address and size of each free memory block are displayed.

OPTIONS: -? Displays the usage of MFREE.

-e Displays an extended free memory list.

EXAMPLE:

```
$ mfree

Current total free RAM:    #181504   $2C118

$ mfree -e

Minimum allocation size:       0.25 K-bytes
Number of memory segments      6
Total RAM at startup           256 K-bytes
Current total free RAM         171.75 K-bytes

Free memory map:

     Segment Address          Size of Segment
    -----------------    -------------------------
        $29100               $F00        3.75 K-bytes
        $2D100               $26E00    155.50 K-bytes
        $57800               $1600       5.50 K-bytes
        $59400               $100        0.25 K-bytes
        $59600               $1800       6.00 K-bytes
        $5B600               $300       85.00 K-bytes
```

OS9GEN SYSTEM MANAGEMENT UTILITY COMMAND OS9GEN

Build and Link a Bootstrap File

SYNTAX: OS9GEN [<opts>] <devname> [<path> [<opts>]]

FUNCTION: OS9GEN is used to create and link the "OS9Boot" file that is required on any disk from which OS-9 is to be bootstrapped. OS9GEN can be used to make a copy of an existing boot file, to add modules to an existing boot file or to create an entirely new boot file (for a different system, for example).

EXECUTION: Type "os9gen", followed by the name of the device on which the "OS9Boot" file is to be installed. OS9GEN creates a working file called "TempBoot" on the device specified. It then reads file names (pathlists) from the command line.

If the "-z" option is used, OS9GEN will first use the files specified on the command line and then the file names from its standard input, or the specified pathlist (one pathlist per line). If the names are entered manually, no prompts are given, and the end-of-file key (usually <escape>) or a blank line is entered after the line containing the last pathlist.

Every file named is opened and copied to "TempBoot". The process is repeated until all command line names have been copied. If the "-z" option is used, OS9GEN will proceed to copy files until an end-of-file or a blank line is reached.

The "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" contains a specification of what modules need to be contained in the boot file.

After all input files have been copied to "TempBoot", the old "OS9Boot" file, if present, is renamed "OldBoot". If an "OldBoot" file is already present, it is deleted before the "OS9Boot" is renamed.

"TempBoot" is then renamed to "OS9Boot". Its starting address and size are linked in the disk's Identification Sector (LSN 0) for use by the OS-9 bootstrap firmware.

WARNING: Any "OS9Boot" file must be stored in physically contiguous sectors. Therefore, OS9GEN is normally used on a freshly formatted disk. If the "OS9Boot" file is fragmented, OS9GEN will first try to copy the "TempBoot" file to a contiguous file named "Tmp2Boot". If this fails OS9GEN will print an error message indicating the disk can not be used to bootstrap OS-9.

(continued)

OPTIONS:
- -? Displays the usage of OS9GEN.
- -b=<size> Assigns <size>k of memory for OS9GEN. Default memory size is 4k.
- -q=<file> Sets sector zero pointing to <file> (Quick boot).
- -s=<size> Expands "TempBoot" to <size>.
- -x Searches the execution directory for pathlists.
- -z Reads the file names from standard input.
- -z=<file> Reads the file names from <file>.

The "-q" option updates information in the disk's Identification Sector by directing it to point to a file already contained in the root directory of the specified device. OS9GEN will not copy the file unless the specified file is not contiguous. In that case, OS9GEN will try to copy it to a contiguous file.

The "-q" option is useful when restoring the "OldBoot" file as the valid boot on the disk. OS9GEN renames the specified file to be "OS9Boot" and saves the current boot as described previously.

The "-s" option may be used to expand the "TempBoot" to a size large enough to hold the new boot file in a contiguous space on the disk. If the size of the boot is known, the "-s" option eliminates the need for OS9GEN to try to find contiguous space after having built the "TempBoot" file.

EXAMPLES:

This command manually installs a boot file on device "d1" which is an exact copy of the "OS9Boot" file on device "d0".

```
$ os9gen /d1 /d0/os9boot
```

If the size of /d0/os9boot was slightly less than 20K bytes, the "-s" option could be used to pre-expand the "TempBoot" file to be 20K contiguous bytes. If the expansion fails, OS9GEN will abort:

```
$ os9gen /d1 /d0/os9boot -s=20K
```

(continued)

All three methods below manually installs a boot file on device "d1" which is a copy of the "OS9Boot" file on device "d0" with the addition of modules stored in the files "/d0/tape.driver" and "/d2/video.driver":

```
method 1:  $ os9gen /d1 /d0/os9boot /d0/tape.driver /d2/video.driver

method 2:  $ os9gen /d1 /d0/os9boot -z
           /d0/tape.driver
           /d2/video.driver
           [ESCAPE]

method 3:  $ os9gen /d1 -z
           /d0/os9boot
           /d0/tape.driver
           /d2/video.driver
           [ESCAPE]
```

A boot file may automatically be installed by building a "bootlist" file and using the "-z" option (redirecting OS9GEN standard input or using the specified file as input):

```
$ build /d0/bootlist              Create file "bootlist"
? /d0/os9boot                     Enter first file name
? /d0/tape.driver                 Enter second file name
? /d2/video.driver                Enter third file name
? * V1.2 of video driver          Comment line
? [RETURN]                        Terminate "build"

$ os9gen /d1 -z </d0/bootlist     Redirects standard input

$ os9gen /d1 -z=/d0/bootlist      Reads input from pathlist
```

NOTE: OS9GEN treats any input line preceded by "*" as a comment.

The following command makes the "OldBoot" file the current boot and saves the current "OS9BOOT" file as "OldBoot":

```
$ os9gen /d1 -q=oldboot
```

PD BASIC UTILITY COMMAND PD

Print Working Directory

SINTAX: PD [<opts>]

FUNCTION: PD displays a pathlist that shows the path from the root directory to the user's current data directory. PD can be used by programs to discover the actual physical location of files, or by users to find their whereabouts in the file system. PD -x displays the pathlist from the root directory to the current execution directory.

OPTIONS:

-?	Displays the usage of PD.
-x	Displays the path to the current execution directory.

EXAMPLES:

```
$ chd /D1/STEVE/TEXTFILES/MANUALS

$ pd
/d1/STEVE/TEXTFILES/MANUALS

$ chd ..

$ pd
/d1/STEVE/TEXTFILES

$ chd ..

$ pd
/d1/STEVE

$ pd -x
/d0/CMDS
```

PR BASIC UTILITY COMMAND PR

Print Files

SYNTAX: PR [<opts>] {<path> [<opts>]}

FUNCTION: PR produces a listing of one or more files to the standard output.

EXECUTION: Type "pr" and the pathlist(s) of the files to be listed. The listing is separated into pages. Each page has the page number, the name of the listing, and the date and time printed at the top.

PR can produce multi-column output. It can also print files simultaneously, one per column. When printing multiple output columns. with the "-m" option, if an output line exceeds the column width, the output line will be truncated.

If no files are specified on the command line and the "-z" option is used, standard input is assumed to be a list of file names (one file name per input line) to be printed out. If no files are specified on the command line and the "-z" option is not used, standard input will be displayed on standard output.

Files and options may be intermixed.

A typical page of output will consist of 66 lines of output. PR handles this as 61 lines of output with 5 blank lines as a trailer. The 61 lines of output contains one line for the title, 5 blank lines for a header, and 55 lines of text. The trailer can be reduced or eliminated by expanding the number of lines per page.

OPTIONS: All "=" in option specifications are optional.

-? Displays the usage of PR.

-c=<char> Uses <char> as the specified column separator. <space> is the default column separator.

-f Pads the page using a series of "\n" (new line), instead of a "\f" (form feed).

-h=<num> Sets the number of blank lines after title line (default = 5).

-k=<num> Sets the <num> columns that the output file will be listed in for multi-column output.

-l=<num> Sets the left margin to <num> (default = 0).

(continued)

-m	Prints files simultaneously; one file per column. If three files are given on the command line, each file will be printed in its own column on the page.
-n=<num>	Specifies the line numbering increment: <num> (default = 1).
-o	Truncates lines that are longer than right margin (default: long lines are wrapped around to the next line).
-p=<num>	Specifies the number of lines per page: <num> (default = 61).
-r=<num>	Sets the right margin to <num> (default = 79).
-t	Does not print title.
-u=<title>	Uses specified title instead of file name.
-x=<num>	Sets the starting page number to <num> (default = 1).
-z	Reads the file names from standard input.
-z=<file>	Reads the file names from <file>.

EXAMPLES:

$ dir -u ! qsort ! pr -z	Displays a sorted output listing of a directory.
$ dir -u ! pr -n	Displays a numbered, unformatted listing of the data directory.

PRINTENV PROGRAMMER UTILITY COMMAND PRINTENV

Print environment parameters

SYNTAX: PRINTENV

FUNCTION: PRINTENV prints (to standard output) any environment parameters that have been set.

OPTIONS: None.

EXAMPLES:

```
$ printenv
'name=andy'
'term=abm85'
'list=/p1'
'As_long_as_you_want=long_value'
```

SEE ALSO: SETENV and UNSETENV utility command descriptions.

