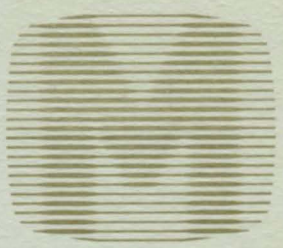


MICROWARE®

OS-9 OPERATING SYSTEM

USER'S MANUAL

Microware Systems Corporation
5835 Grand Avenue
Des Moines, Iowa 50312
Telephone 515-279-8844
Telex 910-520-2535

OS-9 OPERATING SYSTEM

USER'S MANUAL

OS-9 Operating System User's Manual

Revision G

For Use With OS-9 Level One and OS-9 Level Two

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 and should not be construed as a commitment by Microware Systems Corporation.

Publication date: January 1, 1983

Microware Systems Corporation
5835 Grand Avenue
Des Moines, Iowa 50312 U.S.A
Telephone 515-279-8844
Software Support 515-279-8898
Telex 910-520-2535

OS-9 OPERATING SYSTEM USERS MANUAL
Table of Contents

CHAPTER 4 - ADVANCED FEATURES OF THE SHELL

CHAPTER 5 - MULTIPROGRAMMING AND MEMORY MANAGEMENT

CHAPTER 6 - USE OF THE SYSTEM DISK

CHAPTER 7 - COMMAND DESCRIPTIONS

INTRODUCTION TO THE OS-9 USERS MANUAL

Welcome to the world of OS-9!

This book is designed to help you learn to use a sophisticated and powerful operating system that will enhance the performance and versatility of your 6809 computer system. Whatever you use your computer for, you will find that OS-9 will help you get the job done quickly and elegantly. And you will discover that OS-9's logical and consistent design makes it easy to learn and use.

You can use this manual as a learning guide and also as a reference guide for daily use. Programmers who use OS-9 for advanced applications should also consult the companion manual, the "OS-9 System Programmer's Manual" for more information on assembly language programming.

If you don't have experience with OS-9 or Unix, it may take some time to for you to familiarize yourself with all of OS-9's features. This manual was designed to introduce the basics in Chapters 1 and 2 so you can start using the system almost immediately. Therefore, we recommend that beginners read these chapters carefully and scan the individual command descriptions in Chapter 7. As soon as possible you should study the remainder of the book. You will find cross-references throughout the manual to help you understand the relationship between different parts of the system.

This is the fourth major revision of the basic user documentation for OS-9 since it was first released in 1980. New material relating to OS-9 Level Two and features of OS-9 Level One Version 1.2 have been added. Many of the changes in this edition reflect comments and suggestions received by Microware from hundreds of OS-9 users all over the world. We extend our thanks to the many individuals who sent thoughtful letters suggesting improvements in the OS-9 software and documentation.

AN OVERVIEW OF THE OS-9 SOFTWARE FAMILY

OS-9 can be used with almost any 6809-based computer, from single-board control systems up to fully-equipped timesharing systems. This versatility is one of OS-9's major advantages and distinguishes it from many other operating systems. In particular, OS-9 software is "portable" upward or downward, so you can develop software for a smaller target system on a larger development system.

OS-9 is available in two versions:

* OS-9 Level One is used on small- to medium-sized systems that have up to 56K bytes of memory.

* OS-9 Level Two is used on larger multiuser systems equipped with memory management hardware and up to two megabytes of memory.

From the user's point of view, both versions work almost identically, and the information in this manual applies to both versions except where otherwise noted.

Some of the features of OS-9 are:

* User-friendly Unix-like environment
* Multiuser/Multitasking Real-Time Operating System
* Extensive support for structured, modular programming
* Device-independent interrupt-driven input/output system
* Multi-level directory system

System software for OS-9 available from Microware includes:

* Basic09 - Extended Structured Basic
* Pascal - ISO Standard Pascal Compiler
* CIS Cobol - ANSI 1974 Standard Cobol Compiler
* Forms 2 - Interactive Program Generator for CTS COBOL
* C - Standard C Compiler (Unix V6 compatible)
* Macro Text Editor
* Assembler
* Interactive Debugger
* Additional OS-9 Device Drivers
* OS-9 User Source Code Package (Partial Source Code for OS-9)

1.0 HOW TO INSTALL OS-9

Because there are many different versions of OS-9 customized for 6809 computers produced by many manufacturers, this section describes the typical installation and startup procedure. If your computer was shipped from the factory with OS-9 already installed, your system should be ready to run. Otherwise, specific instructions for your version of OS-9 accompany the software package.

1.0.1 PREPARATION AND SETUP OF THE HARDWARE

Almost every circuit board in your system has a number of switches, jumpers, and sockets that define the way the hardware functions. How these are set up is critical - if even one switch is in the wrong position OS-9 may not work at all, even if the same computer previously had run some other software properly.

--> EXPERIENCE HAS SHOWN THAT MOST PROBLEMS BRINGING UP OS-9 OCCUR WHEN THE USER INCORRECTLY ASSUMES THAT THE HARDWARE IS ALREADY SET UP PROPERLY.

Set all switches and jumpers per the installation sheets supplied with the OS-9 package. Also, two (sometimes three) ROM chips must be installed on the system's CPU board. Here is a checklist for system setup:

PROCESSOR (CPU) BOARD:
* ROMs and ROM socket configuration jumpers
* memory management (DAT) system configuration
* on-card device addressing and interrupt (IRQ) enabling jumpers

DISK CONTROLLER BOARD:
* device address switches
* interrupt enable and DMA mode control (if applicable)
* default disk size and unit selection

MEMORY BOARD(S):
* Addressing switches - all system RAM should be addressed contiguously from address zero upward. OS-9 Level One usually requires a minimum of 12k RAM; Level Two requires 48K RAM.

INPUT/OUTPUT INTERFACE BOARD(S):
* device address jumper(s) or switch(s) set to correct address
* interrupt (IRQ) enable jumper(s) or switch(s) ON
* baud rate selection jumper(s) or switch(s) to match terminal

1.0.2 STARTING THE SYSTEM

To start the system insert the OS-9 System Disk in drive zero (usually the drive on the left side) and close the door, then depress the RESET switch. On Level One systems, the disk should select within several seconds. LEVEL TWO SYSTEMS HAVE A MINUTE OR TWO DELAY before the disk is selected. OS-9 will then begin its "bootstrap" loading process, which can take up to 30 seconds. When the system startup has finished, a message will be displayed on the terminal.

If the system does not start up, depress the computer's reset switch to retry the bootstrap sequence. If this is not successful after several attempts, turn the computer off and recheck the hardware setup instructions given in Section 1.0.1.

TROUBLESHOOTING HINTS

* If the disk drive never selects, the problem may be:

  -- ROMs incorrectly installed
  -- Jumpers on CPU, memory, or disk controller
  -- Defective memory

* If the disk drive selects for one or two seconds only:

  -- System disk defective (contact supplier)
  -- Defective or incorrectly configured memory

* If the disk drive selects for ten to twenty seconds, then deselects:

  -- Wrong type of System Disk
  -- Improper terminal cable, terminal interface board setup
  -- Terminal/Interface baud rate mismatch

* If the disk drive select and stays on continuously:

  -- System Disk defective
  -- Disk Controller not configured properly
  -- Defective or insufficient amount of memory

1.0.3 INITIAL EXPLORATIONS

When OS-9 first comes up, it will first display a welcoming message, and then a prompt like this:

```
OS9:
```

This indicates that the system is in an idle state and awaiting a command. The first command you should use is "setime" (some systems run "setime" automatically during startup). This is a prerequisite for multitasking and also allows OS-9 to keep track of the date and time of creation of new files and disks. To do so type:

```
setime
```

And then hit the "return" key. Enter the current date and time in the format requested. You may now wish to examine the files on the System Disk using the "dir" (for "directory") command. To do so, type:

```
dir
```

followed by a "return". OS-9 should respond with a listing of file names which should look something like this:

```
OS9Boot    startup    CMDS    SYS    DEFS
```

The file "OS9Boot" contains the OS-9 code which is loaded into memory at system startup. The file "startup" is a "command file" which is automatically run when the system starts up, and has the commands that printed the welcoming message. You may replace this startup file with your own customized version after becoming familiar with OS-9.

The last three files are "directory files". They are files that contain other file names instead of programs or data. The "dir" command has options which can give you more detailed information about each file (see 3.4 and 3.8.1). The file "CMDS" is a directory that consists of all the system commands such as "dir", "list", "format", etc. To see the files contained in this directory, enter:

```
dir cmds
```

which tells "dir" to show files on the directory file "CMDS". The directory file "SYS" contains two files: a file called "errmsg" which contains text for descriptions of error messages, and "password", which is an example password file for timesharing systems. The directory "DEFS" has several files containing symbolic definitions useful when writing assembly-language programs.

1.1 MAKING A BACKUP OF THE SYSTEM DISK

Before experimenting further with OS-9, it is wise to make one or more exact copies of your System Disk in case some misfortune befalls your one and only master System Disk. The two basic ways to make a backup copy of the System Disk are shown in the following sections.

1.1.1 FORMATTING BLANK DISKS

Before the actual backup procedure is initiated (or any fresh diskette is used by OS-9 for any purpose), the blank disk which is to become the backup disk must be initialized by OS-9's "format" command. The same command is used to format all types of diskettes including flexible disks and rigid disks such as Winchesters.

--> WARNING: MANY HARD DISKS ARE DELIVERED BY THE MANUFACTURER PRE-FORMATTED FOR OS-9 AND MAY CONTAIN IMPORTANT DATA. CHECK THE MANUFACTURER'S INSTRUCTIONS BEFORE ATTEMPTING TO FORMAT A HARD DISK.

To format a fresh disk, place the blank disk in drive one and the System Disk in drive zero, then type:

```
format /d1
```

This command initiates the "format" utility and indicated that the disk in drive one (drive one's logical name is "/d1"). The "format" program will then print a "Table of Format Variables". This table indicates the format OS-9 assumes will be used. In the line labelled "Density", single density is called as "FM", and double density is called "MFM". Other than density and number of sides, most other table variables are not of concern at this point. After the table is displayed, the program will then ask the question:

```
Formatting on drive /D1
y (yes), n (no), or q (quit)
Ready?
```

IF THE DEVICE NAME (/D1) IS NOT DISPLAYED: enter "q" to "quit" RIGHT NOW and start over, OR YOU MAY ERASE your only System Disk.

IF YOU ARE GOING TO PERFORM THE COMPLEX BACKUP PROCEDURE AND THE FORMAT TABLE IS CORRECT FOR YOUR HARDWARE: answer "y" for "yes" which will initiate the format process using the table values.

IF YOU ARE GOING TO PERFORM THE SIMPLE BACKUP PROCEDURE: answer "n" for "no". This indicates that you wish to override the assumptions listed in the Table of Format Variables. Consequently, you will be asked a series of questions regarding the desired format. Answer the questions as appropriate to make a single-sided, single-density disk

EVEN IF YOUR SYSTEM CAN USE DOUBLE SIDED AND/OR DOUBLE DENSITY DISKS. This is because OS-9 distribution System Disks are almost always shipped on single-sided, single density disks, and the "backup" command which will be used next REQUIRES that the old and new disks have the exact same format. When requested, enter "S" for single density, "1" for the number of surfaces, and the number of tracks that were given for "number of cylinders" in the "Table of Format Variables" initially displayed by the format command. After you have answered all three questions, a new table will be displayed and the "yes/no/quit" choice requested. If everything looks OK, answer "y" for yes. If you are confused, refer to the "format" command description in this book.

It usually takes several minutes for the format program to run. During the later phase of the process the hexadecimal number of each track will be displayed as each track is verified (checked for bad sectors). If any bad sectors are found, an error message will be displayed along with the number of the offending sector(s).

--> WHEN MAKING BACKUP DISKS, NEVER BACKUP TO A DISK THAT HAS ANY BAD SECTORS.

1.1.2 A SIMPLE BACKUP PROCEDURE

This method uses OS-9's "backup" command and is easiest to do. However, is has the disadvantage of making an exact copy AND FORMAT OF the OS-9 distribution System Disk, which may be of a format which has less storage capacity than the system's disk drives true capability. Despite this limitation, it is the least complicated way to make your first backup disk.

First format a blank disk following the procedure in 1.1.1. To initiate the backup process, type:

```
backup
```

The "backup" command will respond with:

```
Ready to BACKUP from /D0 to /D1 ? :
```

Now enter "y" for yes. It will then ask:

```
X is being scratched
OK ?:
```

Answer "y" for yes again, and the backup process should begin. If you get an error message at this point, it usually means the disk you formatted in the previous step was not formatted the same way at the master System Disk. The "backup" command has two phases: the first phase copies everything from drive zero to drive one checking for errors while reading from the master but not for "write" errors.