PROCS BASIC UTILITY COMMAND PROCS

Display Processes

SYNTAX: PROCS [<opts>]

FUNCTION: PROCS displays a list of processes running on the system that are owned by the user invoking the routine. The display is a "snapshot" taken at the instant the command is executed: processes can switch states rapidly, usually many times per second.

EXECUTION: "procs" (without any options) displays 10 pieces of information for each process:

Id: the process id.

PId: the parent process id.

Grp.usr: the owner of the process (group and user).

Prior: the initial priority of the process.

MemSiz: the amount of memory the process is using.

Sig: the number of any pending signals for the process.

S: the process status:

"w" = waiting
"s" = sleeping
"a" = active
"@" = currently executing

CPU Time: the amount of CPU time the process has used.

Age: the elapsed time since the process started.

Module & I/O: the process name and standard I/O paths:

"<" = standard input
">" = standard output
">>" =standard error output

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

(continued)

"PROCS -a" displays nine pieces of information: the process id, the parent process id, the process name and standard I/O paths and 6 new pieces of information:

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

The "-b" option displays all of the information mentioned above. The "-e" option will cause information for all processes in the system to be displayed.

Detailed explanation of all information displayed by PROCS is available in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

OPTIONS:

-?	Displays the usage of PROCS.
-a	Displays alternate information.
-b	Displays regular and alternate PROCS information.
-e	Displays all processes of all users.

EXAMPLES:

```
$ procs
 Id PId Grp.Usr Prior MemSiz Sig S     CPU Time   Age Module & I/O
  2   1   0.0    128   0.25k   0 w        0.01   ??? sysgo <>>>term
  3   2   0.0    128   4.75k   0 w        4.11 01:13 shell <>>>term
  4   3   0.0      5   4.00k   0 a    12:42.06 00:14 xhog <>>>term
  5   3   0.0    128   8.50k   0 *        0.08 00:00 procs <>>term >d0
  6   0   0.0    128   4.00k   0 s        0.02 01:12 tsmon <>>>t1
  7   0   0.0    128   4.00k   0 s        0.01 01:12 tsmon <>>>t2
```

EXAMPLES (continued):

```
$ procs -a
 Id PId  Aging  F$calls I$calls Last     Read Written Module & I/O
  2   1   129        5       1 Wait        0       0 sysgo <>>>term
  3   2   132      116     127 Wait      282     129 shell <>>>term
  4   3    11        1       0 TLink       0       0 xhog  <>>>term
  5   3   128        7       4 GPrDsc      0       0 procs <>>>term
  6   0   130        2       7 ReadLn      0       0 tsmon <>>>t1
  7   0   129        2       7 ReadLn      0       0 tsmon <>>>t2
```

QSORT PROGRAMMER UTILITY COMMAND QSORT

In-memory quick sort

SYNTAX: QSORT [<opts>] {<path> [<opts>]}

FUNCTION: QSORT is a quick sort algorithm that will sort any number of lines up to the maximum capacity of memory.

EXECUTION: Type "qsort" and the pathlist(s) of the file(s) to be sorted. QSORT will sort the file(s) by a user-specified field or field one (default). The field separation character may be specified or it will default to a space. If no file names are given on the command line, standard input is assumed.

CAVEAT: Multiple separation characters in a row are counted as a single field separator. For example, if a comma is specified as the field separation character, three commas in a row (,,,) will signify only one field separator. If the intent was to create two "null" fields, a space must be inserted between each comma (, , ,).

OPTIONS:

-?	Displays the usage of QSORT.
-c=<char>	Specifies the field separation character. If "*", "?" or "," are used as field separation characters, the option and the character must be enclosed by quotation marks.
-f=<num>	Specifies the sort field.
-z	Reads the file names from standard input.
-z=<file>	Reads the file names from <file>.

EXAMPLES:

$ qsort file1 file2 file3	Sort files and display.
$ dir -ue ! qsort -f=7	Sort extended dir listing by entry name (field 7).
$ qsort file -f=2 "-c=*"	Sort file by field 2 using "*" as field separation character.
$ qsort file -f=2 "-c=,"	Sort file by field 2 using "," as field separation character.
$ qsort -z	Read file names from standard input.

RENAME BASIC UTILITY COMMAND RENAME

Change file name

SYNTAX: RENAME [<opts>] <path> [<opts>] <new name>

FUNCTION: RENAME assigns a new name to the mass storage file specified in the pathlist.

EXECUTION: Type "rename", followed by the name of the file to be renamed, followed by the new name. The user must have write permission for the file to change its name. It is not possible to use the names "." or ".." for <path>.

OPTIONS:

-?	Displays the usage of RENAME.
-x	Indicates that <path> starts at current execution directory; the user must have execute permission for the specified file.

EXAMPLES:

```
$ dir

   Directory of .  16:22:53
myfile          blue

$ rename blue purple

$ dir

   Directory of .  16:23:22
myfile          purple

$ rename /h0/HARRY/test1 test2

$ rename -x screenclear clearscreen
```

SAVE PROGRAMMER UTILITY COMMAND SAVE

Save memory module(s) on a file

SYNTAX: SAVE [<opts>] {<modname> [<opts>]}

FUNCTION: SAVE copies the specified module(s) into your current data directory. The file(s) created in your directory will have the same name(s) as the specified module(s).

EXECUTION: Type "save", followed by the name(s) of the module(s) to be saved. <modname> must exist in the module directory when saved. The new file is given access permissions for all modes except public write.

If multiple modules are specified, each module will be stored in a separate file unless the "-f" option is used. In that case, all modules listed are saved into the specified file.

NOTE: SAVE's default directory is the current data directory. Executable modules should generally be saved in the default execution directory.

OPTIONS:

-?	Displays the usage of SAVE.
-f=<path>	Saves all specified modules to <path>.
-r	Rewrites existing files.
-x	Changes default directory to execution directory.
-z	Reads the module names from standard input.
-z=<file>	Reads the module names from <file>.

EXAMPLES:

```
$ save -x dir copy

$ save -f=/d1/math_pack add sub mul div
```

SET PROGRAMMER UTILITY COMMAND SET

Set Shell options

SYNTAX: SET [<opts>]

FUNCTION: SET changes SHELL options for the individual SHELL in which they are declared.

EXECUTION: Type "set" followed by the SHELL options desired. This command is the equivalent of typing the options directly after the SHELL prompt on the command line. This is a preferred method of changing SHELL parameters within procedure files because of its clarity.

The dash that usually procedes declared options is unnecessary when using the SET command.

The options specified by the SET command will change the SHELL parameters only in the SHELL in which they are declared. All descendant SHELLs will have the default parameters unless changed within the new SHELL.

NOTE: SET is a built-in SHELL command and therefore is not in the "CMDS" directory.

OPTIONS:

?	Displays the usage of SET.
e=<file>	Prints error messages from <file>. If no file is specified, the default file used is /dd/sys/errors. Without the "-e" option, SHELL will print only error numbers with no message description.
ne	Prints no error messages (default).
t	Echoes input lines.
nt	Does not echo input lines (default).
p	Displays prompt (default prompt: "$ ").
np	Does not display prompt.
p=<string>	Sets current SHELL prompt equal to <string>.
x	Aborts process upon error (default).
nx	Does not abort on error.

(continued)

EXAMPLES: All commands on the same line have the same effect:

```
$ set x              $ set -x              $ -x
$ set xp="JOE"       $ set -xp="JOE"       $ -xp="JOE"
```

SETENV PROGRAMMER UTILITY COMMAND SETENV

Set environment parameters

SYNTAX: SETENV <eparam> <evalue>

FUNCTION: SETENV sets environment parameters within a SHELL for use by the individual SHELL's child processes.

EXECUTION: Type "setenv", followed by <eparam> and <evalue>. <eparam> and <evalue> are strings that are stored in the environment storage area by SHELL. These parameters are known to the SHELL in which they are declared and descendent processes from that SHELL.

NOTE: SETENV should not be confused with the SHELL SET command. It has a completely different function. SET is a built-in SHELL command and therefore is not in the "CMDS" directory.

OPTIONS: None.

EXAMPLES:

```
$ setenv name andy

$ setenv term abm85

$ setenv list /p1

$ setenv As_long_as_you_want long_value
```

SETIME BASIC UTILITY COMMAND SETIME

Activate and set system clock

SYNTAX: SETIME [y m d h m s [am/pm]]

FUNCTION: This command sets the system date and time. Once set, it activates the real time clock.

EXECUTION: Type "setime". You may enter the year, month, day, hour, minute, second and "am" or "pm" as parameters on the command line.

SETIME does not require field delimiters, but allows the following delimiters between year, month, day, etc.:

colon (:), semicolon (;), slash (/), comma (,) or space ( ).

If semicolons are used as field delimiters the date and time string must be enclosed by quotes. For example:

```
$ setime "85;6;1;1;25;30;pm"
```

If no parameters are given, SETIME will issue the prompt:

```
$ setime
yy/mm/dd hh:mm:ss [am/pm]
Time:
```

When no am/pm field is specified, OS-9 system time uses the 24 hour clock, i.e., 1520 is 3:20 pm. Midnight is specified as 0000. Noon is specified as 1200. Using the am/pm field allows you to use the 12 hour clock. If there is a conflict between the time and the am/pm field (i.e., 1520 pm) the system will ignore the am/pm designation.

SETIME will echo the date/time when set.

IMPORTANT NOTE: This command must be executed before OS-9 can perform multitasking operations. If the system does not have a real time clock this command should still be used to set the date for the file system.

SYSTEMS WITH BATTERY BACKED UP CLOCKS: Setime should still be run to start time-slicing, but only the "-s" need be given. The date and time will be read from the clock.

(continued)

OPTIONS:

-?	Displays the usage of SETIME.
-d	Will not echo date/time when set.
-s	Must be used for battery backed up clocks.

EXAMPLES:

$ setime 82 12 22 15 45	Set to: Dec. 22, 1981, 3:45 PM
$ setime 821222 154500	Same as above
$ setime 82/12/22/3/45/pm	Same as above
$ setime -s	For system with battery-backup clock
$ setime yy/mm/dd hh:mm:ss [am/pm] Time:	no parameters given, therefore a prompt is given

SETPR SYSTEM MANAGEMENT UTILITY COMMAND SETPR

Set Process Priority

SYNTAX: SETPR <procID> <number>

FUNCTION: This command changes the CPU priority of a process.

EXECUTION: Type "setpr", followed by the process ID and the new priority number of the process to be changed. It may only be used with a process having the user's ID. The priority number is a decimal number in the range of 1 (lowest) to 65536 (hex FFFF).

The PROCS command can be used to obtain the ID number and present priority of any current process.

NOTE: This command does not appear in the CMDS directory as it is a built-in SHELL command.

EXAMPLE: $ setpr 8 250 Changes the priority of process #8 to 250

SHELL BASIC UTILITY COMMAND SHELL

OS-9 Command Interpreter

SYNTAX: SHELL [[SET] <arglist>]

FUNCTION: SHELL is OS-9's command interpreter program. It reads data from its standard input (the keyboard or a file) and interprets the data as a sequence of commands. The basic function of SHELL is to initiate and control execution of other OS-9 programs.

EXECUTION: Usually SHELL is entered automatically upon logging into OS-9. It will display a "$" prompt to show that it is ready and waiting for a command line. A new SHELL may also be created by typing "shell" optionally followed by a command line.

SHELL reads and interprets one text line at a time from standard input. After interpretation of each line, it reads another until an end-of-file condition occurs, at which time it terminates itself.

There is one exception: when SHELL is called from another program. In this case, SHELL takes the remainder of the command line as its first line of input. If this command line consists only of SHELL parameters or "chd" or "chx" commands, more lines will be read and processed; otherwise control will return to the calling program after the single command line is processed.

"Special characters" are used by SHELL for various purposes. Special characters consist of the following:

modifiers:	"#"	memory allocation
	"^"	process priority modification
	">"	standard output redirection
	"<"	standard input redirection
	">>"	standard error output redirection
separators:	";"	sequential execution
	"&"	concurrent execution
	"!"	pipe: interprocess communication
wild cards:	"*"	stands for any string of characters
	"?"	stands for any single character

If you wish to send one of these special characters to a utility program, you must use a method called "quoting" to prevent SHELL from interpreting the special character. Quoting consists of enclosing the sequence of characters to be passed to a routine in single or double quotes ('' or "").

For example the following command line will print the indicated string:

```
$ echo "Hello; goodbye"
Hello; goodbye
```

However, the following command would display the string "Hello" on your terminal and then attempt to execute a program called "goodbye".

```
$ echo Hello; goodbye
```

SHELL expands the two wild cards to build pathlists. The "?" wild card is used to match any single character. The "*" wild card will match any string of characters.

The command "dir ????" would display the names of files in the current directory that are four characters long. The command "dir s*" would display all names of files in the current directory that begin with "s".

Any command that uses a pathlist on the command line, will accept a pathlist specified with wild cards.

When SHELL expands the wild cards, if no explicit directory is given, the files in the current data directory will be searched for the matched expansion. If an explicit directory name is given in the pathlist, the specified directory will be searched.

NOTE: If a command uses an option to search for a file in the current execution directory, wild cards should not be used.

SETTING SHELL OPTIONS

There are two methods of setting SHELL options. The first method is to type the option on the command line, or after the command, "shell". For Example:

```
$ -np              turns off the SHELL prompt.

$ shell -np        creates a new shell that will not prompt.
```

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

```
$ set np          turns off the SHELL prompt.

$ shell set np    creates a new shell that will not prompt.
```

As you can see, the two methods accomplish the exact same function. They are both provided for your convenience. Use the method that is clearer for you.

OPTIONS:

-e=<file>	Prints error messages from specified <file>. If no file is not specified, the default file used is /dd/sys/errors. Without the "-e" option, SHELL will print only error numbers with no message description.
-ne	Prints no error messages (default).
-t	Echoes input lines.
-nt	Does not echo input lines (default).
-p	Displays prompt (default prompt: "$ ").
-np	Does not display prompt.
-p=<string>	Sets current SHELL prompt equal to <string>.
-x	Aborts process upon error (default).
-nx	Does not abort on error.

(continued)

ENVIRONMENTAL PARAMETERS

The SHELL supports a special type of parameter called an "environment" parameter. The environment parameter will be known to all processes called by the SHELL and all descendent SHELLS. This allows you to declare parameters that are "global" for the SHELL and all descendent processes of that SHELL. If the environment parameters are redefined in a child process, they will be known only to that process and descendents of that process.

OS-9 provides three commands to facilitate environment parameters:

1. SETENV declares the parameter and sets its value. The parameter is put in an environment storage area that is accessed by the SHELL. For example:

```
$ setenv name andy
$ setenv path1 /d1/doc/utilities
```

2. PRINTENV prints the environment parameter and its value to standard input. For example:

```
$ printenv
name andy
path1 /d1/doc/utilities
```

3. UNSETENV clears the value of the environment parameter and removes it from the list in environment storage. For example:

```
$ unsetenv name
$ unsetenv path1
```

These three commands are fully described in the utility command descriptions.

(continued)

SHELL COMMAND LINE SYNTAX

The SHELL command line consists of a "keyword" and optionally any of the parts listed below. The keyword must appear first on a command line. The optional parts' order will depend on the nature of the command and the desired effect. The command line consists of:

COMMAND LINE UNIT	DESCRIPTION
Keyword	A program, procedure file or built-in SHELL command. The SHELL built-in commands are: **ex** executes a process as overlay. **chd** changes your data directory. **chx** changes your execution directory. **kill** aborts a specified process. **set** sets SHELL options. **setenv** sets environment parameters. **setpr** sets process priority. **unsetenv** clears value of environment parameter. **w** waits for process to finish. All of the above built-in commands have a detailed description of their use in this chapter.
Object	A file or directory name. Wildcards may be used to identify object names. They are: * matches any character. ? matches any single character.

SHELL COMMAND LINE SYNTAX (continued)

COMMAND LINE UNIT	DESCRIPTION
Parameters	These are values, variables, constants, options, etc. to be passed to the program.
Execution Modifiers	These modify a program's execution by redirecting I/O or changing the priority or memory allocation of a process. They are: < redirects standard input. >[- or +] redirects standard output. >>[- or +] redirects standard error output. The "-" following the modifiers above signify to write over a specified file. The "+" means to append the file with the redirected output. #<mem size> allocates the specified memory to a process. ^<priority> sets the priority of a process to the specified priority.

(continued)

SHELL COMMAND LINE SYNTAX (continued)

COMMAND LINE UNIT	DESCRIPTION
Separators	Separators connect command lines together in the same command line. They specify to the SHELL how they are to be executed. They are: ; indicates sequential execution. & indicates concurrent execution. ! creates a communication "pipe" between processes. Pipes connect the standard output of one process to the standard input of another.

EXAMPLE COMMAND LINES:

`dir -u ! pr -n` Displays a numbered listing of the data directory. "Dir" is a keyword indicating the DIR utility. "-u" is a parameter for the DIR utility. "!" is a pipe that redirects the unformatted output of DIR to the standard input of "pr". "pr" is a keyword. "-n" is a parameter for the "PR" utility.

`list s*` Lists all files in the current data directory that begin with "s". "List" is the keyword. "s*" is the object.

`update <master ! sort >/p1`

"Update" uses "master" as standard input. "Update"'s output is used as input for "sort". The output from "sort" is redirected to the printer.

SLEEP SYSTEM MANAGEMENT UTILITY COMMAND SLEEP

Suspend process for period of time

SYNTAX: SLEEP [<opts>] <count> [<opts>]

FUNCTION: SLEEP puts the user's process to "sleep" for a number of "ticks". It is generally used to generate time delays or to "break up" CPU-intensive jobs.

EXECUTION: Type "sleep", followed by the number of ticks you want the process to sleep. A tick count of 1 causes the process to "give up" its current time slice. A tick count of zero causes the process to sleep indefinitely (usually awakened by a signal). The duration of a tick is system-dependent.

OPTIONS: -? Displays the usage of SLEEP.

-s Changes count representation to seconds.

NOTE: Only one option may be used on the command line. If not specified, <count> will default to 0.