The second pass is the "verify" pass which makes sure everything was copied onto the new disk correctly. Backing up in two passes is actually faster than doing everything in one pass. If any errors are reported during the first (copy) pass, there is a problem with the master disk or its drive. If errors occur during the second (verify) pass, there is a problem with the new disk and the "backup" should be attempted until an error-free backup has been performed. If backup repeatedly fails on the second pass, reformat the disk and try to backup again. If backup fails again, the disk is physically defective.

1.1.3 A MORE COMPLEX DISK BACKUP PROCEDURE

This procedure allows you to make a working copy of the System Disk using the maximum-performance format possible on your system. This method uses a command called "dsave", which automatically creates a file of OS-9 commands which will copy all the files, one by one using the OS-9 "copy" command, from the System Disk to the new disk. The "copy" command can be used to copy from a disk to another having a different format.

First format a blank disk as outlined in 1.1.1 using the desired format. Leave the master System Disk in drive zero, and the newly formatted disk in drive one, then type:

```
dsave -b /d0 /d1 >/d0/tempfile
```

This results is the creation of a "procedure file" called "tempfile" which will include all the commands needed to copy all files and directories from drive zero to drive one. Then enter the command:

```
/d0/tempfile
```

The command "/d0/tempfile" executes the copy sequence. You will see all the copy commands generated by "dsave" displayed as they are performed. When the system finished copying all files, delete the procedure file "tempfile" by entering:

```
del tempfile
```

Once you have made the new System Disk, it you can use the "backup" command to make additional copies on disks having the same format.

2.0 INTRODUCTION TO THE SHELL

The "shell" is a command interpreter program that gives the user convenient and versatile access to OS-9's powerful capabilities. It provides an easy-to-use interface between the user and the internal functions of the operating system. The shell is entered following the system start-up, or after logging on to a timesharing terminal. You will know when the shell is waiting for input because it displays the prompt:

```
OS9:
```

This prompt indicates that the shell is active and awaiting a command from your keyboard. You can now respond by typing a "command line" followed by a carriage return (which should always be the last character on a line). It usually makes no difference whether you use upper-case letters, lower-case letters, or a combination of both as OS-9 matches letters of either case.

The command line always begins with a name of a program (which may already be present in memory) OR a pathlist (file name) which can be:

* The name of a machine language program
* The name of an executable program compiled by a high-level language such as Basic09, Pascal, Cobol, etc. (See 4.8)
* The name of a procedure file (See 4.6)

When processing the command line, the shell searches for a program having the name specified in the following sequence:

1 - If the program named is already in memory, it is run.

2 - The user's "execution directory" (usually "CMDS") is searched. If a file having the name given is found, it is loaded and run (See 5.4.1).

3 - The user's "data directory" is searched. If a file having the name given is found, it is processed as a "procedure file" which means that the file is assumed to contain one or more command lines which are processed by the shell in same manner as if they had manually typed one by one.

Mention is made above of the "data directory" and the "execution directory". At all times each user is associated with two file directories. A more detailed explanation of directories is presented in section 3.3. The execution directory includes files which are executable programs. In most systems all users share a directory called "CMDS" which contains command programs such as "dir", "list",

etc. This leads to a subtle but very important point: the shell itself does not execute commands such as "dir". Rather, most of the system commands are individual programs which are stored on files and are called by the shell when you type their names.

The name may be optionally followed by one or more "parameters" which are passed to the program called by the shell. A command line may also include one or more "modifiers" which are specifications used by the shell to alter the program's standard input/output files or memory assignments (See 4.2).

2.0.1 SENDING OUTPUT TO THE PRINTER

By default, most commands and programs display output on the terminal display. The output of these programs can alternatively be printed by specifying output redirection on the command line. This is done by including the following at the end of any command line:

```
>/p
```

The ">" character tells the shell to redirect output (See 4.3.2) to the printer, which is a device named "/P" on most systems (See 3.2). For example, to redirect the output of the "dir" command to the printer, enter:

```
dir >/p
```

2.1 SHELL COMMAND LINE PARAMETERS

Parameters are generally used to either specify file name(s) or to select options to be used by the program specified in the command line given to the shell. Parameters are separated from the command name and from each other by space characters (hence parameters and options cannot themselves include spaces). Each command program supplied with OS-9 has an individual description in the last section of this manual which describe the correct usage of the parameters of each command.

For example, the "list" program is used to display the contents of a text file on your terminal. It is necessary to indicate to the "list" program which file it is to be displayed, therefore. the name of the desired file is given as a parameter in the command line. For example, to list the file called "startup" (the system initialization procedure file), you enter the command line:

```
list startup
```

Some commands have two parameters. For example, the "copy" command is used to make an exact copy of a file. It requires two parameters: The name of the file to be copied and the name of the file which is to be the copy, for example:

```
copy startup newstartup
```

Other commands have parameters which select options. For example:

```
dir
```

shows the names of the files in the user's data directory. Normally it simply lists the file names only, but if the "e" (for entire) option is given, it will also give complete statistics for each file such as the date and time created, size, security codes, etc. To do so enter:

```
dir e
```

The "dir" command also can accept a file name as a parameter which specifies a directory file other than the (default) data directory. For example, to list file names in the directory "sys", type:

```
dir sys
```

It is also possible to specify both a directory name parameter and the "e" option, such as:

```
dir sys e
```

giving file names and complete statistics (See example in 3.8.1).

2.3 SOME COMMON COMMAND FORMATS

This section is a summary of some commands commonly used by new or casual OS-9 users, and some common formats. Refer to the individual command descriptions in Section 8 for more detailed information and additional examples. Parameters or options shown in brackets are optional. Whenever a command references a directory file name, the file must be a directory file.

CHD filename — chd DATA.DIR

Changes the current data working directory to the directory file specified.

COPY filename1 filename2 — copy oldfile newfile

Creates "filename2" as a new file, then copies all data from "filename1" to it. "filename1" is not affected.

DEL filename — del oldstuff

Deletes (destroys) the file specified.

DIR [filename] [e] [x] — dir myfiles e

List names of files contained in a directory. If the "x" option is used the files in the current execution directory are listed, otherwise, if no directory name is given, the current data directory will be listed. The "e" option selects the long format which shows detailed information about each file.

FREE devicename — free /d1

Shows how much free space remains on the disk whose name is given.

LIST filename — list script

Displays the (text) contents of the file on the terminal.

MAKDIR filename — makdir NEWFILES

Creates a new directory file using the name given. Often followed by a "chd" command to make it the new working data directory.

RENAME filename1 filename2 — rename zip zap

Changes the name of filename1 to filename2.

2.4 TERMINAL CONTROL KEY FUNCTIONS

There are a number of useful control functions that can be generated from terminal keyboards when the shell and most other OS-9 programs are running. Most of these functions use "control keys" which are generated by simultaneously depressing the key marked "control" and a regular character key. For example, if you make a mistake while entering a line, you can use the "backspace" key (CONTROL-H on most keyboards), or you can delete the entire line using the "line delete" key (usually CONTROL-X). There are several other control keys that operate as listed below. Note: it is possible to redefine which keys correspond to these functions: see the TMODE command description.

Terminal Control Keys

CONTROL A - Repeat previous input line. The last line entered will be redisplayed but not processed, with the cursor positioned at the end of the line. You may hit return to enter the line, or edit the line by backspacing, typing over characters to correct them, and entering control A again to redisplay the edited line.

CONTROL C - Program Interrupt - sends an "interrupt" signal (signal code 3) to the program presently running.

CONTROL D - Redisplay present input line

CONTROL H - Backspace

CONTROL Q - Quit Program - This key can be used to abort execution of command programs (which returns you to the shell). Sends a "program abort" signal (signal code 2) the the program presently running.

CONTROL W - display wait - This control key will temporarily halt data output to the terminal so the screen can be read before the data scrolls off. Output is resumed when any other key is hit.

CONTROL X - line delete

ESCAPE (CONTROL [) - End-of-File - This key is used to send an end-of-file to programs that read input from the terminal in place of a disk or tape file. It must be the first character on the line in order for it to be recognized.

2.5 LOGGING ON AND OFF TIMESHARING SYSTEMS

If you are using a terminal on a timesharing system other than the system's master terminal, you will probably have to "log-on" which involves entering a correct user number and password before you will be permitted access to the system.

When you log in, you must use the user name and password that was assigned to you by the person responsible for managing the system. The system manager usually is the only person who may change passwords and user names.

When a terminal is idle (i.e., between sessions), it will respond to any key by beeping, except for the "return" key which will initiate the login sequence. You can shorten the login procedure by typing your user name and password on the same line. An example of the log-in procedure is shown below:

```
OS-9 Level 2 Version 1.0 Timesharing System   11/16/81 14:32:12

User name?: peter
Password: ace

Process #6 logged  11/16/81 12:34:20

Shell

OS9:
```

If you don't enter a valid user name or password, an appropriate message will be displayed and you will be asked to reenter the user name or password. If you cannot login after three attempts the login sequence will terminate which may result in disconnection of a dial-up telephone line.

To log off, return to the shell (if you are running another program), and hit the escape "ESC" key (end-of-file). This will return the terminal to the idle state.

3.0 INTRODUCTION TO THE UNIFIED INPUT/OUTPUT SYSTEM

OS-9 has a unified input/output system in which data transfers to ALL I/O devices are performed in almost exactly the same manner, regardless of the specific hardware devices involved. It may seem that varying operational characteristics make this difficult. After all, line printers and disk drives behave much differently. However, these differences can mostly be overcome by defining a set of standardized logical functions for all devices and by making all I/O devices conform to these conventions, using software routines to eliminate hardware dependencies wherever possible. This produces a much simpler and more versatile input/output system.

OS-9's unified I/O system is based upon logical entities called "I/O paths". Paths are analogous to "software I/O channels" which can be routed from a program (process) to a mass-storage file or any other I/O device. All input/output operations require use of paths.

The behavior of paths (as seen by programs) are generally uniform, data transferred through paths may be processed by OS-9 to conform to the hardware requirements of the specific I/O device involved. Data transfers can be either bidirectional (read/write) or unidirectional (read only or write only), depending on the device and/or how the path was established.

Data transferred through a path is considered to be a stream of 8-bit binary bytes that have no specific type or value: what the data actually represents usually depends on how it is used by each program.

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

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

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

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

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

3.1 RULES FOR CONSTRUCTING PATHLISTS

Whenever a path is established (or "opened"), OS-9 must be given a description of the "routing" of the path. This description is given in the form of a character string called a "pathlist". It specifies a particular mass-storage file, directory file, or any other I/O device. OS-9 "pathlists" are similar to "filenames" used by other operating systems.

The name "pathlist" is used instead of "pathname" or "filename" because in many cases it is a list consisting of more than one name to specify a particular I/O device or file. In order to convey all the information required, a pathlist may include a device name, one or more directory file names and a data file name. Each name within a pathlist is separated by slash "/" characters.

Names are used to describe three kinds of things:

* Names of Physical I/O Devices
* Names of Regular Files
* Names of Directory Files

Names can have one to 29 characters, all of which are used for matching. They must begin with an upper- or lower-case letter followed by any combination of the following characters:

uppercase letters: A - Z
lowercase letters: a - z
decimal digits: 0 - 9
underscore: _
period: .

Here are examples of legal names:

raw.data.2	project_review.backup
reconciliation.report	X042953
RJJones	search.bin

Here are examples of illegal names:

22November	(does not start with a letter)
max*min	(* is not a legal character)
.data	(does not start with a letter)
open orders	(cannot contain a space)
this.name.obviously.has.more.than.29.characters	(too long)

3.2 I/O DEVICE NAMES

Each physical input/output device supported by the system must have a unique name. The actual names used are defined when the system is set up and cannot be changed while the system is running. Although the specific device names used on a particular system are somewhat 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
D0	- Disk drive unit zero
D1	- Disk drive unit one
D2, D3, etc.	- Other 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 that of an I/O device. If the device is not a mass-storage multifile device the device name is the only name allowed. This is true for devices such as terminals, printers, etc. Some examples of of pathlists that refer to I/O devices are:

```
/TERM
/P
/modem3
```

I/O device names are actually the names of the "device descriptor modules" kept by OS-9 in an internal data structure called the "module directory" (See the OS-9 System Programmer's manual for more information about device driver and descriptor modules). This directory is automatically set up during OS-9's system start up sequence, and updated as modules are added or deleted while the system is running.