EXAMPLES:

```
$ sleep 25          Sleep for 25 ticks

$ sleep -s 1000     Sleep for 1000 seconds
```

TEE PROGRAMMER UTILITY COMMAND TEE

Copy standard input to multiple output paths

SYNTAX: TEE [<path>]

FUNCTION: TEE is a filter that copies all text lines from its standard input to its standard output and any other additional pathlists given as parameters.

EXECUTION: Type "tee" followed by the pathlist(s) to which standard input is to be redirected. This utility is generally used in coordination with input redirected through a pipe.

OPTIONS: -? Displays the usage of TEE.

EXAMPLES:

The example below uses a pipeline and TEE to simultaneously send the output listing of the DIR command to the terminal, printer, and a disk file:

```
$ dir -e ! tee /printer /d0/dir.listing
```

The following example sends the output of an assembler listing to a disk file and the printer:

```
$ asm pgm.src l ! tee pgm.list >/printer
```

The example below "broadcasts" a message to three terminals:

```
$ echo WARNING System down in 10 minutes ! tee /t1 /t2 /t3
```

TMODE BASIC UTILITY COMMAND TMODE

Change terminal operating mode

SYNTAX: TMODE [<opts>] [<arglist>] [<opts>]

FUNCTION: TMODE is used to display or change the operating parameters of the user's terminal.

EXECUTION: Type "tmode", followed by any parameters that are desired to be changed. If no arguments are given, the present values for each parameter are displayed. Otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be given, and are separated by spaces or commas.

The "-w=<path#>" option can be used to specify the path number to be affected. If none is given, standard input is affected.

NOTE: If the TMODE command is used in a SHELL procedure file, the option "-w=<path#>" must be used to specify one of the standard paths (0, 1 or 2) to change the terminal's operating characteristics. The change will remain in effect until the path is closed. To effect a permanent change to a device characteristic, the device descriptor must be changed. You may alter the device descriptor to set a device's initial operating parameters using the XMODE utility. See the description for XMODE for more information.

The TMODE command can work only if a path to the file/device has already been opened. The "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" contains more information on device descriptors.

OPTIONS: -? Displays the usage of TMODE.

-w=<path#> Changes the path number <path#> affected.

TMODE PARAMETER NAMES

NAME	FUNCTION
upc	Upper case only. Lower case characters are converted automatically to upper case.
noupc	Upper and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).

(continued)

NAME	FUNCTION
nobsb	No erase on backspace: echoes single backspace only.
bsl	Backspace over line (default): lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals).
nobsl	No backspace over line: lines are "deleted" by printing a "new line" sequence (for hard-copy terminals).
echo	Input characters "echoed" back to terminal (default).
noecho	No echo.
lf	Auto line feed on (default): line feeds automatically echoed to terminal on input and output carriage returns
nolf	Auto line feed off.
null=n	Set null count (default = 0): number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal.
pause	Screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
nopause	Screen pause mode off.
pag=n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp=h	Set input backspace character (normally control H, default = 08). Numeric value of the character in hexadecimal.
del=h	Set input delete line character (normally control X, default = 18). Numeric value of character in hexadecimal.

(continued)

NAME	FUNCTION
eor=h	Set end-of-record input character (normally carriage return, default = 0D). Numeric value of character in hexadecimal.
eof=h	Set end-of-file input character (normally <esc>, default = 1B). Numeric value of character in hexadecimal.
reprint=h	Set reprint line character (normally control D, default = 04). Numeric value of character in hexadecimal.
dup=h	Set duplicate last input line character (normally control A, default = 01). Numeric value of character in hexadecimal.
psc=h	Set pause character (normally control W, default = 17). Numeric value of character in hexadecimal.
abort=h	Abort character (normally control C, default = 03). Numeric value of character in hexadecimal.
quit=h	Quit character (normally control E, default = 05). Numeric value of character in hexadecimal.
bse=h	Set output backspace character (default = 08). Numeric value of character in hexadecimal.
bell=h	Set bell (alert) output character (default = 07) Numeric value of character in hexadecimal.
type=h	ACIA initialization value: sets parity, character size and number of stop bits. Value in hexadecimal. This value will also be affected by changing the individual par(ity), cs (character length) and stop (stop bits) values. This value will not be affected by "tmode normal" command.

(continued)

NAME	FUNCTION
par=s	Sets parity using one of the following strings: odd, even or none. Setting this value will affect the type value. This value will not be affected by the "tmode normal" command.
cs=n	Sets character length to one of the following values: n = 8, 7, 6 or 5 (bits) Setting this value will affect the type value. This value will not be affected by the "tmode normal" command.
stop=n	Sets number of stop bits to one of the following: n = 1, 1.5 or 2 (stop bits) Setting this value will affect the type value. This value will not be affected by the "tmode normal" command.
baud=n	Baud rate: The baud rate may currently be set to the following values: n = 50 300 2400 19200 75 600 3600 38400 110 1200 4800 extern 134.5 1800 7200 150 2000 9600 This value will not be affected by the "tmode normal" command.
xon=h	DC1 resume output character (normally control Q, default = 11). Numeric value of character in hexadecimal.

(continued)

NAME	FUNCTION
xoff=h	DC2 suspend output character (normally control S, default = 13). Numeric value of character in hexadecimal.
tabc=h	Tab character (normally control I, default = 09). Numeric value of character in hexadecimal.
tabs=n	Number of characters between tab stops (Default = 4). The number is in decimal.
normal	Set the terminal back to its default characteristics. This will not affect the following values: type, baud rate, parity, character length and stop bits.

EXAMPLES:

```
$ tmode noupc lf null=4 bse=1F pause

$ tmode pag=24 pause bsl noecho bsp=8 bsl=C

$ tmode xon xoff quit=5
```

TOUCH PROGRAMMER UTILITY COMMAND TOUCH

Update file date

SYNTAX: TOUCH [<opts>] {<path> [<opts>]}

FUNCTION: TOUCH updates the last modification date of a file. This command is usually used with a MAKE command's "makefile". Associated with every file is the date that the file was last modified. Touch simply opens a file and closes it to update the time that the file was last modified to the current date.

EXECUTION: Type "touch", followed by the pathlist of the file to be updated. TOUCH searches the current data directory for the file to be updated (if another directory or the "-x" option is not specified).

NOTE: If the specified file is not found, TOUCH will create a file with a current modification date.

OPTIONS:

-?	Displays the usage of TOUCH.
-c	Will not create a file if not found.
-q	Will not quit if an error occurs.
-x	Searches the execution directory for the file.
-z	Reads the file names from standard input.
-z=<path>	Reads the file names from <path>.

EXAMPLES:

```
$ touch -c /h0/doc/program

$ touch -cz

$ dir -u ! touch
```

TR PROGRAMMER UTILITY COMMAND TR

Transliterate characters

SINTAX: TR [<opts>] <str1> [<str2>] [<path1>] [<path2>] [<opts>]

FUNCTION: TR transliterates characters from <str1> into a corresponding character from <str2>.

EXECUTION: Type "tr", followed by the characters to be searched for (<str1>), and optionally the replacement characters (<str2>), the input file's pathlist (<path2>) and the output file's pathlist (<path2>).

<str1> is required. If <str2> is missing, all characters in <str1> will be deleted from the output. If <path1> and <path2> are missing, standard input and output are assumed. If only one path is specified, it will be used as the input file pathlist.

EXPRESSION MODE:

TR can also recognize certain sets of characters when the "-e" option is used. The set of characters is specified by the use of a "regular expression" (RE). A string which is a member of this set is said to match the RE.

In expression mode, a RE is used for <str1>. The entire expression must match before <str2> is substituted in the output stream. If the length of <str2> exceeds the length of <str1>, <str2> will be truncated in the output.

To facilitate the creation of REs, some metacharacters are defined to enable the creation of complex sets of characters. The chart on the following page defines these metacharacters and explains their use.

(continued)

CHAR	NAME: DEFINITION
.	ANY: "." is defined to match any ASCII character except new line.
[]	CHARACTER CLASS: "[]" defines a group of characters which will match any single character in the compare string. TR recognizes certain abbreviations to aid the entry of ranges of strings: [a-z] is equivalent to the character string "abcdefghijklmnopqrstuvwxyz". [m-pa-f] is equivalent to the character string "mnopabcdef". [0-7] is equivalent to the character string "01234567".
~	BOL or NEGATE: "~" is defined to modify a character class, (as described above) when between []. At the beginning of an entire RE, it requires the RE to compare and match the string only at the beginning of the line. The NEGATE character modifies the character class so that it will match any ASCII character NOT in the given class, or newline.
$	EOL: "$" is defined to require the RE to compare and match the string only at the end of the line.
\	ESCAPE: "\" is defined to remove special significance from special characters. It is followed by a base and a numeric value or a special character. If no base is specified, the base for the numeric value will default to hexadecimal. An explicit base of decimal or hexadecimal can be specified by preceding the numeric value with a qualifier of "d" or "x", respectively. It also allows entry of some non-printable characters: \t = tab char \n = new-line char \l = line feed char \b = backspace char \f = form feed char

EXAMPLE EXPRESSIONS:

Any combination of metacharacters and normal characters can be combined to create a RE.