3.3 MULTIFILE DEVICES AND DIRECTORY FILES

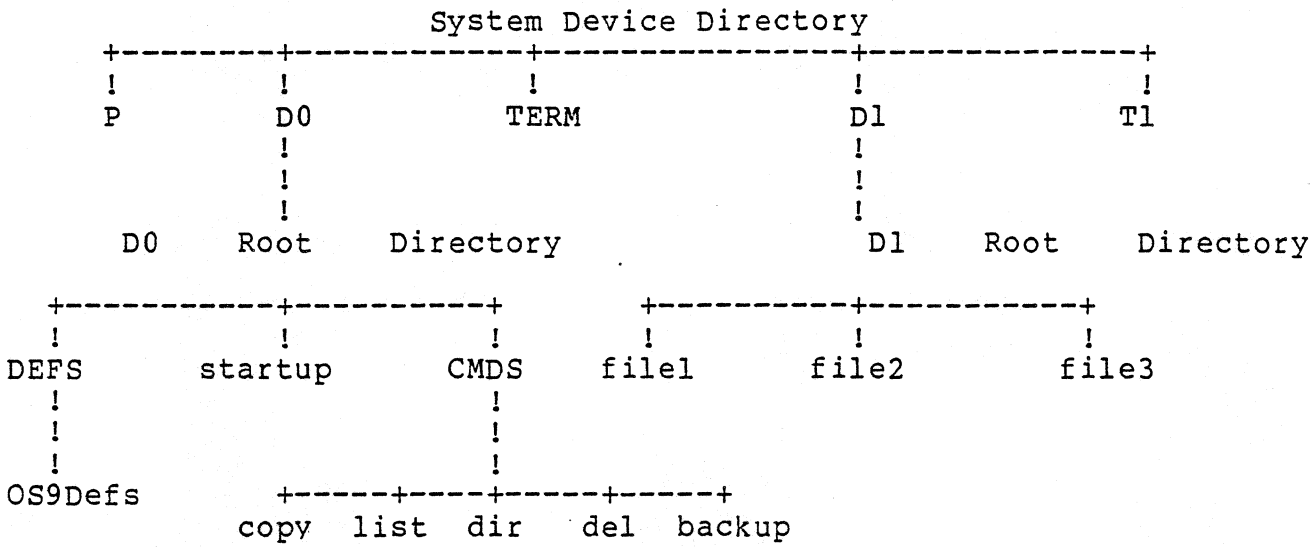
Multifile devices are mass storage devices (usually disk systems) that store data organized into separate logical entities called "files". Each file has a name which is entered in a directory file. Every multifile device has a master directory (called the "root directory") that includes the names of the files and sub-directories stored on the device. The root directory is created automatically when the disk is initialized by the "format" command (see 1.1.1).

Pathlists that refer to multifile devices may have more than one name. For example, to refer to the file "mouse" whose name appears in the root directory of device "D1" (disk drive one) the following pathlist is used:

```
/d1/mouse
```

When OS-9 is requested to establish an I/O path, it uses the names in the pathlist sequentially from left to right to search various directories to obtain the necessary routing information. These directories are organized as a tree-structured hierarchy. The highest-level directory is called the "device directory", which contains names and linkages to all the I/O devices on a given system. If any of the devices are of a multifile type they each have a root directory, which is the next-highest level.

The diagram below is a simplified file system tree similar to typical OS-9 systems. Note that device and directory names are capitalized and ordinary file names are not. This is a customary (but not mandatory) practice which allows you to easily identify directory files using the short form of the "dir" command.



The device names in this example system are "TERM", "T1", "P", "D0" and "D1". The root directory of device "D0" includes two directory files, DEFS and CMDS, and one ordinary file "startup". Notice that device "D1" has in its root directory three ordinary files. In order to access the file "file2" on device "d1", a pathlist having two names must be used:

```
list /d1/file2
```

To construct a pathlist to access the file "dir" on device "d0" it is necessary to include in the pathlist the name of the intermediate directory file "CMDS". For example, to copy this file requires a pathlist having three names to describe the "from" file:

```
copy  /d0/cmds/dir  temp
```

3.4 CREATING AND USING DIRECTORIES

It is possible to create a virtually unlimited number of levels of directories on a mass storage device using the "makdir" command. Directories are a special type of file (see 3.8.1). They can be processed by the same I/O functions used to access regular files which makes directory-related processing fairly simple.

To demonstrate how directories work, assume that the disk in drive one ("d1") has been freshly formatted so that it has a root directory only. The build command can be used to create a text file on "d1". The build command will print out "?" as a prompt to indicate that it is waiting for a text line to be entered. It will place each line into the text file until an empty line with only a carriage return is entered, as shown below:

```
OS9: build /d1/file1
? This is the first file that
? we created.
? [RETURN]
```

The "dir" command will now indicate the existence of the new file:

```
OS9: dir /d1

   Directory of /d1   15:45:29
file1
```

The "list" command can be used to display the text stored in the file:

```
OS9: list /d1/file1

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

The "build" command again is again used to create two more text files:

```
OS9: build /d1/file2
? This is the second file
? that we created.
? [RETURN]

OS9: build /d1/file3
? This is another file.
? [RETURN]
```

The dir command will now show three file names:

```
OS9: dir /d1
```

```
   Directory of /d1  15:52:29
filel           file2           file3
```

To make a new directory in this directory, the "makdir" command is used. The new directory will be called "NEWDIR". Notice that throughout this manual directory names are always capitalized. This is not a requirement of OS-9 (see 3.1). Rather, it is a practice popular with many OS-9 users because it allows easy identification of directory files at all times (assuming all other file names use lower-case letters).

```
OS9: makdir /d1/NEWDIR
```

The directory file "NEWDIR" is now a file listed in d1's root directory:

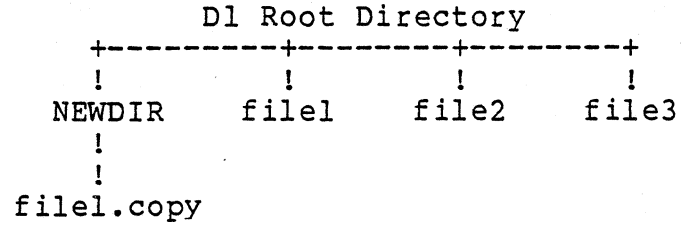
```
OS9: dir /d1

   Directory of /d1  16:04:31
filel           file2           file3           NEWDIR
```

Now we will create a new file and put in the new directory using the COPY command to duplicate "filel":

```
OS9: COPY /d1/filel /d1/NEWDIR/filel.copy
```

Observe that the second pathlist now has three names: the name of the root directory ("D1"), the name of the next lower directory ("NEWDIR"), then the actual file name ("filel.copy"). Here's what the directories look like now:



The dir command can now show the files in the new directory:

```
OS9: dir /d1/NEWDIR

   Directory of /d1/NEWDIR
filel.copy
```

It is possible to use "makdir" to create additional new directories in "NEWDIR", and so on, limited only by available disk space.

3.5 DELETING DIRECTORY FILES

The "del" command cannot be used to directly delete a directory file. If a directory file that still contained file names were to be deleted, OS-9 would have no way to access the files or to return their storage to the unallocated storage pool. Therefore, the following sequence must be performed to delete a directory file:

1 - All file names in the directory must be deleted.

2 - The "attr" command is used to turn off the files directory attribute (-d option), making it an ordinary file (see 3.8).

3 - The file may now be deleted using the "del" command.

3.6 ADDITIONAL INFORMATION ABOUT DIRECTORIES

The OS-9 directory system is very useful because it allows each user to privately organize files as desired (by project, function, etc.), without affecting other files or other user's files. Another advantage of the hierarchical directory system is that files with identical names can be kept on the same device as long as the names are in different directories. For example, you can have a set of test files to check out a program using the same file names as the program's actual working files. You can then run the program with test data or actual data simply by switching directories.

Here are some important characteristics relating to use of directory files:

-- Directories have the same ownership and security attributes and rules as regular files. See Section 3.6.

-- The name of a given file appears in exactly one directory.

-- Files can only be added to directories when they are created.

-- A file and the directory in which its name is kept must reside on the same device.

3.7 USING AND CHANGING WORKING DIRECTORIES

Each program (process) has two "working directories" associated with it at all times: a "data directory" and an "execution directory". The working directory mechanism allows the name searching involved in pathlist processing to start at any level (subtree) of the file system hierarchy. Any directory that the user has permission to access (see 3.8) can be made a working directory.

The rules used to determine whether pathlists refer to the current working directory or not are simple:

---> When the first character of a pathlist IS a "/", processing of the pathlist starts at the device directory, e.g., the first name MUST be a device name.

---> When the first character of a pathlist IS NOT a "/", processing of the pathlist starts at the current working directory.

Notice that pathlists starting with a "/" must be complete, in other words, they must have all names required to trace the pathlist from the device directory down through all intermediate directories (if any). For example:

```
/d2/JOE/WORKINGFILES/testresults
```

On the other hand, use of the current working directory allows all names in the file hierarchy tree to be implied instead of explicitly given. This not only makes pathlists shorter, but allows OS-9 to locate files faster because (typically) fewer directories need be searched. For example, if the current working directory is "/d1/PETE/GAMES" and a pathlist is given such as:

```
baseball
```

the actual pathlist implied is:

```
/d1/PETE/GAMES/baseball
```

Pathlists using working directories can also specify additional lower-level directories. Referring to the example above, the pathlist:

```
ACTION/racing
```

implies the complete pathlist:

```
/D1/PETE/GAMES/ACTION/racing
```

3.7.1 Automatic Selection of Working Directories

Recall that two working directories are referred to as the "current execution directory" and the "current data directory". The reason two working directories are maintained is so that files containing programs can be organized in different directories than files containing data. OS-9 automatically selects either working directory, depending on the usage of the pathlist:

---> OS-9 will search the execution directory when it attempts to load files into memory assumed to be executable programs. This means that programs to be run as commands or loaded into memory must be in the current execution directory (See 5.4.1).

---> The data directory is used for all other file references (such as text files, etc.)

Immediately after startup, OS-9 will set the data directory to be (the root directory of) the system disk drive (usually "d0"), and the working directory to be a directory called "cmds" on the same drive ("/d0/cmds"). On timesharing systems, the "login" command selects the initial execution and data directories to the file names specified in each user's information record stored in the system password file.

Here is an example of a shell command statement using the default working directory notation, and its equivalent expansion:

```
copy file1 file2
```

If the current execution directory is "/d0/CMDS" and the current data directory is "/d0/JONES", the same command, fully expanded to show complete pathlists implied is:

```
OS9: /d0/CMDS/copy /d0/JONES/file1 /d0/JONES/file2
```

Notice that the first pathlist "copy" expands to the current working directory pathlist because it is assumed to be an executable program but the two other file names expand using the data directory because they are not assumed to be executable.

3.7.2 Changing Current Working Directories

The built-in shell commands "chd" and "chx" can be used to independently change the current working data and execution directories, respectively. These command names must be followed by a pathlist that describes the new directory file. You must have permission to access the directory according to normal file security rules (See 3.8). Here are some examples:

```
OS9: chd /d1/MY.DATAFILES

OS9: chx /d0/TESTPROGRAMS
```

When using the CHD or CHX commands, pathlists work the same as they do for regular files, except for the last name in the pathlist must be a directory name. If the pathlist begins with a "/" , OS-9 will begin searching in the device directory for the new working directory, otherwise searching will begin with the present directory (See 3.6). For example, the following sequence of commands set the working directory to the same file:

```
OS9: CHD /d1/SARAH
OS9: CHD PROJECT1

OS9: CHD /d1/SARAH/PROJECT1     (same effect as above)
```

3.7.3 Anonymous Directory Names

Sometimes is useful to be able to refer to the current directory or the next higher-level directory, but its name (full pathlist) may not be known. Because of this, special "name substitutes" are available. They are:

- `.` refers to the present working directory
- `..` refers to the directory that contains the name of the present directory (e.g., the next highest level directory)

These can be used in place of pathlists and/or the first name in a pathlist. Here are some examples:

`OS9: dir .`	lists file names in the working data directory
`OS9: dir ..`	lists names in the working data directory's parent directory.
`OS9: DEL ../temp`	deletes the file "temp" from the working data directory's parent directory.

The substitute names refer to either the execution or data directories, depending on the context in which they are used (See 3.7.1). For example, if ".." is used in a pathlist of a file which will be loaded and/or executed, it will represent the parent directory of the execution directory. Likewise, if "." is used in a pathlist describing a program's input file. it will represent the current data directory.

3.8 THE FILE SECURITY SYSTEM

Associated with each file (including directory files) are properties called ownership and attributes which who may access the file and how it many be used.

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

Usage and security functions are based on "attributes", which define how and by whom the file can be accessed. There are a total of seven attributes, each of which can be turned "off" or "on" independently. The "d" attribute is used to indicate (when on) that the file is a directory file. The other six attributes control whether the file can be read, written to, or executed, by either the owner or by the "public" (all other users). Specifically, these six attributes are:

WRITE PERMISSION FOR OWNER: If on, the owner may write to the file or delete it. This permission can be used to protect important files from accidental deletion or modification.

READ PERMISSION FOR OWNER: If on, the owner is allowed to read from the file. This can be used to prevent "binary" files from being used as "text" files (See 3.9)

EXECUTE PERMISSION FOR OWNER: If on, the owner can load the file into memory and execute it. Note that the file must contain one or more valid OS-9 format memory modules in order to actually load (See 3.9.4 and 5.4.1).

The following "public permissions" work the same way as the "owner permissions" above but are applied to processes having DIFFERENT user numbers than the file's owner.

WRITE PERMISSION FOR PUBLIC - If on, any other user may write to or delete the file.

READ PERMISSION FOR PUBLIC - If on, any other user may read (and possibly copy) the file.

EXECUTE PERMISSION FOR PUBLIC - If on, any other user may execute the file.

For example, if a particular file had all permissions on except "write permit to public" and "read permit to public", the owner would have unrestricted access to the file. but other users could execute it, but not read, copy, delete, or alter it.

3.8.1 Examining and Changing File Attributes

The "DIR" command may be used to examine the security permissions of the files in any particular directory when the "e" option is used. An example using the "dir e" command to show the detailed attributes of the files in the current working directory is:

```
   Directory of .  10:20:44

Owner Last Modified Attributes Sector Bytecount Name
----- ------------- ---------- ------ --------- ----
    1  81/05/29 1402  --e--e-r     47        42 file1
    0  81/10/12 0215  ---wr-wr     48        43 file2
    3  81/04/29 2335  -s----wr     51        22 file3
    1  82/01/06 1619  d--wr-wr     6D       800 NEWDIR
```

This display is fairly self-explanatory. The "attributes" column shows which attributes are currently on by the presence or absence of associated characters in the following format:

```
dsewrewr
```

The character positions correspond to from left to right: directory; sharable: public execute; public write; public read; owner execute; owner write; owner read. The "attr" command is used to examine or change a file's attributes. Typing "attr" followed by a file name will result in the present attributes to be displayed, for example:

```
OS9: attr file2
-s-wr-ewr
```

If the command is used with a list of one or more attribute abbreviations, the file's attributes will be changed accordingly (if legal). For example, the command:

```
OS9: attr file2  pw pr -e -pe
```

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

The "directory" attribute behaves somewhat differently than the read, write, and execute permissions. This is because it would be quite dangerous to be able to change directory files to normal files (See 3.5), and creation of a directory requires special initialization (See 3.4). Therefore, the "attr" command cannot be used to turn the directory (d) attribute on (only "makdir" can), and can be used to turn it off only if the directory is empty.

The "sharable" attribute, when on, indicates a file cannot be accessed by two or more tasks simultaneously (see 3.9.7).

3.9 READING AND WRITING FROM FILES

A single file type and format is used for all mass storage files. Files store an ordered sequence of 8-bit bytes. OS-9 is not usually sensitive to the contents of files for most functions. A given file may store a machine language program, characters of text, or almost anything else. Data is written to and read from files exactly as given. The file can be any size from zero up to the maximum capacity of the storage device. and can be expanded or shortened as desired.

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

Any part of a file can also be read or written in non-sequential order by using a function called "seek" to reposition the file pointer to any byte address in the file. This is used when random access of the data is desired.

To expand a file. you can simply write past the previous end of the file. Reading up to the last byte of a file will cause the next "read" request to return an end-of-file status.

3.9.1 File Usage in OS-9

Even though there is physically only one type of file. the logical usage of files in OS-9 covers a broad spectrum. Because all OS-9 files have the same physical type. commands such as "copy", "del", etc., can be used with any file regardless of its logical usage. Similarly, a particular file can be treated as having a different logical usage at different times by different programs. The main usage of files covered in this section are:

TEXT
RANDOM ACCESS DATA
EXECUTABLE PROGRAM MODULES
DIRECTORIES
MISCELLANEOUS

3.9.2 Text Files

These files contain variable-length sequences ("lines") of ASCII characters. Each line is terminated by a carriage return character. Text files are used for program source code, procedure files, messages, documentation, etc. The OS-9 Macro Text Editor operates on this file format.

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

The content of text files may be examined using the "list" command.

3.9.3 Random Access Data Files

Random-access data files are created and used primarily from within high-level languages such as Basic09, Pascal, C, and Cobol. In Basic09 and Pascal, "GET", "PUT", and "SEEK" functions operate on random-access files.

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

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

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

3.9.4 Executable Program Module Files

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

OS-9 program modules resident in memory have a standard module format that, besides the object code, includes a "module and a CRC check value. Program module(s) stored in files contain exact binary copies of the programs as they will exist in memory, and not one byte more (See 5.4.1). OS-9 does not require a "load record" system commonly used by other operating systems because OS-9 programs are position-independent code and therefore do not have to be loaded into specific memory addresses.

In order for OS-9 to load the program module(s) from a file. the file itself must have execute permission (See 3.8) and each module must have a valid module header and CRC check value. If a program module has been altered in any way, either as a file or in memory, its CRC check value will be incorrect And OS-9 will refuse to load the module. The "verify" command can be used to check the correctness of the check values, and update them to corrected values if necessary.

On Level One systems, if a file has two or more modules, they are treated as independent entities after loading and reside at different memory regions. On Level Two systems, two or more modules loaded from a the same file comprise a "group", are always assigned contiquous memory locations, and are treated somewhat collectively. (See 5.4.2)

Like other files that contain "binary" data, attempts to "list" program files will result in the display of random characters on the terminal giving strange effects. The "dump" command can be used to safely examine the contents of this kind of file in hexadecimal and controlled ASCII format.

3.9.5 Directory Files

Directory files play a key role in the OS-9 file system. Sections 3.3 through 3.7 of this chapter describe how they are used by various OS-9 features.

Directory files can only be created by the "makdir" command, and can be identified by the "d" attribute being set (see 3.8.1). The file is organized into 32-byte records. Each record can be a directory entry. The first 29 bytes of the record is a string of characters which is the file name. The last character of the name has its sign bit (most significant bit) set. If the record is not in use the first character position will have the value zero. The last three bytes of the record is a 24-bit binary number which is the logical sector number where the file header record (see 3.10) is located.

The "makdir" command initializes all records in a new directory to be unused entries except for the first two entries. These entries have the names "." and ".." along with the logical sector numbers of the directory and its parent directory, respectively (see 3.7.3).

Directories cannot be copied or listed - the "dir" command is used instead. Directories also cannot be deleted directly (see 3.5).

3.9.6 Miscellaneous File Usages

OS-9's basic file functions are so versatile it is possible to devise an almost unlimited number of special-purpose file formats for particular applications, which do not fit into any of the three previously discussed categories.

Examples of this category are COBOL Indexed Sequential (ISAM) files and some special word processor file formats which allow random access of text lines (See 3.9.2). As discussed in Sec. 3.9.1, most OS-9 utility commands work with any file format including these special types. In general, the "dump" command is the preferred method for examining the contents of unusually formatted files.

3.9.7 Record Lockout (Level Two Only)

When a file is accessed by two or more processes simultaneously, the possibility exists that they may attempt to update the same record of the file at the same time, with unpredictable results. To avoid this potential problem, OS-9 Level Two automatically "locks" sections of all files opened in "update" mode. The lock covers any disk sectors containing the bytes last read by each process accessing the file. If another process attempts to access a locked portion of a file, the process is put to sleep until the area is no longer locked.

4.0 ADVANCED FEATURES OF THE SHELL

The basic shell functions were introduced in Section 2 in order to provide an understanding of how basic OS-9 commands work. In this section the more advanced capabilities of the shell are discussed. In addition to basic command line processing, the shell has functions that facilitate:

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

There is a virtually unlimited combination of ways these capabilities can be used, and it is impossible to give more than a representative set of examples in this manual. You are therefore encouraged to study the basic rules, use your imagination, and explore the possibilities on your own.

4.1 A MORE DETAILED DESCRIPTION COMMAND LINE PROCESSING

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

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

Note that only the verb (the program or command name) need be present, the other parts are optional After the verb has been identified, the shell processes modifiers (if any). Any other text not yet processed is assumed to be parameters and passed to the program called.

Unless the verb is a "built-in command", the shell will run the program named as a new process (task). It then deactivates itself until the program called eventually terminates, at which time it gets another input line, then the process is repeated. This happens over and over until an end-of-file condition is detected on the shell's input path which causes the shell to terminate its own execution.

Here is a sample shell line which calls the assembler:

```
asm sourcefile 1 -o >/p #12k
```

In this example:

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

The verb must be the first name 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. Otherwise, the shell assumes it is a program name and attempts to locate and execute it as described in Sections 5.3 and 5.4.1.

4.2 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. Characters which comprise modifiers are stripped from the part(s) of the command line passed to the program as parameters, therefore, the characters reserved for use as modifiers (# ; ! < > &) cannot be used inside parameters, but can be used before or after the parameters.

4.2.1 Alternate Memory Size Modifier

When command programs are invoked by the shell, they are allocated the minimum amount of working RAM memory specified in the program's module header. A module header is part of all executable programs and holds the program's name, size, memory requirements, etc. (See 5.4). Sometimes it is desirable to increase this default memory size. Memory can be assigned in 256-byte pages using the modifier "#n" where n is the decimal number of pages, or in 1024 byte increments using the modifier "#nK". The two examples below behave identically:

```
OS9: copy #8 file1 file2     (gives 8*256 = 2048 bytes)
OS9: copy #2K file1 file2    (gives 2*1024 = 2048 bytes)
```

4.2.2 I/O Redirection Modifiers

The second kind of modifier is used to redirect the program's "standard I/O paths" to alternate files or devices. Well-written OS-9 programs use these paths for routine I/O. Because the programs do not use specific file or device names, it is 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 the standard I/O paths as defined below:

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

STANDARD OUTPUT PATH: This path is normally used to output data from the program to the 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 sometimes misleading since many other kinds of messages besides errors are sent on this path.

When new processes are created, they inherit their parent process' standard I/O paths (See 5.3). Therefore, when the shell creates new processes, they usually inherit its standard I/O paths. When you log-on 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 as given below:

<	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 pathlist describing the file or device the I/O is to be redirected to or from. For example, the standard output of "list" can be redirected to write to the system printer instead of the terminal:

```
OS9: 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 file "/D1/savelisting":

```
OS9: DIR >/D1/savelisting
```

If the LIST command is used on the file "/D1/savelisting", output from the DIR command will be displayed as shown below:

```
OS9: LIST /D1/savelisting

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

Redirection modifiers can be used before and/or after the program's parameters, but each modifier can only be used once.

4.3 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 means that one program must complete its function and terminate before the next program is allowed to begin execution. Concurrent execution means that several programs are allowed to begin execution and run simultaneously.

4.3.1 Sequential Execution

Programs are executed sequentially when each is entered on a separate line. More than one program can be specified on a single shell command line by separating each <program name> <parameters> from the next one with a ";" character. For example:

```
OS9: COPY myfile /D1/newfile ;  DIR >/p
```

This command line will first execute the COPY command and then the DIR command.

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), otherwise, ";" and "return" are identical separators.

Here are some more examples:

```
OS9: copy oldfile newfile; del oldfile; list newfile

OS9: dir >/d1/myfile ; list temp >/p; del temp
```

All programs executed sequentially are in fact separate, child processes of the shell (See 5.3). After initiating execution of a program to be executed sequentially, the shell enters the "wait" state (See 5.2) until execution of the called program terminates.

4.3.2 Concurrent Execution

The second kind of separator is the "&" which implies concurrent execution, meaning that the program is run (as a separate, child process, see 5.3), but the shell does not wait for it to complete before processing the next command.

The concurrent execution separator is therefore the means by which multiprogramming (running two or more programs simultaneously) is accomplished. 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 versus the memory requirements of the specific programs. Here is an example:

```
OS9: DIR >/P&
&007

OS9:
```

This command line will cause shell to start the DIR command executing, print the process ID number (&007), and then immediately display the "OS9:" prompt and wait for another command to be entered. Meanwhile the DIR command will be busy sending a directory listing to the printer. You can display a "status summary" of all processes you have created by using the PROCS command. Below is another example:

```
OS9: DIR >/P&  LIST file1&  COPY file1 file2 ; DEL temp
```

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

4.3.3 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".

Pipes are 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.

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

Pipelines are created by the shell when an input line having one or more "!" separators is processed. For each "!", the standard output of the program named to the left of the "!" is redirected via a pipe to the standard input of the program named to the right of the "!". Individual pipes are created for each "!" present. For example:

```
OS9: update <master_file ! sort ! write report >/p
```

In the example above, the program "update" has its input redirected from a path called "master_file". Its standard output becomes the standard input for the program "sort". Its 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 cannot flow through a pipeline any faster than the slowest program can process it. Some of the most useful applications of pipelines are jobs like character set conversion, print file formatting, data compression/decompression, etc. Programs which are designed to process data as components of a pipeline are often called "filters". The "tee" command, which uses pipes to allow data to be simultaneously "broadcast" from a single input path to several output paths, is a useful filter.

4.4 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:

```
OS9: (dir /d0; dir /d1) >/p
```

gives the same result as:

```
OS9: dir /d0 >/p; dir /d1 >/p
```

except for the subtle difference that 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, such as:

```
OS9: (del file1; del file2; del file3)&
```

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

```
OS9: (dir CMDS; dir SYS) ! makeuppercase ! transmit
```

4.5 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 the shell. These commands are executed without loading a program and creating a new process, and generally affect how the shell operates. They can be used at the beginning of a line, or following any program separator (";", "&", or "!"). Two or more adjacent built-in commands can be separated by spaces or commas.

The built-in commands and their functions are:

chd <pathlist>	change the working data directory to the directory specified by the pathlist (see 3.6).
chx <pathlist>	change the working execution directory to the directory specified by the pathlist (see 3.6).
ex name	directly execute the module named. This transforms the shell process so it ceases to exist and a new module begins execution in its place.
w	wait for any process to terminate.
* text	comment: "text" is not processed.
kill <proc ID>	abort the process specified.
setpr <proc ID> <priority>	changes process' priority (see 5.1).
x	causes shell to abort on any error (default)
-x	causes shell not to abort on error (See 4.7)
p	turns shell prompt and messages on (default)
-p	inhibits shell prompt and messages
t	makes shell copy all input lines to output
-t	does not copy input lines to output (default)

The change directory commands switch the shell's working directory and, by inheritance, any subsequently created child process. The "ex" command is used where the shell is needed to initiate execution of a program without the overhead of a suspended "shell" process. The name used is processed according to standard shell operation, and modifiers can be used.

4.6 SHELL PROCEDURE FILES

The shell is a reentrant program that can be simultaneously executed by more than one process at a time. As is the case with most other OS-9 programs, it uses standard I/O paths for routine input and output (see 4.2.3). Specifically, it requests command lines from the standard input path and writes its prompts and other data to the standard error path.

The shell can start up another process also running the shell by means of the "shell" command. If the standard input path is redirected to a mass storage file. the new "incarnation" of the shell can accept and execute command lines from the file instead of a terminal keyboard. The text file (see 3.9.2) to be processed is called a "procedure file". It contains one or more command lines that are identical to command lines that are manually entered from the keyboard. This technique is sometimes called "batch" or "background" processing.

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 data directory for a file with the desired name. If one is found, shell will automatically execute it as a procedure file (see 2.0).

Execution of 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".

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

Here are two simple ways to use the shell to create another shell:

```
OS9: shell <procfile

OS9: procfile
```

Both do exactly the same thing: execute the commands of the file "procfile". To run the procedure file in a "background" mode you simply add the ampersand operator:

```
OS9: procfile&
```

OS-9 does not have any constraints on the number of jobs that can be

simultaneously executed as long as there is memory available (see 5.4). Also, the procedure files can themselves cause sequential or concurrent execution of additional procedure files Here's a more complex example of initiating two processing streams with redirection of each shell's output to files:

```
OS9: proc1 T >>stat1&  proc2 T >>stat2&
```

Note that the built-in command "T" (copy input lines to error output) was used above. They make the output file contain a record of all lines executed, but without useless "OS9" prompts intermixed. The "-x" built-in command can be used if you do not want processing to stop if an error occurs. Note that the built-in commands only affect the shell that executes them, and not any others that may exist.

4.7 ERROR REPORTING

Many programs (including the shell) use OS-9's standard error reporting function, which displays an error number on the error output path. The standard error codes are listed in the Appendix of this manual. If desired, the "printerr" command can be executed, which replaces the smaller, built-in error display routine with a larger (and slower) routine that looks up descriptive error messages from a text file called "/d0/sys/errmsg". Once the "printerr" command has been run it cannot be turned off. Also, its effect is system-wide.

Programs called by the shell can return an error code in the MPU "B" register (otherwise B should be cleared) upon termination. This type of error, as well as errors detected by the shell itself, will cause an error message to be displayed and processing of the command line or procedure file to be terminated unless the "-x" built-in command has been previously executed (See 4.5).

4.8 RUNNING COMPILED INTERMEDIATE CODE PROGRAMS

Before the shell executes a program, it checks the program module's language type. If its type is not 6809 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. Most versions of shell call Basic09 when appropriate, and Level Two versions of shell can also call the Pascal P-code interpreter (PascalN), or the CIS Cobol runtime system (RunC).

For example, if you wanted to run a BASIC09 I-code module called "adventure", you could type the command given below:

```
OS9: BASIC09 adventure
```

Or you could accomplish the same thing by typing the following:

```
OS9: adventure
```

4.9 SETTING UP TIMESHARING SYSTEM PROCEDURE FILES

OS-9 systems 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. A procedure file which initiates the timesharing monitor for each terminal is executed to start up the system. The procedure file first starts the system clock, then initiates concurrent execution of a number of processes that have their I/O redirected to each timesharing terminal.

5.0 MULTIPROGRAMMING AND MEMORY MANAGEMENT

This section discusses OS-9's multiprogramming (sometimes called "multitasking") functions. An integral part of this discussion is the more general topic of resource management.

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

CPU Time
Memory
The input/output system

In order for the computer to have reasonable performance, these resources must be managed in the most efficient manner possible. Therefore, OS-9 uses many techniques and strategies to optimize system throughput and capacity in order to derive the best performance possible.

5.1 PROCESSOR TIME ALLOCATION AND TIMESLICING

CPU time is a finite resource that must be allocated wisely to maximize the computer's throughput. It is characteristic of many programs to spend much unproductive time waiting for various events, such as an input/output operation. A good example is an interactive program which communicates with a person at a terminal on a line-by line basis. Every time the program has to wait for a line of characters to be typed or displayed, it (typically) cannot do any useful processing and would waste CPU time. An efficient multi-programming operating system such as OS-9 automatically assigns CPU time to only those programs that can effectively use the time.

OS-9 uses a technique called timeslicing which allows processes to share CPU time with all other active processes. Time-slicing is implemented using both hardware and software functions. The system's CPU is interrupted by a real time clock many (typically 10 to 100) times each second. This basic time interval is called a "tick", hence, the interval between ticks is a time slice. This technique is called timeslicing because each second of CPU time is sliced up to be shared among several processes. This happens so rapidly that to a human observer all processes appear to execute continuously, unless the computer becomes overloaded with processing. If this happens, a noticeable delay in response to terminal input may occur, or "batch" programs may take much longer to run than they ordinarily do. At any occurrence of a tick, OS-9 can suspend execution of one program and begin execution of another. The starting and stopping of programs is done in a manner that does not affect the program's execution. How frequently a process is given time slices depends upon its assigned priority relative to the assigned priority of other active processes.

The percentage of CPU time assigned to any particular process cannot be exactly computed because there are dynamic variables such as time the process spends waiting for I/O devices. It can be roughly approximated by dividing the process's priority by the sum of the priority numbers of all processes:

$$\text{Process CPU Share} = \frac{\text{Process Priority}}{\text{Sum of All Active Process' Priorities}}$$

5.2 PROCESS STATES

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

ACTIVE: processes which can currently perform useful processing. These are the only processes assigned CPU time.

WAITING: processes which have been suspended until another process terminates. This state is used to coordinate execution of sequential programs. The shell, for example, will be in the waiting state during the time a command program it has initiated is running.

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

Sleeping and waiting processes are not given CPU time until they change to the active state.

5.3 CREATION OF NEW PROCESSES

The sequence of operations required to create a new process and initially allocate its resources (especially memory) are automatically performed by OS-9's "fork" function. If for any reason any part of the sequence cannot be performed the fork is aborted and the prospective parent is passed an appropriate error code. The most frequent reason for failure is unavailablity of required resources (especially memory) or when the program specified to be run cannot be found. A process can create many new processes, subject only to the limitation of the amount of unassigned memory available.

When a process creates a new process, the creator is called the "parent process", and the newly created process is called the "child process". The new child can itself become a parent by creating yet another process. If a parent process creates more than one child process, the children are called "siblings" with respect to each other. If the parent/child relationship of all processes in the system is examined, a hierarchical lineage becomes evident. In fact, this hierarchy is a tree structure that resembles a family tree. The "family" concept makes it easy to describe relationships between processes, and so it is used extensively in descriptions of OS-9's multiprogramming operations.

When the parent issues a fork request to OS-9, it must specify the following required information:

-- A PRIMARY MODULE, which is the name of the program to be executed by the new process. The program can already be present in memory, or OS-9 may load it from a mass storage file having the same name (see 5.4.1).

-- PARAMETERS, which is data specified by the parent to be passed to and used by the new process. This data is copied to part of the child process' memory area. Parameters are frequently used to pass file names, initialization values, etc. The shell passes command line parameters this way (see 4.1).

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

-- A USER NUMBER which is used by the file security system and is used to identify all processes belonging to a specific user (this is not the same as the "process ID", which identifies a specific process). This number is usually obtained from the system password file when a user logs on. The system manager

always is user number zero (see 3.8).

-- STANDARD INPUT AND OUTPUT PATHS: the three paths (input, output, and error/status) used for routine input and output. Note that most paths (files) may be shared simultaneously by two or more processes (see 4.2.2). The two current working directories are also inherited.

-- PROCESS PRIORITY which determines what proportion of CPU time the process receives with respect to others (see 5.1).

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

-- A PROCESS ID: a number from 1 to 255, which is used to identify specific processes. Each process has a unique process ID number (see 4.3.2).

-- MEMORY: enough memory required for the new process to run. Level Two systems give each process a unique "address space". In Level One systems, all processes share the single address space. A "data area", used for the program's parameters, variables, and stack is allocated for the process' exclusive use. A second memory area may also be required to load the program (primary module) if it is not resident in memory (see 5.4).

To summarize, the following items are given to or associated with new processes:

-- Primary Module (program module to be run)
-- Parameter(s) passed from parent to child
-- User Number
-- Standard I/O paths and working directories
-- Process Priority
-- Process ID
-- Memory

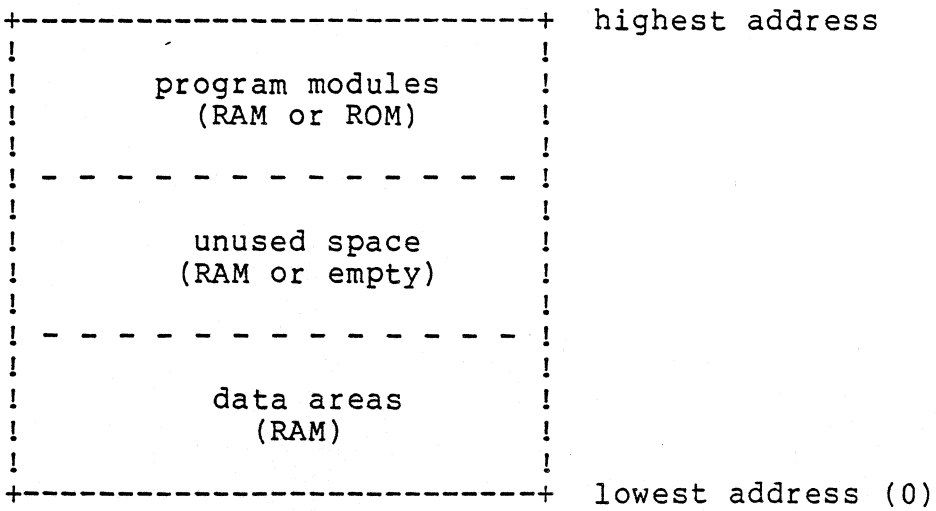
5.4 BASIC MEMORY MANAGEMENT FUNCTIONS

An important OS-9 function is memory management. OS-9 automatically allocates all system memory to itself and to processes, and also keeps track of the logical contents of memory (meaning which program modules are resident in memory at any given time).

In OS-9 Level One systems, the operating system and all processes share a single address space having up to 64K of RAM and ROM. Most Level One systems have up to 56K or 60K of RAM memory. Memory management is performed entirely by OS-9 software routines. RAM memory is assigned in 256-byte pages.

In OS-9 Level Two systems, the operating system and each process have individual address spaces. Each address space has the potential to contain up to 64K of RAM and ROM memory. Using memory management unit (MMU) hardware, OS-9 moves memory into and out of each address space from time to time, as required for various purposes. Each process is subject to a 64K maximum program size, however, a user can run several processes simultaneously and therefore utilize more than 64K overall. The maximum number of address spaces IS NOT related to the number of MMU registers, but (up to a point) more registers may improve OS-9's process switching speed. RAM memory is logically assigned in 256-byte pages, but is physically assigned in the MMU hardware's block size (usually 2K or 4K bytes). Each physical memory block has an extended address which is called a block number. For example the 4K physical block residing at addresses $3C000 - $3CFFF is called block number $3C.

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



5.4.1 LOADING PROGRAM MODULES INTO MEMORY

When performing a fork operation, OS-9's first step is to attempt to locate the requested program module by searching the "module directory", which has the address of every module present in memory. The 6809 instruction set supports a type of program called "reentrant code" which means the exact same "copy" of a program can be shared by two or more different processes simultaneously without affecting each other, provided that each "incarnation" of the program has an independent memory area for its variables.

Almost all OS-9 family software is reentrant and can make most efficient use of memory. For example, Basic09 requires 22K bytes of memory to load into. If a request to run Basic09 is made, but another user (process) had previously caused it to be loaded into memory, both processes will share the same copy, instead of causing another copy to be loaded (which would use an additional 22K of memory). OS-9 automatically keeps track of how many processes are using each program module and deletes the module (freeing its memory for other uses) when all processes using the module have terminated. OS-9 Level Two will automatically switch the same copy of a module into multiple address spaces if the program is called by more than one process.

If the requested program module is not already in memory, the name is used as a pathlist (file name) and an attempt is made to load the program from mass storage (see 3.9.4).

Every program module has a "module header" that describes the program and its memory requirements. OS-9 uses this to determine how much memory for variable storage should be allocated to the process (it can be given more memory by specifying an optional parameter on the shell command line). The module header also includes other important descriptive information about the program, and is an essential part of OS-9 operation at the machine language level. A detailed description of memory modules and module headers can be found in the "OS-9 System Programmer's Manual".

Programs can also be explicitly loaded into memory using the "load" command. As with fork, the program will actually be loaded only if it is not already in memory. If the module is not in memory, OS-9 will copy a candidate memory module from the file into memory, verify the CRC, and then, if the module is not already in the module directory, add the module to the directory. This process is repeated until all the modules in the file are loaded, the 64K memory limit is exceeded, or until a module with an invalid format is encountered. OS-9 always links to the first module read from the file.

Level One systems load modules on 256-byte page boundaries. Level Two systems load modules contiguously on memory block boundaries. The block size is usually 2K per block for systems equipped with MC6829 MMU's, or 4K bytes for most SS-50 buss systems. Free memory to be used for user data area need not be contiguous because the MMU can map scattered free blocks to be logically contiguous. Since OS-9 will request the largest physically contiguous memory block available (up to 56K) to load program modules, load operations can fail even if sufficient total free memory exists. Any of this memory not used by the load operation is returned to the system.

If the program module is already in memory, the load will proceed as described above, loading the module from the specified file, verifying the CRC, and when attempting to add the valid module to the module directory, noticing that the module is already known, the load merely increments the known module's link count (the number of processes using the module.) The load command can be used to "lock" a program into memory. This can be useful if the same program is to be used frequently because the program will be kept in memory continuously, instead of being loaded repeatedly.

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

On Level Two systems, multiple modules loaded from a single file are logically associated by the memory management logic. All modules in the group will occupy contiguous physical memory blocks. The group's memory can only be deallocated when all modules which are members of the group have zero link counts. Similarly, linking to one module within a group causes all other modules in the group to be mapped into the process's address space. (see 3.9.4).

Here is an example of the use of "load" and "unlink" to lock a program in memory. Suppose the "copy" command will be used five times. Normally, the copy command would be loaded each time the "copy" command is called. If the "load" command is used first, "copy" will be locked into memory first, for example:

```
OS9: load copy
OS9: copy file1 file1a
OS9: copy file2 file2a
OS9: copy file3 file3a
OS9: unlink copy
```

It is important to use the "unlink" command after the program is no longer needed, or the program will continue to occupy memory which otherwise could be used for other purposes. Be very careful not to completely unlink modules in use by any process! This will cause the memory used by the module to be deallocated and its contents destroyed. This will certainly cause all programs using the unlinked module to crash.

5.4.2 LOADING MULTIPLE PROGRAMS

Another important aspect of program loading is the ability to have two or more programs resident in memory at the same time. This is possible because all OS-9 program modules are "position-independent code", or "PIC". PIC programs do not have to be loaded into specific, predetermined memory addresses to work correctly, and can therefore be loaded at different memory addresses at different times. PIC programs require special types of machine language instructions which few computers have. The ability of the 6809 microprocessor to use this type of program is one of its most powerful features.

The "load" command can therefore be used two or more times (or a single file may contain several memory modules, see 3.9.4), and each program module will be automatically loaded at different, non-overlapping addresses (most other operating systems write over the previous program's memory whenever a new program is loaded). This technique also relieves the user from having to be directly concerned with absolute memory addresses. Any number of program modules can be loaded until available system memory is full.

5.4.3 MEMORY FRAGMENTATION

Even though PIC programs can be initially loaded at any address where free memory is available, program modules cannot be relocated dynamically afterwards, e.g., once a program is loaded it must remain at the address at which it was originally loaded (however Level Two systems can "load" (map) memory resident programs at different addresses in each process' address space). This characteristic can lead to a sometimes troublesome phenomenon called "memory fragmentation". When programs are loaded, they are assigned the first sufficiently large block of memory at the highest address possible in the address space. If a number of program modules are loaded, and subsequently one or more modules which are located in between other modules are "unlinked", several fragments of free memory space will exist. The sum of the sizes of the free memory space may be quite large, but because they are scattered, not enough space will exist in a single block to load a program module larger than the largest free space. This problem tends to be more severe on OS-9 Level One systems because all processes must share a single address space. Fragmentation is fairly infrequent in Level Two systems because each process has its own address space.

The "mfree" command shows the location and size of each unused memory area and the "mdir e" command shows the address, size, and link (use) count of each module in the address space. These commands can be used to detect fragmentation. Memory can usually be de-fragmented by unlinking scattered modules and reloading them. Make certain none are in use before doing so.

6.0 USE OF THE SYSTEM DISK

Disk-based OS-9 systems use a system disk to load many parts of the operating system during the system startup and to provide files frequently used during normal system operations. Therefore, the system disk is generally kept in the master disk drive ("/d0") when the system is running.

Two files used during the system startup operation, "OS9Boot" and "startup" must reside in the system disk's root directory. Other files are organized into three directories: CMDS (commands), DEFS (system-wide definitions), and SYS (other system files).

Other files and directories created by the system manager and/or users may also reside on the system disk. These frequently include each user's initial data directory.

6.1 The Bootstrap File

The file called "OS9Boot" loaded into RAM memory by the "bootstrap" routine located in the OS-9 firmware. It includes file managers, device drivers and descriptors, and any other modules which are permanently resident in memory. A typical Microware OS-9 distribution disk's "OS9Boot" file contains the following modules:

OS9p2	OS-9 Level Two Kernel (Level Two only)
INIT	System Initialization Table (Level Two only)
IOMAN	OS-9 Input/Output Manager
RBF	Random Block (disk) File Manager
SCF	Sequential Character (terminal) File Manager
Pipeman	Pipeline File Manager
Piper	Pipeline Driver
Pipe	Pipeline Device Descriptor
ACIA	Terminal Device Driver (MC6850)
PIA	Printer Device Driver (MC6821)
DISK	Disk Driver
D0, D1, etc.	Disk Device Descriptor
TERM	Terminal Device Descriptor
T1, T2, etc.	Other Terminal Device Descriptors
P	Printer (parallel) Device Descriptor
P1	Printer (serial) Device Descriptor
CLOCK	Real-Time Clock Module
SYSGO	System Startup Process

Users may create new bootstrap files which may include additional modules (see "OS9Gen command). Any module loaded as part of the bootstrap cannot be unlinked and is stored in memory with a minimum of fragmentation. In Level One, it is advantageous to include in

the OS9Boot file any module used constantly during normal system operation. In Level Two, however, since files placed in the OS9boot file will be loaded into the same memory block, when the system switches the boot block into its own address space, the non-system files decrease the amount of memory addressable in system mode. Alternatively, optional modules should be placed in a separate file that is the loaded as part of the system startup procedure.

6.2 The SYS Directory

The directory "/d0/SYS" contains three important files:

password - the system password file (see "login" command)
motd - message of the day file, displayed during login
errmsg - the error message file (see 4.7)

These files (and the SYS directory itself) are not absolutely required to boot OS-9, they are needed if "login", "tsmon", or "printerr" will be used. Users may add other system-wide files of similar nature if desired.

6.3 The Startup File

The file "/d0/startup" is a shell procedure file (see 4.6) which is automatically processed immediately after system startup. The user may include in "startup" any legal shell command line. Often this will include "setime" to start the system clock. If this file is not found during startup, the system will still start up, running the shell on the console terminal.

6.4 The CMDS Directory

The directory "/d0/CMDS" is the system-wide command object code directory, which is normally shared by all users as their working execution directory (see 3.7). If "shell" is not part of the "OS9Boot" file, it must be present in this directory. The system startup process "sysgo" sets CMDS to be the initial execution directory.

6.5 The DEFS Directory

The directory "/d0/DEFS" is a directory that contains assembly language source code files which contain common system-wide symbolic definitions, and are normally included in assembly language programs by means of the OS-9 Assembler "use" directive. The presence and use of this directory is optional, but highly recommended for any

system used for assembly language software development.

The files commonly contained in this directory are:

```
OS9Defs.ed -  main system-wide definition file
OS9SysDefs -  OS-9 internal system definition file
OS9IODefs  -  I/O Manager definition file
OS9RbfDefs.ed -  RBF file manager definition file
OS9ScfDefs.ed -  SCF file manager definition file
Sysdefs.ed    -  System configuration definition file
Systype       -  System types definition file
li.equates    -  Equates for old Level One (V1.1) names
```

The extension ".ed" is a qualifier for various editions of the definition files.

6.6 Changing System Disks

The system disk is not usually removed while the system is running, especially on multiuser systems. If it is, the "chx" and "chd" (if the working data directory was on the system disk) commands should be executed to reset the working directory pointers because the directories may be at different addresses on the new disk, for example:

```
chx /d0/cmds
chd /d0
```

In general, it is unwise to remove a disk and replace it with another if any paths are open to files resident on the disk. It is dangerous to exchange any disk if any files on it are open in WRITE or UPDATE modes.

6.7 Making New System Disks

To make a system disk, the following steps must be performed:

1. The new disk must be formatted.
2. The "OS9Boot" file must be created and linked by the "OS9Gen" or "Cobbler" (Level 1 only) commands.
3. The "startup" file must be created or copied.
4. The CMDS and SYS directories and the files they contain must be copied.

Steps 2 through 4 may be performed manually, or automatically by any of the following methods:

1. By a shell procedure file created by the user.
2. By a shell procedure file generated by the "dsave" command (see 1.1.3).
3. By the "backup" command, if the new disk has the same physical size and format as the original (see 1.1.2).

7.0 SYSTEM COMMAND DESCRIPTIONS

This section contains descriptions for each of the command programs that are supplied with OS-9. These programs are usually called using the shell, but can be called from most other OS-9 family programs such as BASIC09, Interactive Debugger, Macro Text Editor, etc. Unless otherwise noted, these programs are designed to run as individual processes.

WARNING - ALTHOUGH MANY OS-9 COMMANDS MAY WORK ON LEVEL ONE OR LEVEL TWO SYSTEMS, THERE ARE DIFFERENCES. TAKE CARE NOT TO MIX COMMAND FILES FROM LEVEL ONE SYSTEMS ON LEVEL TWO, OR THE REVERSE.

7.1 FORMAL SYNTAX NOTATION

Each command description includes a syntax definition which describes how the command sentence can be constructed. These are symbolic descriptions that use the following notation:

[ ]	= Brackets indicate that the enclosed item(s) are optional.
{ }	= Braces indicate that the enclosed item(s) can be either omitted or repeated multiple times.
<path>	= Represents any legal pathlist.
<devname>	= Represents any legal device name.
<modname>	= Represents any legal memory module name.
<procID>	= Represents a process number.
<opts>	= One or more options defined in the command description.
<arglist>	= a list of arguments (parameters).
<text>	= a character string terminated by end-of-line.

NOTE: The syntax of the commands given does not include the shell's built in options such as alternate memory size, I/O redirection, etc. This is because the shell will filter its options out of the command line before it is passed to the program being called.

ATTR

Change file security attributes

Syntax: ATTR <path> [{ <permission abbreviations> }]

This command is used to examine or change the security permissions of a file. To enter the command, type "ATTR" followed by the pathlist for the file who's security permissions are to be changed, followed by a list of permissions which are to be turned on or off. A permission is turned on by giving its abbreviation, or turned off by preceding its abbreviation with a minus sign. Permissions not explicitly named are not affected. If no permissions are given the current file attributes will be printed. You can not change 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 = Sharable file
 r = Read permit to owner
 w = Write permit to owner
 e = Execute permit to owner
pr = Read permit to public
pw = Write permit to public
pe = Execute permit to public
```

The ATTR command may be used to change a directory file to a non-directory file if all entries have been deleted from it. Since the DEL command will only delete non-directory files, this is the only way a directory may be deleted. You cannot change a non-directory file to a directory file with this command (see MAKDIR).