Regular Expression	Same as
abcd	abcd
ab.d	abcd, abxd, ab?d, etc.
~abcd	abcd (only if very first chars on a line)
abcd$	abcd (only if very last chars on a line)
~abcd$	abcd (only if abcd is the complete line)
[a-z]bcd	abcd, bbcd, cbcd, ..., zbcd
[Aa]bcd	abcd, Abcd
abcd[0-9a-zA-z]	abcd followed by any alpha-numeric char
abcd[a-d]	abcd followed by a, b, c or d
bcd[~a-d]	bcd followed by any ASCII char except a, b, c, d or new-line

OPTIONS:

-? Displays the usage of TR.

-c Transliterates all ASCII characters (1 through $7F) to <str2>, except for the set of characters in <str1>.

-d Deletes all matching input characters and expressions.

-e Allows regular expressions (expression mode).

OPTIONS (continued):

-s	Squeezes all repeated output characters or expressions in <str2> to single characters or expressions.
-v	Same as "-c" option.
-z	Reads standard input for list of file names.

Options can generally be given anywhere on the command line. If you wish to use the pathlists but not <str2>, the "-d" option must be specified prior to the pathlists. Similarly, if you use the "-z" option to read pathlists from standard input, the "-z" must precede <path2>.

The "c" (or "v") option is only recognized in the character mode. It makes little sense in expression mode. The "-s" and "-d" options are mutually exclusive.

WARNING: Beware of double negatives. Using [~xyz] along with the "-c" option will give unexpected results.

Note that if you use the "-c" option to change all but a certain sequence of characters, it will also change carriage returns and newlines unless they are specified in the sequence of characters.

EXAMPLES:

All of the following examples will use standard input for the input to TR. The output will be sent to standard output. Thus, the first line following each command line will be the standard input, and the second line will be the standard output.

```
$ tr abcd jklm          Transliterates standard input,
aabdc_efg               converting "a" to "j", "b" to
jjklm_efg               "k", "c" to "l" and "d" to "m"

$ tr abcd j             Transliterates standard input,
abcd_efgh               converting each "a", "b", "c",
jjjj_efgh               and "d" to "j".
```

(continued)

EXAMPLES (continued):

```
$ tr abcd                  Transliterates standard input,
abcd_efgh                  deleting each "a", "b", "c",
_efgh                      and "d".

$ tr abcd jk               Transliterates standard input,
abcd_efgh                  converting each "a" to "j" and
jkkk_efgh                  each "b", "c" and "d" to "k"

$ tr -e abcd jkl           Transliterates standard input,
abc_abcd_efgh              converting each string of
abc_jkl_efgh               "abcd" to "jkl".

$ tr -e [a-d] k            Transliterates standard input,
abc_abcd-efgh              converting each character
kkk_kkkk-efgh              contained in the expression
                           "abcd" to "k".

$ tr -c [a-z A-Z] \n       Transliterates standard input,
one word per line          converting all non-alphabetic
one                        characters to newline characters.
word
per
line
```

TSMON **SYSTEM MANAGEMENT UTILITY COMMAND** **TSMON**

Timesharing monitor

SYNTAX: TSMON [<device name>]

FUNCTION: TSMON is used to supervise idle terminals and initiate the login sequence in timesharing applications. If a pathlist is given, standard I/O paths are opened for the device. When a carriage return is typed, TSMON will automatically call the LOGIN command. If LOGIN fails because the user could not supply a valid user name or password, it will return to TSMON.

Most programs will terminate when an end of file character (normally <escape>) is entered as the first character on a command line. This will log you off of the system and return control to TSMON.

OPTIONS: -? Display the usage of TSMON

NOTE: The LOGIN command and its password file must be present for TSMON to work correctly (see the LOGIN command description).

EXAMPLES:

```
$ tsmon /t1&
&005
```

UNLINK SYSTEM MANAGEMENT UTILITY COMMAND UNLINK

Unlink memory module

SYNTAX: UNLINK [<opts>] [<modname> [<opts>]]

FUNCTION: UNLINK tells OS-9 that the memory module(s) named are no longer needed by the user.

EXECUTION: Type "unlink", followed by the name(s) of the module(s) to be unlinked. The link count will then be decremented by one. If the link count becomes zero, the module(s) are destroyed and their memory reassigned.

It is good practice to UNLINK modules whenever possible to make most efficient use of available memory resources.

WARNING: Never UNLINK a module you did not load or link to. Unlinking a module more than once may prematurely lower its link count and possibly destroy the module while it is still in use.

OPTIONS:

-? Displays the usage of UNLINK.

-z Reads the module names from standard input.

-z=<file> Reads the module names from <file>.

EXAMPLES:

```
$ mdir
   Module Directory at 14:44:35
kernel      init        p32clk      rbf         p32hd
h0          p32fd       d0          d1          ram
r0          dd          edit

$ unlink edit
$ mdir
   Module Directory at 14:44:35
kernel      init        p32clk      rbf         p32hd
h0          p32fd       d0          d1          ram
r0          dd
```

(continued)

EXAMPLES (continued):

$ unlink pgm pm5 pgm9	Unlinks "pgm", "pgm5" and "pgm9" and lowers the link count of each module by one.
$ dir -u ! unlink -z	Pipes an unsorted listing of the current data directory to UNLINK. This unlinks all modules contained in the directory and lowers the link count of each module by one.
$ unlink -z=namefile	Unlinks each module listed in "namefile" and lowers the link count of each module by one.

UNSETENV PROGRAMMER UTILITY COMMAND UNSETENV

Clear environment parameter

SYNTAX: UNSETENV <eparam>

FUNCTION: UNSETENV clears an environment parameter in the evironment list.

EXECUTION: Type "unsetenv", followed by the environment parameter to be "unset". This clears the value of the parameter and removes it from the environment list.

NOTE: If the specified parameter has not been previously declared, UNSETENV has no effect, and it will give you no message.

OPTIONS: None.

EXAMPLES:

```
$ unsetenv name
$ unsetenv term
```

SEE ALSO: SETENV and PRINTENV utility command descriptions.

XMODE SYSTEM MANAGEMENT UTILITY COMMAND XMODE

Examine or Change Device Initialization Mode

SYNTAX: XMODE [<opts>] <devname> [<arglist>] [<devname> [<opts>]]

FUNCTION: This command is used to display or change the initialization parameters of any SCF-type device such as a video display, printer, RS-232 port, etc. Some common uses are to change the baud rates and control key definitions.

EXECUTION: Type "xmode" and any parameters that are desired to be changed. If no arguments are given, the present values for each parameter are displayed. Otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be given, and are separated by spaces or commas. You must specify a device name, if the parameter(s) given in the argument list are to be processed.

XMODE is very similar to the TMODE command. TMODE only operates on open paths so its effect is temporary. XMODE actually updates the device descriptor. The change persists as long as the computer is running, even if paths to the device are repetitively opened and closed. If XMODE is used to change parameter(s) and then OS9GEN is used to make a new system disk, the changed parameter will be permanently reflected on the new system disk.

XMODE PARAMETER NAMES

NAME	FUNCTION
upc	Upper case only. Lower case characters are converted automatically to upper case.
noupc	Upper and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
nobsb	No erase on backspace: echoes single backspace only.
bsl	Backspace over line (default): lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals).

(continued)

NAME	FUNCTION
nobsl	No backspace over line: lines are "deleted" by printing a "new line" sequence (for hard-copy terminals).
echo	Input characters "echoed" back to terminal (default).
noecho	No echo.
lf	Auto line feed on (default): line feeds automatically echoed to terminal on input and output carriage returns
nolf	Auto line feed off.
null=n	Set null count (default = 0): number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal.
pause	Screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
nopause	Screen pause mode off.
pag=n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp=h	Set input backspace character (normally control H, default = 08). Numeric value of the character in hexadecimal.
del=h	Set input delete line character (normally control X, default = 18). Numeric value of character in hexadecimal.
eor=h	Set end-of-record input character (normally carriage return, default = 0D). Numeric value of character in hexadecimal.
eof=h	Set end-of-file input character (normally <esc>, default = 1B). Numeric value of character in hexadecimal.

(continued)

NAME	FUNCTION
reprint=h	Set reprint line character (normally control D, default = 04). Numeric value of character in hexadecimal.
dup=h	Set duplicate last input line character (normally control A, default = 01). Numeric value of character in hexadecimal.
psc=h	Set pause character (normally control W, default = 17). Numeric value of character in hexadecimal.
abort=h	Abort character (normally control C, default = 03). Numeric value of character in hexadecimal.
quit=h	Quit character (normally control E, default = 05). Numeric value of character in hexadecimal.
bse=h	Set output backspace character (default = 08). Numeric value of character in hexadecimal.
bell=h	Set bell (alert) output character (default = 07) Numeric value of character in hexadecimal.
type=h	ACIA initialization value: sets parity, character size and number of stop bits. Value in hexadecimal. This value will also be affected by changing the individual par(ity), cs (character length) and stop (stop bits) values. This value will not be affected by "xmode normal" command.
par=s	Sets parity using one of the following strings: odd, even or none. Setting this value will affect the type value. This value will not be affected by the "xmode normal" command.