For more information see: 3.8, 3.8.1

Examples:

```
attr myfile -pr -pw

attr myfile r w e pr rw pe


attr datalog
-s-wr-wr
```

BACKUP

Make a backup copy of a disk

Syntax: BACKUP [e] [s] [-v] [<devnam> [<devname>]]

This command is used to physically copy all data from one device to another. 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.

If both device name 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.

The options are:

```
   E = Exit if any read error occurs.
   S = Print single drive prompt message.
  -V = Do not verify.
 #nK = more memory makes backup run faster
```

Examples:

```
backup /D2 /D3

backup -V



OS9: backup

Ready to BACKUP from /D0 to /D1 ?: Y
MYDISK is being scratched
OK ?: Y
Number of sectors copied: $04D0
Verify pass
Number of sectors verified: $04D0
OS9:
```

BACKUP (continued)

Below is an example of a single drive backup. BACKUP will read a portion of the source disk into memory, you remove the source disk and place the destination disk into the drive, BACKUP writes on the destination disk, you remove the destination disk and place the source disk into the drive. This continues until the entire disk has been copied. Giving BACKUP as much memory as possible will cause fewer disk exchanges to be required.

For more information see: 1.1.2

```
OS9:backup /D0 #10k

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

(several repetitions)

Ready DESTINATION, hit a key:
Number of sectors copied: $4D0
Verify pass
Number of sectors verified: $4D0
```

BINEX
EXBIN

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

Syntax: BINEX <path1> <path2>
EXBIN <path2> <path1>

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

BINEX converts "path1", an OS-9 binary format file, to a new file named "path2" in S-Record format. If invoked on a non-binary load module file, a warning message is printed and the user is asked if BINEX should proceed anyway. A "Y" response means yes; any other answer will terminate the program. S-Records have a header record to store the program name for informational purposes and each data record has an absolute memory address which is not meaningful to OS-9 since it uses position-independent-code. However, the S-Record format requires them so BINEX will prompt the user for a program name and starting load address. For example:

```
binex /d0/cmds/scanner scanner.S1
Enter starting address for file: $100
Enter name for header record: scanner
```

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

```
list scanner.S1 >/T1
```

EXBIN is the inverse operation; "path1" is assumed to be a S-Record format text file which EXBIN converts to pure binary form on a new file called "path2". The load addresses of each data record must describe continguous 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 or VERIFY commands can be used to check the validity of the modules if they are to be loaded or run. Example:

```
exbin program.S1 cmds/program
```

BUILD

Build a text file from standard input

Syntax: BUILD <path>

This command is used to build short text files by copying the standard input path into the file specified by <path>. BUILD creates a file according to the pathlist parameter, them displays a "?" prompt to request an input line. Each line entered is written to the output path (file). Entering a line consisting of a carriage return only causes BUILD to terminate.

Example:

```
build small_file
build /p                    (copies keyboard to printer)
```

The standard input path may also be redirected to a file. Below is an example:

```
build <mytext /T2           (copies file "mytext" to terminal T2)
```

```
OS9: build newfile

? The powers of the OS-9
? operating system are truly
? fantastic.
? [RETURN]

OS9: list newfile

The powers of the OS-9
operating system are truly
fantastic.
```

CHD
Change working data directory

CHX
Change working execution directory

Syntax: chd <pathlist>
chx <pathlist>

These are shell "built in" commands used to change OS-9's working data directory or working execution directory. Many commands in OS-9 work with user data such as text files, programs, etc. These commands assume that a file is located in the working data directory. Other OS-9 commands will assume that a file is in the working execution directory.

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

For more information see: 3.7, 3.7.2

Examples:

```
chd /d1/PROGRAMS

chx ..

chx binary_files/test_programs

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

CMP

File Comparison Utility

Syntax: CMP <file1> <file2>

Opens two files and performs a comparison of the binary values of the corresponding data bytes of the files. If any differences are encountered, the file offset (address) and the values of the bytes from each file are displayed in hexadecimal.

The comparison ends when 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.

Example:

```
OS9: cmp  red  blue

 Differences

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

Bytes compared:   0000002D
Bytes different:  00000005

OS9: cmp  red  red

 Differences

   None ...

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

COBBLER

Make a bootstrap file
(Level One only)

Syntax: COBBLER <device name>

COBBLER is used to create the "OS9Boot" file required on any disk from which OS-9 is to be bootstrapped. The boot file will consist of the same modules which were loaded into memory during the most recent bootstrap. To add modules to the bootstrap file use the "OS9Gen" command. Level Two systems must use "OS9Gen" to create bootstrap files.

NOTE: The boot file must fit into one contiguous block on the mass-storage device. For this reason COBBLER is normally used on a freshly formatted disk. If COBBLER is used on a disk and there is not a contiguous block of storage large enough to hold the boot file, the old boot file may have been destroyed and OS-9 will not be able to boot from that disk until it is reformatted.

NOTE: COBBLER cannot be used with Exordisk controllers at other than 1 Mhz.

For more information see: 1.1.2, 6.1

Examples:

```
OS9: cobbler /D1
```

COPY

Copy data from one path to another

Syntax: COPY <path> <path> [-s]

This command copies data from the first file or device specified to the second. The first file or device must already exist, the second file is automatically created if the second path 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.

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.

The "-s" option causes COPY to perform a single drive copy operation. The second pathlist must be a full pathlist if "-s" appears. COPY will read a portion of the source disk into memory, you remove the source disk and place the destination disk into the drive, enter a "C" whereupon COPY writes on the destination disk, this process continues until the entire file is copied.

Using the shell's alternate memory size modifier to give a large memory space will increase speed and reduce the number of media exchanges required for single drive copies.

Examples:

```
copy file1 file2 #15k              (copies file1 to file2)

copy /D1/joe/news /D0/peter/messages

copy /TERM /P                      (copies console to printer)

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

DATE

Display system date and time

Syntax: DATE [t]

This command will display the current system date, and if the "t" option is given, the current system time.

Examples:

```
 date t

 date t >/p                    (Output is redirected to printer)

  OS9:setime

         YY/MM/DD HH:MM:SS
  TIME ? 81/04/15 14:19:00

  OS9:date

  April 15, 1981

  OS9:date t

  April 15, 1981  14:20:20
```

DCHECK

Check Disk File Structure

Syntax: DCHECK [-opts] <devnam>

It is possible for sectors on a disk to be marked as being allocated but in fact are not actually 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 (see 3.5). DCHECK is a diagnostic that can be used to detect this condition, as well as the general integrity of the directory/file linkages.

DCHECK is given as a parameter the name of the disk device to be checked. After verifying and printing some vital file structure parameters, DCHECK follows pointers down the disk's file system tree to all directories and files on the disk. As it does so, it verifies the integrity of the file descriptor sectors, reports any discrepancies in the directory/file linkages, and builds a sector allocation map from the segment list associated with each file. If any file descriptor sectors (FDs) describe a segment with a cluster not within the file structure of the disk, a message is reported like:

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

This indicates that a segment starting at sector xxxxxx and ending at sector yyyyyy cannot really be on this disk. Because there is a good chance the entire FD is bad if any of it's segment descriptors are bad, the allocation map is <u>not</u> updated for corrupt FDs.

While building the allocation map, DCHECK also makes sure that each disk cluster appears only once and only once in the file structure. If this condition is detected, DCHECK will display a message like:

Cluster $xxxxxx was previously allocated

This message indicates that cluster 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.

DCHECK (continued)

The newly created allocation map is then compared to the allocation map stored on the disk, and any differences are reported in messages like:

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

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

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

Available DCHECK options are:

-w=<path>	pathlist to directory for work files
-p	print pathlists for questionable clusters
-m	save allocation map work files
-b	suppress listing of unused clusters
-s	display count of files and directories only
-o	print DCHECK's valid options

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

DCHECK builds its disk allocation map in a file called <pathlist>/DCHECKpp0, where <pathlist> is as specified by the "-w=" option and pp is the process number in hexadecimal. Each bit in this bitmap file corresponds to a cluster of sectors on the disk. If the "-p" option appears on the command line, DCHECK creates a second bitmap file (<pathlist>/DCHECKpp1) that has a bit set for each cluster DCHECK finds as "previously allocated" or "in file structure but not in allocation map" while building the allocation map. DCHECK then makes another pass through the directory structure to determine the pathlists for these questionable clusters. These bitmap work files may be saved by specifying the "-m" option on the command line.

Restrictions:

For best results, DCHECK should have exclusive access to the disk being checked. Otherwise DCHECK may be fooled if the disk allocation map changes while it is building its bitmap file from the changing file structure. DCHECK cannot process disks with a directory depth greater than 39 levels.

For more information see: 3.10, 3.5, FORMAT
6.1 of OS-9 Systems Programmer's Manual

Examples:

```
OS9: dcheck /d2   (workfile is on /D0)

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, allocation map and root directory
Building allocation map work file...
Checking allocation map file...


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

DCHECK (continued)

```
OS9: dcheck -mpw=/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, allocation map and root directory
Building allocation map work file...
Cluster $00040 was previously allocated
*** Bad FD segment ($111111-$23A6F0) for file: /d0/test/junky.file
Checking allocation map file...
Cluster $000038 in file structure but not in allocation map
Cluster $00003B in file structure but not in allocation map
Cluster $0001B9 in allocation map but not in file structure
Cluster $0001BB in allocation map but not in file structure

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

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

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

DEL

Delete a file

Syntax: DEL [-x] <path> {<path>} [-x]

This command is used to delete the file(s) specified by the pathlist(s). The user must have write permission for the file(s). Directory files cannot be deleted unless their type is changed to non-directory: see the "ATTR" command description.

If the -x option appears, the current execution directory is assumed.

For more information see: 3.5, 3.8.1

Examples:

```
del test_program old_test_program

del /D1/number_five

 OS9:dir /D1

    Directory of /D1  14:29:46
myfile          newfile


 OS9:del /D1/newfile
 OS9:dir /D1

    Directory of /D1  14:30:37
myfile

 OS9:del myprog -x
 OS9:del -x CMDS.SUBDIR/file
```

DELDIR

Delete All Files In a Directory System

Syntax: DELDIR <directory name>

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

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

```
OS9: deldir OLDFILES
Deleting directory file.
List directory, delete directory, or quit ? (l/d/q)
```

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.

A "d" response will initiate the process of deleting files.

A "q" response will abort the command before action is taken.

The directory to be deleted may include directory files, which may themselves include directory files, etc. In this case, DELDIR operates recursively (e.g., it calls itself) so all lower-level directories are deleted as well. In this case 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 both must reside in the current execution directory.

DIR

Display the names of files contained in a directory

Syntax: DIR [e] [x] [<path>]

Displays a formatted list of files names in a directory file on the standard output path. 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, it 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.

For more information see: 1.0.3, 3.4, and 3.8.1

Examples:

dir	(display data directory)
dir x	(display execution directory)
dir x e	(display entire description of execution dir)
dir ..	(display parent of working data directory)
dir newstuff	(display newstuff directory)
dir e test_programs	(display entire description of "test_programs)

DISPLAY

Display Converted Characters

Syntax: DISPLAY <hex> {<hex>}

Display reads one or more hexadecimal numbers given as parameters, converts them to ASCII characters, and writes them to the standard output. It is commonly used to send special characters (such as cursor and screen control codes) to terminals and other I/O devices.

Examples:

```
display 0C 1F 02 7F

display 15 >/p          (sends "form feed" to printer)