(continued)

NAME	FUNCTION
cs=n	Sets character length to one of the following values: n = 8, 7, 6 or 5 (bits) Setting this value will affect the type value. This value will not be affected by the "xmode normal" command.
stop=n	Sets number of stop bits to one of the following: n = 1, 1.5 or 2 (stop bits) Setting this value will affect the type value. This value will not be affected by the "xmode normal" command.
baud=n	Baud rate: The baud rate may currently be set to the following values: n = 50 300 2400 19200 75 600 3600 38400 110 1200 4800 extern 134.5 1800 7200 150 2000 9600 This value will not be affected by the "xmode normal" command.
xon=h	DC1 resume output character (normally control Q, default = 11). Numeric value of character in hexadecimal.
xoff=h	DC2 suspend output character (normally control S, default = 13). Numeric value of character in hexadecimal.
tabc=h	Tab character (normally control I, default = 09). Numeric value of character in hexadecimal.

(continued)

NAME	FUNCTION
tabs=n	Number of characters between tab stops (Default = 4). The number is in decimal.
normal	Set the terminal back to its default characteristics. This will not affect the following values: type, baud rate, parity, character length and stop bits.

OPTIONS: -? Display the usage of XMODE

-z Reads device names from standard input

-z=<file> Reads device names from <file>

EXAMPLES: $ xmode /term noupc lf null=4 bse=1F pause

$ xmode /t1 pag=24 pause bsl noecho bsp=8 bsl=C

end of chapter 6

USER NOTES

OS-9 ERROR CODES

ERROR NUMBER	DESCRIPTION
000:200 E$PthFul	PATH TABLE FULL - This error is generally returned when a user program has tried to open more than 32 I/O paths simultaneously. When the system path table becomes full, the kernel automatically expands it. However, this error could be returned if there is not enough (contiguous) memory to expand it.
000:201 E$BPNum	ILLEGAL PATH NUMBER - This error is returned when the path number was too large or for a nonexistant path. This could occur whenever passing a path number to an I/O call.
000:202 E$Poll	INTERRUPT POLLING TABLE FULL - This error is returned when an attempt is made to install an IRQ Service Routine into the system polling table, and the table is full. To install another interrupt producing device, one must first be removed. The system's INIT module specifies the maximum number of IRQ devices that may be installed.
000:203 E$BMode	ILLEGAL MODE - This error is returned when an attempt is made to perform an I/O function of which the device or file was incapable. This could occur, for instance, when trying to read from an output file (for example, a printer.

ERROR NUMBER		DESCRIPTION
000:204	E$DevOvf	DEVICE TABLE FULL - This error is returned when the specified device can not be added to the system because the device table is full. To install another device, one must first be removed. The system's INIT module specifies the maximum number of devices that may be supported, but this may be changed to add more
000:205	E$BMID	ILLEGAL MODULE HEADER - This error is returned when the specified module can not be loaded because its module sync code is incorrect.
000:206	E$DirFul	MODULE DIRECTORY FULL - This error is returned when the specified module can not be added to the system, because the module directory is full. To load or create another module, one must first be unlinked. While OS-9 expands the module directory when it becomes full, this error may be returned because there is not enough memory or the memory is too fragmented to use.
000:207	E$MemFul	MEMORY FULL - This error is returned when the process will not execute because there is not enough contiguous RAM free. This can also occur if a process has already been alloacated the maximum number of blocks permitted by the system. This could occur when trying to load a module.

ERROR NUMBER		DESCRIPTION
000:208	E$UnkSvc	ILLEGAL SERVICE REQUEST - This error is returned when the specified service call has an unknown or invalid service code number. This can also occur if a getstat/setstat call is made with an unknown status code.
000:209	E$ModBsy	MODULE BUSY - This error is returned when trying to access a non-sharable module that is in use by another process.
000:210	E$BPAddr	BOUNDARY ERROR - This error is returned when a memory allocation or deallocation request is not on a page boundary or an attempt is made to deallocate memory not previously assigned.
000:211	E$EOF	END OF FILE - This error is returned when an end of file condition is encountered on a read operation.
000:212	E$VctBsy	VECTOR BUSY - This error is returned when a device is trying to use an IRQ vector that is currently being used by another device.
000:213	E$NES	NON-EXISTING SEGMENT - This error is returned when a search is made for a disk file segment that could not be found. The device may have a damaged file structure.

ERROR NUMBER		DESCRIPTION
000:214	E$FNA	FILE NOT ACCESSABLE - This error is returned when trying to open a file or device without the correct access permissions. Check the file's attributes and the owner ID.
000:215	E$BPNam	BAD PATH NAME - This error is returned when there is a syntax error in the specified pathlist (illegal character, etc.). This can occur whenever referencing a path by name.
000:216	E$PNNF	PATH NAME NOT FOUND - This error is returned when the specified pathlist can not be found. This could be caused by misspellings or incorrect directories, etc.
000:217	E$SLF	SEGMENT LIST FULL - This error is returned when a file is too fragmented to be expanded any further. This can be caused by expanding a file many times without regard to allocation of memory. This can also occur on a disk that has little free memory left or one whose free memory is too scattered. The simplest way to solve this problem is to copy the file (or disk), which should move it into more contiguous areas.
000:218	E$CEF	FILE ALREADY EXISTS - This error occurs when trying to create or duplicate a file using a name that already appears in the current directory.

ERROR NUMBER		DESCRIPTION
000:219	E$IBA	ILLEGAL BLOCK ADDRESS - This error is returned when a search for an illegal block address has occurred. An invalid pointer or block size has been passed or the device's file structure is damaged.
000:220	E$Hangup	TELEPHONE (MODEM) DATA CARRIER LOST
000:221	E$MNF	MODULE NOT FOUND - This error is returned when a request is made to link to a module that is not found in the module directory.
000:222	E$NoClk	NO CLOCK - This error is returned when a request is made that uses the system clock and the system has no clock running. For example, a SLEEP request will return this error if there is no system clock running. SETIME is used to start the system clock.
000:223	E$DelSP	SUICIDE ATTEMPT - This error is returned when a User requests to deallocate and return the memory where the user's stack is located. This could be caused, for example, by using the F$Mem system call to contract the data memory of the specified process.
000:224	E$IPrcID	ILLEGAL PROCESS NUMBER - This error is returned when a system call is passed a process ID to a non-existant process or a process that the user may not access.

ERROR NUMBER		DESCRIPTION
000:225	E$Param	BAD POLLING PARAMETER - This error is returned when an impossible vector number is passed to the IRQ polling system.
000:226	E$NoChld	NO CHILDREN - This error is returned when a F$Wait request is made and the process can not wait for its child process' I/O, because it has no child process.
000:227	E$ITrap	ILLEGAL TRAP CODE - This error is returned when an unavailable (already in use) or invalid trap code is used in a TLINK call.
000:228	E$PrcAbt	PROCESS ABORTED - This error is returned when a process is aborted by the signal code: 000:0.
000:229	E$PrcFul	PROCESS TABLE FULL - This error is returned when the system process table is full (too many processes currently running). While OS-9 automatically tries to expand the table, this error may occur if there is not enough contiguous memory to do so.
000:230	E$IForkP	ILLEGAL PARAMETER AREA - This error occurs when the ridiculous parameters are passed to fork call.

ERROR NUMBER		DESCRIPTION
000:231	E$KwnMod	KNOWN MODULE - This error is returned when a call is made to install a module that is already in memory.
000:232	E$BMRC	INCORRECT MODULE CRC - This error is returned when the specified module being checked or verified has a bad CRC value. To generate a valid CRC, use the FIXMOD utility.
000:233	E$USigP	SIGNAL ERROR - This error is returned by F$Send when the receiving process has a previous unprocessed signal pending. The sending process should try again later.
000:234	E$NEMod	NON-EXISTENT MODULE - This error is returned when a process tries to search for an unlocatable module. This might occur when using F$Chain or F$Link.
000:235	E$BNam	BAD NAME - This error occurs when there is a syntax error in the specified name.
000:236	E$BMHP	BAD PARITY - This error is returned when the specified module has bad module header parity.

ERROR NUMBER		DESCRIPTION
000:237	E$NoRAM	RAM FULL - This error occurs when there is no free system RAM available at the time of the request for memory allocation. This also occurs when there is not enough contiguous memory to process a fork request.
000:238	E$DNE	DIRECTORY NOT EMPTY - This error is returned when attempting to remove the directory attribute from a directory that is not empty.
000:239	E$NoTask	NO TASK NUMBER AVAILABLE - This error occurs when all task numbers are currently in use and a request is made for execution or creation of a new task. This error will not occur on OS-9 Level One systems.

DEVICE DRIVER ERRORS

The following error codes are generated by I/O device drivers, and are somewhat hardware dependent.

ERROR NUMBER		DESCRIPTION
000:240	E$Unit	ILLEGAL DRIVE NUMBER
000:241	E$Sect	BAD SECTOR - bad disk sector number.
000:242	E$WP	WRITE PROTECT - device is write protected.
000:243	E$CRC	CRC ERROR - CRC error on read or write verify.
000:244	E$Read	READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
000:245	E$Write	WRITE ERROR - hardware error during disk write operation.
000:246	E$NotRdy	NOT READY - device has "not ready" status.

ERROR NUMBER		DESCRIPTION
000:247	E$Seek	SEEK ERROR - physical seek to non-existent sector.
000:248	E$Full	MEDIA FULL - insufficient free space on media.
000:249	E$BType	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
000:250	E$DevBsy	DEVICE BUSY - non-sharable device is in use.
000:251	E$DIDC	DISK ID CHANGE - This error is returned when the disk media was changed with open files. RBF copies the disk ID number (from sector 0) into the path descriptor of each path when it is opened. If this does not agree with the driver's current disk ID, this error is returned. The driver updates the current disk ID only when sector 0 is read. It is thus possible to swap disks without RBF noticing. This check helps to prevent this possibility.
000:252	E$Look	RECORD IS LOCKED-OUT - This error is returned if another process is accessing the requested record. Normal record locking routines will wait forever for a record in use by another user to become available. However, RBF may be told to wait for a finite amount of time with a setstat. If the time expires before the record becomes free, this error is returned.

ERROR NUMBER		DESCRIPTION
000:253	E$Share	NON-SHARABLE FILE BUSY - The requested file or device has the single user bit set or was opened in single user mode and another process is accessing the requested file. A common way to get this error is to attempt to delete a file that is currently open.
000:254	E$DesdLk	I/O DEADLOCK - Two processes are attempting to use the same two disk areas simultaneously. Each process is locking out the other process, producing the I/O deadlock. To proceed, one of the two processes must release its control to allow the other to proceed.
000:255	E$Format	DEVICE IS FORMAT PROTECTED - This error is returned when an attempt is made to format a disk that has been format protected. A bit in the device descriptor may be changed to allow the device to be formatted. Formatting is usually inhibited on hard disks to prevent erasure.

UNINITIALIZED TRAP ERRORS

ERROR NUMBER		DESCRIPTION
000:102	E$BusErr	BUS ERROR - bus error exception occurred.
000:103	E$AdrErr	ADDRESS ERROR - address error exception occurred.
000:104	E$IllIns	ILLEGAL INSTRUCTION - illegal instruction exception occurred.
000:105	E$ZerDiv	ZERO DIVIDE - zero divide exception occurred.
000:106	E$Chk	CHECK - CHK instruction exception occurred.
000:107	E$TrapV	TRAPV - TrapV instruction exception occurred.
000:108	E$Violat	PRIVILEGE VIOLATION - privilege violation exception occurred.
000:109	E$Trace	TRACE ERROR - uninitialized trace exception occurred.

ERROR NUMBER	DESCRIPTION
000:110 E$1010	1010 TRAP - Line 1010 emulator exception occurred.
000:111 E$1111	1111 TRAP - Line 1111 emulator exception occurred.
000:112 - 000:123 E$Resrvd	reserved: an invalid TRAP (#12 - 23) occurred.
000:124 - 000:138 E$Trap	uninitialized user TRAP 1-15 executed.

OTHER ERRORS

ERROR NUMBER		DESCRIPTION
000:002		KEYBOARD QUIT - This error is returned when the "keyboard abort" function (control E) is sent.
000:003		KEYBOARD INTERRUPT - This error is returned when the "keyboard interrupt" function (control C) is sent.
000:064	E$IllFno	ILLEGAL FUNCTION CODE Math trap handler error.
000:065	E$FmtErr	FORMAT ERROR Math trap handler error.
000:066	E$NotNum	NUMBER NOT FOUND Math trap handler error.
000:067	E$IllArg	ILLEGAL ARGUMENT Math trap handler error.
000:139	E$Permit	NO PERMISSION - you must be super user to perform the requested function.

ERROR NUMBER		DESCRIPTION
000:140	E$Differ	DIFFERENT ARGUMENTS - the arguments to F$ChkNam do not match.
000:141	E$StkOvf	STACK OVERFLOW - F$ChkNam can cause this error if the pattern string is too complex.
000:142	E$EvntID	ILLEGAL EVENT ID - This error is returned when an invalid or illegal event ID number is specified.
000:143	E$EvNF	EVENT NAME NOT FOUND - This error is returned when an attempt to link to or delete an event is made, but the name is not found in the event table.
000:145	E$EvBusy	EVENT BUSY - This error is returned when an attempt to delete an event is made and its link count is non-zero. This can also occur if an attempt to create an already existant named event is made.
000:146	E$EvParm	IMPOSSIBLE EVENT PARAMETER - This error is returned when impossible parameters are passed to F$Event.

end of appendix a

USER NOTES

APPENDIX B
ASCII CONVERSION CHART

ASCII is an acronym for American Standard Code for Information Interchange. It consists of 96 printable and 32 unprintable characters. The following conversion table includes Binary, Decimal, Octal, Hexadecimal, and ASCII. The unprintable characters are defined at the end of this appendix.

BINARY	DECIMAL	OCTAL	HEXADECIMAL	ASCII
0000000	0	0	0	NUL
0000001	1	1	1	SOH
0000010	2	2	2	STX
0000011	3	3	3	ETX
0000100	4	4	4	EOT
0000101	5	5	5	ENQ
0000110	6	6	6	ACK
0000111	7	7	7	BEL
0001000	8	10	8	BS
0001001	9	11	9	HT
0001010	10	12	A	LF
0001011	11	13	B	VT
0001100	12	14	C	FF
0001101	13	15	D	CR
0001110	14	16	E	SO
0001111	15	17	F	SI
0010000	16	20	10	DLE
0010001	17	21	11	DC1
0010010	18	22	12	DC2
0010011	19	23	13	DC3
0010100	20	24	14	DC4
0010101	21	25	15	NAK
0010110	22	26	16	SYN
0010111	23	27	17	ETB
0011000	24	30	18	CAN
0011001	25	31	19	EM
0011010	26	32	1A	SUB
0011011	27	33	1B	ESC
0011100	28	34	1C	FS
0011101	29	35	1D	GS
0011110	30	36	1E	RS
0011111	31	37	1F	US
0100000	32	40	20	SP
0100001	33	41	21	!
0100010	34	42	22	"
0100011	35	43	23	#
0100100	36	44	24	$
0100101	37	45	25	%
0100110	38	46	26	&
0100111	39	47	27	'
0101000	40	50	28	(

BINARY	DECIMAL	OCTAL	HEXADECIMAL	ASCII
0101001	41	51	29	)
0101010	42	52	2A	*
0101011	43	53	2B	+
0101100	44	54	2C	,
0101101	45	55	2D	-
0101110	46	56	2E	.
0101111	47	57	2F	/
0110000	48	60	30	0
0110001	49	61	31	1
0110010	50	62	32	2
0110011	51	63	33	3
0110100	52	64	34	4
0110101	53	65	35	5
0110110	54	66	36	6
0110111	55	67	37	7
0111000	56	70	38	8
0111001	57	71	39	9
0111010	58	72	3A	:
0111011	59	73	3B	;
0111100	60	74	3C	<
0111101	61	75	3D	=
0111110	62	76	3E	>
0111111	63	77	3F	?
1000000	64	100	40	@
1000001	65	101	41	A
1000010	66	102	42	B
1000011	67	103	43	C
1000100	68	104	44	D
1000101	69	105	45	E
1000110	70	106	46	F
1000111	71	107	47	G
1001000	72	110	48	H
1001001	73	111	49	I
1001010	74	112	4A	J
1001011	75	113	4B	K
1001100	76	114	4C	L
1001101	77	115	4D	M
1001110	78	116	4E	N
1001111	79	117	4F	O
1010000	80	120	50	P
1010001	81	121	51	Q
1010010	82	122	52	R
1010011	83	123	53	S
1010100	84	124	54	T
1010101	85	125	55	U
1010110	86	126	56	V
1010111	87	127	57	W

BINARY	DECIMAL	OCTAL	HEXADECIMAL	ASCII
1011000	88	130	58	X
1011001	89	131	59	Y
1011010	90	132	5A	Z
1011011	91	133	5B	[
1011100	92	134	5C	\
1011101	93	135	5D	]
1011110	94	136	5E	
1011111	95	137	5F	_
1100000	96	140	60	`
1100001	97	141	61	a
1100010	98	142	62	b
1100011	99	143	63	c
1100100	100	144	64	d
1100101	101	145	65	e
1100110	102	146	66	f
1100111	103	147	67	g
1101000	104	150	68	h
1101001	105	151	69	i
1101010	106	152	6A	j
1101011	107	153	6B	k
1101100	108	154	6C	l
1101101	109	155	6D	m
1101110	110	156	6E	n
1101111	111	157	6F	o
1110000	112	160	70	p
1110001	113	161	71	q
1110010	114	162	72	r
1110011	115	163	73	s
1110100	116	164	74	t
1110101	117	165	75	u
1110110	118	166	76	v
1110111	119	167	77	w
1111000	120	170	78	x
1111001	121	171	79	y
1111010	122	172	7A	z
1111011	123	173	7B	{
1111100	124	174	7C	|
1111101	125	175	7D	}
1111110	126	176	7E	~
1111111	127	177	7F	DEL

ASCII SYMBOL DEFINITIONS

SYMBOL	DEFINITION	SYMBOL	DEFINITION
ACK	acknowledge	FS	file separator
BEL	bell	GS	group separator
BS	backspace	HT	horizontal tabulation
CAN	cancel	LF	line feed
CR	carriage return	NAK	negative acknowledgement
DC	device control	NUL	null
DEL	delete	RS	record shipment
DLE	data link escape	SI	shift in
EM	end of medium	SO	shift out
ENQ	enquiry	SOH	start of heading
EOT	end of transmission	SP	space
ESC	escape	STX	start of text
ETB	end of transmission	SUB	substitute
ETX	end of text	SYN	synchronous idle
FF	form feed	US	unit separator
		VT	vertical tabulation

end of appendix b

APPENDIX C
CONTROL KEYS

KEY	FUNCTION
CONTROL A	Repeats the previous input line. The last line entered will be redisplayed but not executed. The cursor will be positioned at the end of the line. You may enter the line as is or you can add more characters to it. You can edit the line by backspacing and typing over old characters.
CONTROL C	Sends an "interrupt" signal to the most recent program. This functions differently from program to program. If the program does not make specific "interrupt" provisions, it will abort the program. If the program has provisions for interrupts, CONTROL C usually provides a way to stop the current function and return to a master menu or command mode. In the SHELL, it can be used to convert the "foreground" program to a "background" program if the program has not begun I/O to the terminal.
CONTROL D	Redisplays the current input line. This is mainly used for hardcopy terminals that can not erase deleted characters.
CONTROL E	Sends a "program abort" signal to the program presently running. This key will prematurely ABORT the current program and return you to the SHELL.
CONTROL H	Backspaces to erase previous characters. Most keyboards have a special BACKSPACE key that can be used directly without using the CONTROL key.
CONTROL Q	Resumes output that was halted by CONTROL S. The CONTROL Q function is known as XON.

KEY	FUNCTION
CONTROL S	Halts output until CONTROL Q is entered. The CONTROL S function is known as XOFF. This is a function used by many serial I/O devices (such as printers) to control output speed.
CONTROL W	Temporarily halts output so you can read the screen before data scrolls off. Output resumes when any other key is hit. See the discussion below of the "screen pause" feature.
CONTROL X	Deletes (erases) the entire current line.
ESCAPE or CONTROL [	Indicates end-of-file: All OS-9 I/O devices (including terminals) are accessed as files. This simulates the effect of reaching the end of a disk file.

end of appendix c

application programs - A program that needs an operating system environment to execute. For example, word processing, accounting, or spreadsheet programs.

ASCII - The standard code of symbols (including alphanumerics) used in a computer environment. ASCII stands for American Standard Code for Information Interchange.

attributes - A set of status codes that control access to a file for security. Also indicates if a file is a directory or not.

BACKUP - A utility provided with OS-9 that allows you to create a duplicate copy of an existing disk. Also, the copied disk.

boot (bootstrap or cold start) - A startup function that initially loads the operating system into memory and starts it after the computer is first turned or after it is reset.

byte - Unit of memory consisting of 8 binary on/off switches (bits).

cold start - see boot.

command - A request made from the keyboard for the execution of a specific operation. Also, sometimes refers to one of the utility programs set provided with OS-9.

command interpreter - Software that translates input commands into machine language commands that cause the computer to perform the requested actions. The name of OS-9's command interpreter program is "SHELL".

command line - A single line of input that includes a keyword that the operating system can understand and act upon. A command line may also include an object and the parameters of the command.

concurrent execution - The act of deliberately running a program at the same time as another program; also the effect multitasking has on programs. See also: **multitasking, sequential execution.**

data directory - A directory used by OS-9 to locate data files used by programs. The user can change at will which directory is the current data directory. See also: **directory**

data module - A type of module used for shared variable storage by two or more tasks. See also: **module.**

device descriptor module - A type of module which contains the identification and initialization values for a specific I/O device. The name of the device driver module is also the logical name by which the device is referred to by the software.

device driver module - A program module than contains the software necessary to interface OS-9 to a particular type of I/O device. A single driver module is often shared by many identical types of I/O ports (such as for terminals).

directory - A special file used by OS-9 which contains the names of other files or directories. A directory allows you to organize your files by placing all files to be grouped together in one place.

execution directory - A directory used by OS-9 to locate files containing programs (commands). The user can change which directory is his/her current execution directory at will, but usually the system-wide commands directory ("CMDS") is used. See also: **directory**

execution modifier - A character in a command line recognized by SHELL that changes the default execution of the command. Modifiers are used to change the memory size ("#"), process priority (" "), and standard I/O paths (">", "<", ">>").

file - An ordered sequence of bytes used for mass storage. A file may contain a program, text, a list of commands, etc.

file pointer - An indicator of where the next access in a file will occur.

file system - The logical organization of mass storage and all other I/O devices into a common and compatible system based on paths, files and directories.

filter - A special type of utility command program specially designed for use with pipes. A filter typically performs some useful function on the data flowing through it such as sorting, editing, etc. See also: **pipe**

FORMAT - A utility provided in OS-9 to initialize a disk before it is used. New disks must be formatted prior to being used. Also refers to the physical division of a disk into sectors, clusters, etc.

keyword - A program, procedure file, or built-in command that SHELL recognizes in a command line.

link - An OS-9 function used to request the location of a memory module of a given name prior to its use. Causes the user count of the module to be increased by one. "Unlink" is the opposite function. See also: **memory module, module directory.**

memory module - A named block of program code or data that is or can be loaded into memory. Memory modules use a special standardized format. See also: **data module, module directory, program module.**

module directory - A list automatically maintained by OS-9 of the name, location, and user count of each memory module which is present in memory. See also: **link, memory module.**

multitasking - A feature of the operating system which allows multiple programs to be run at the same time.

multiuser - A function of the operating system which allows multiple users to use the system at the same time; provides security for the system and each user's files. Sometimes referred to as timesharing.

operating system - The master control program that manages the operation of the computer and provides commonly-used functions (such as I/O) for other programs.

owner attributes - Owner read, owner write, and owner execute. An owner of a file is a user with the same group number or user ID associated with the file. The owner attributes (if set) allow access to the file by the owner. See **public attributes.**

parameters - A character or symbol recognized by SHELL in a command line that specifies additional conditions for the execution of the command.

password - A user-unique code word used to log on a timesharing system that validates identity for security.

password file - A file that contains a list of all valid user names and passwords for users on the system.

path - The routing of input or output between a program and a file or I/O device.

pathlist - A list of names that specifies the location of the file or I/O device to be associated with a path. It may in various combinations include a device name, one or more directory names, and a file name.

permission - Term used to indicate that a certain attribute is set for a file. For example, owner read permission. Also sometimes used for the term attribute.

pipe - A special type of I/O path that connects and synchronizes the standard output of a program to the standard input of another simultaneously running program. Chains of "piped" programs are called "pipelines". See also: **filter, standard I/O paths**

pipeline - See **pipe.**

procedure file - A file that contains a list of commands to be performed by the Shell as if they were typed in from a keyboard.

process - An individual running program; synonymous with task.

process ID - A unique code number assigned by OS-9 when a new process is created. It is used to identify the process in subsequent commands or system calls.

program module - A memory module which contains executable code. It is required that all OS-9 programs be kept in memory module format. See also: memory module.

public attributes - Public read, public write, public execute. The public is defined as any user not having the same user ID or group number as the file. These attributes (if set) allow anyone access to the file. See owner attributes.

RAM disk - A special device driver module that allows the part of the system's main memory to behave as a disk drive. This permits very high speed (but non-permanent) storage for small, commonly used files.

record locking - A special function built into OS-9's file management system which eliminates problems caused by two or more users trying to update the same part of a file at the same time.

redirection - A method of changing the normal input and/or output of a program to alternate files or I/O devices. This is done at the time the program is run through the use of modifiers in the command line, as opposed to the time it is written. See also: standard I/O paths

root directory - The directory entered when the user first logs on the system. This directory is specified in the password file.

separator - A special character recognized by SHELL in the command line that specifies the sequential or concurrent execution of more than one process. The special characters are: ";" (sequential) and "&" (concurrent).

sequential execution - The act of deliberately running programs one at a time in the order specified as opposed to concurrently. This is done when it is necessary for one program to be completed before the next one in a sequence is begun. See also: multitasking, sequential execution, separator.

SHELL - OS-9's command interpreter program. This program acts as an interface between the user and the operating system. See also: command interpreter.

signal - A software interrupt that can be sent from one process to another or from OS-9 to a process. For example, the "Control-E" abort key causes an "abort signal" to be sent to a program.

single user - A mode of operation where only one user utilizes the computer. A file attribute the allows only one user at a time to access the file.

standard I/O path - The default I/O path used by a program for routine input and output. Every process has three standard I/O paths: input, output, and error output. See also: path, redirection

system call - A request from a programming language that causes OS-9 to perform a specific function such as input/output.

system disk - A disk which contains the system boot file plus other common system-wide files such as the utility command set.

task - See process.

timesharing - See multiuser.

UNIX - An operating system similar to OS-9, but with less functionality and special features designed to soak up excess memory, disk space and CPU time on large, expensive computers.

user name - A name used externally to identify each user when logging on the system. Based on the contents of the password file, the system converts this name to the corresponding user ID number for subsequent internal use. See also: user ID, password file.

user ID - A unique code number used to identify the user's files and processes. See also: user name, password file.

utility - One of the set of programs supplied with OS-9 that is used to perform housekeeping, maintenance, customization, and convenience functions.

end of appendix d

INDEX

INDEX