OS9: display 41 42 43 44 45 46
ABCDEF
```

DSAVE - Generate procedure file to copy files

Syntax: dsave [-opts] [<devname>] [<path>]

Dsave is used to backup or copy all files in one or more directories. It is unlike most other commands in that it does NOT directly affect the system, rather, it generates a procedure file which is executed later to actually do the work.

When DSAVE is executed, it writes copy commands to standard output to copy files from the current data directory on <devname> (the default is /D0) to the directory specified by <path>. If <path> does not appear, the copy is performed to the current data directory at the time the DSAVE procedure file is executed. 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. Since DSAVE is recursive in operation, the procedure file will exactly replicate all levels of the file system from the current data directory downward (such a section of the file system is called a "subtree").

If the current working directory happens to be the root directory of the disk, DSAVE 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 floppy disk to a hard disk.

Available DSAVE options are:

-b	make output disk a system disk by using source disk's "OS9Boot" file, if present.
-b=<path>	make output disk a system disk using <path> as source for the "OS9Boot" file.
-i	indent for directory levels
-L	do not process directories below the current level
-m	do not include "makdir" commands in procedure file
-s<integer>	set copy size parameter to <integer> K

For more information see: 1.1.3

Example which copies all files on "d2" to "d1":

```
chd /d2                              (select "from" directory)
dsave /d2 >/d0/makecopy              (make procedure file "makecopy")
chd /d1                              (select "to" directory)
/d0/makcopy                          (run procedure file)

chd /d0/MYFILES/STUFF
dsave -is32 /d0 /d1/BACKUP/STUFF >saver
/d0/MYFILES/STUFF/saver
```

DUMP

Formatted File Data Dump in Hexadecimal and ASCII

Syntax: DUMP [<path>]

This command produces a formatted display of the physical data contents of the path specified which may be a mass storage file or any other I/O device. If a pathlist is omitted, the standard input path is used. The output is written to standard output. This command is commonly used to examine the contents of non-text files.

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 on files exactly as they exist in memory, the addresses shown on the dump correspond to the relative load addresses of memory-module files.

Examples:

```
DUMP                    (display keyboard input in hex)
DUMP myfile >/P         (dump myfile to printer)
DUMP shortfile
```

Sample Output:

```
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.
```

starting address — data bytes in hexadecimal format — data bytes in ASCII format

ECHO

Echo text to output path

Syntax: ECHO <text>

This command 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. The text should not include any of the punctuation characters used by the shell.

Examples:

```
echo >/T2 Hello John how's it going &    (echo to T2)

echo >/term ** warning ** disk about to be scratched!

echo >/p Listing of Transaction File;  list trans >/p


OS9: echo Here is an important message!
Here is an important message!
```

EX

Execute program as overlay

Syntax: EX <module name> [<modifiers>] [<parameters>]

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

This command 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 only use BASIC09 need not waste memory space on SHELL.

The "ex" command should always be the last command on a shell input line because any command line following will never be processed.

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

For more information see: 4.5, 4.6, 4.9

Examples:

```
ex BASIC09

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

FORMAT

Initialize disk media

Syntax: FORMAT <devname> [<option list>]

This command 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.

This command can be used format almost any type of disk including hard disks. A description of the disk is automatically read from the device descriptor module. The <option list> may be used to override these default values if necessary. Format will ask for any required options not given in the command line.

NOTE: If the diskette is to be used as a system disk, "OS9gen" or "cobbler" must be run to create the bootstrap after the disk has been formatted.

When format is used with floppy disks, these options are used:

S = single density (default)
D = double density
1 = single sided (default)
2 = double sided

When used with floppy or hard disks, these options are used:

R = inhibit ready prompt
'number' = number of tracks (in decimal)
:number: = number of sector interleave value (decimal)
"name" = disk name (32 character maximum)

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 cannot be defective.

For more information see: 1.1.1, 3.10

FORMAT (continued)

Examples:

```
format /D1  2  D "database"  '77'

format /D1  S 1
```

```
OS9: format /D1

                    FORMAT 1.1

          TABLE OF FORMAT VARIABLES

              Recording Format:   MFM  <- density: FM=sgl MFM=dbl
          Track density in TPI:    48  <- some 5" disks use 96 TPI
           Number of Cylinders:    77  <- 77 for 8", 35/40/80 for 5"
           Number of Surfaces:      1  <- 1 or 2 sides
        Sector Interleave Offset:   3  <- set by manufacturer


                       Disk type:    8  <- 5, 8, or HARD
Sectors/Track on TRK 0, Side 0:     16  <- set by manufacturer
                  Sectors/Track:    28  <- set by manufacturer

Formatting on drive /D1
y (yes), n (no), or q (quit)            <- answer: y to format, n to
Ready: Y                                   change table, or q to stop
Disk Name: Jim's Disk                   <- enter up to 32 characters


  (track numbers displayed here)

  GOOD SECTOR COUNT = $0860
```

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

FREE

Display free space remaining on mass-storage device

Syntax: FREE <devname>

This command displays the number of unused 256-byte sectors on a device which are available for new files or for expanding existing files. The device name given must be that of a mass-storage multifile device. "Free" also displays the disk's name, creation date, and cluster size.

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 may not be divisible into as many files. For example, if a given disk system uses 8 sectors per cluster, and a "free" command shows 32 sectors free, a maximum of four new files could be created even if each has only one cluster.

For more information see: 3.10

Examples:

```
OS9: free
BACKUP DATA DISK created on: 80/06/12
Capacity: 1,232 sectors (1-sector clusters)
1,020 free sectors, largest block 935 sectors

OS9: 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
```

IDENT

Print OS-9 module identification

Syntax: IDENT [-opts] <path> [-opts]

This command is used to display header information from OS-9 memory modules. IDENT displays the module size, CRC bytes (with verification), and for program and device driver modules, the execution offset and the permanent storage requirement bytes. IDENT will print and interpret the type/language and attribute/revision bytes. In addition, IDENT displays the byte immediately following the module name since most Microware-supplied modules set this byte to indicate the module edition.

IDENT will display all modules contained in a disk file. If the "-m" option appears, <path> is assumed to be a module in memory.

If the "-v" option is specified, the module CRC is not verified.

The "-x" option implies the pathlist begins in the execution directory.

The "-s" option causes IDENT to display the following module information on a single line:

Edition byte (first byte after module name)
Type/Language byte
Module CRC
A "." if the CRC verifies correctly, "?" if incorrect.
(IDENT will leave this field blank if the "-v" option appears.)
Module name

Examples:

```
OS9: ident -m ident
Header for:  Ident                  <Module name>
Module size: $06A5    #1701         <Module size>
Module CRC:  $1CE78A (Good)         <Good or Bad>
Hdr parity:  $8B                    <Header parity>
Exec. off:   $0222    #546          <Execution offset>
Data size:   $0CA1    #3233         <Permanent storage requirement>
Edition:     $05      #5            <First byte after module name>
Ty/La At/Rv: $11 $81                <Type/Language Attribute/Revision>
Prog mod, 6809 obj, re-en           <Module type, Language, Attribute>
```

IDENT (continued)

```
OS9: ident /d0/os9boot -s
    1 $C0 $A366DC . OS9p2
   83 $C0 $7FC336 . Init
    1 $11 $39BA94 . SysGo
    1 $C1 $402573 . IOMan
    3 $D1 $EE937A . RBF
   82 $F1 $526268 . D0
   82 $F1 $D65245 . D1
   82 $F1 $E32FFE . D2
    1 $D1 $F944D7 . SCF
    2 $E1 $F9FE37 . ACIA
   83 $F1 $765270 . TERM
   83 $F1 $B4396C . T1
   83 $F1 $63B73B . T2
   83 $F1 $0F9B78 . T3
   83 $F1 $F83EB9 . T4
   83 $F1 $D6DD9A . T5
    3 $E1 $3EE015 . PIA
   83 $F1 $12A43B . P
    2 $D1 $BBC1EE . PipeMan
    2 $E1 $5B2B56 . Piper
   80 $F1 $CC06AF . Pipe
    2 $C1 $248B2C . Clock
    ^  ^     ^    ^ ^
    |  |     |    | |
    |  |     |    | Module name
    |  |     |    CRC check " " if -v, "." if OK, "?" if bad
    |  |    CRC value
    |  Type/Language byte
    Edition byte (first byte after name)
```

KILL

Abort a process

Syntax: KILL <procID>

This shell "built in" command sends an "abort" signal to the process having the process ID number specified. The process to be aborted must have the same user ID as the user that executed 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. Therfore, if you KILL a process and the PROCS command shows it still exists, it is probably waiting for receive a line of data from a terminal before it can die.

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

For more information see: 4.5, 5.2, PROCS

Examples:

```
kill 5

kill 22
```

```
OS9: procs

Usr #  Id pty  state    Mem Primary module
----- --- --- -------- --- --------------
   20   2   0  active     2 Shell <TERM
   20   1   0  waiting    1 Sysgo <TERM
   20   3   0 sleeping   20 Copy <TERM

OS9: kill 3
OS9: procs

Usr #  Id pty  state    Mem Primary module
----- --- --- -------- --- --------------
   20   2   0  active     2 Shell <TERM
   20   1   0  waiting    1 Sysgo <TERM

OS9:
```

LINK

Link module into memory

Syntax: LINK <memory module name>

This command is used to "lock" a previously loaded module into memory. On Level Two systems, "link" causes a module resident in system memory to be mapped into the user's address space. 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.

For more information see: 5.4, 5.4.1, 5.4.2, 5.4.3

Examples:

OS9: LINK edit

OS9: LINK myprogram

LIST

List the contents of a text file

Syntax: LIST <path> { <path> }

This command copies text lines from the path(s) given as parameters to the standard output path. 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.

This command is most commonly used to examine or print text files.

For more information see: 2.3, 3.9.2

Examples:

```
list /d0/startup >/P &          (output is redirected to printer)

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

list /TERM >/p                  (copy keyboard to printer - use
                                "escape" key to terminate input)
```

```
OS9: build animals
? cat
? cow
? dog
? elephant
? bird
? fish
? [RETURN]

OS9: list animals
cat
cow
dog
elephant
bird
fish
```

LOAD

Load module(s) from file into memory

Syntax: LOAD <path>

The path specified is opened and one or more modules is read from it and loaded into memory. The names of the modules are added to the module directory. If a module is loaded that has the same name and type as a module already in memory, the module having the highest revision level is kept.

For more information see: 3.9.4, 5.4.1, 5.4.2

Example:

```
load new_program
```

```
OS9:mdir

   Module Directory at 13:36:47
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         CDS         H1
Sysgo       Clock       Shell       Tsmon       Copy
Mdir

OS9:load edit
OS9:mdir

   Module Directory at 13:37:14
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         CDS         H1
Sysgo       Clock       Shell       Tsmon       Copy
Mdir        EDIT
```

LOGIN

Timesharing System Log-In

Syntax: LOGIN

Login is used in timesharing systems to provide log-in security. It is automatically called by the timesharing monitor "tsmon", or can be used after initial log-in to change a terminal's user.

Login requests a user name and password, which is checked against a validation file. If the information is correct, the user's system priority, user ID, and working directories are set up according to information stored in the file, and the initial program specified in the password file is executed (usually SHELL). If the user cannot supply a correct user name and password after three attempts, the process is aborted. The validation file is called "PASSWORD" and 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:

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

2. Password (up to 32 characters, may include spaces) If this field is omitted, no password is required by the specific use.

3 User index (ID) number (from 0 to 65535, 0 is superuser). This number is used by the file security system and as the system-wide user ID to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user. (See 3.8)

4. Initial process (CPU time) priority: 1 - 255 (see 5.2)

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

6. Pathlist of initial data directory (specific user's directory)

7. Name of initial program to execute (usually "shell").
   NOTE: This is not a shell command line.

Here's a sample validation file:

```
superuser,secret,0,255,.,.,shell
steve,open sesame,3,128,.,/d1/STEVE,shell
sally,qwerty,10,100,/d0/BUSINESS,/d1/LETTERS,wordprocessor
bob,,4,128,.,/d1/BOB,Basic09
```

LOGIN (continued)

To use the login command, enter:

```
login
```

This will cause prompts for the user's name and (optionally) password to be displayed, and if answered correctly, the user is logged into the system. Login initializes the user number, working execution directory, working data directory, and executes the initial program specified by the password file. The date, time and process number (which is not the same as the user ID, see 5.3) are also displayed.

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. For example:

```
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, a file named "motd" appearing in the SYS directory will cause LOGIN to display it's contents on the user's terminal after successful login. This file is not required for LOGIN to operate.

For more information see: tsmon, 2.5, 3.8, 5.3

Example:

```
OS9: login

OS-9 Level 1 Timesharing System  Version 1.2  82/12/04 13:02:22

User name?: superuser
Password: secret

Process #07 logged   81/12/04 13:03:00
Welcome!
```

MAKDIR

Create directory file

Syntax: MAKDIR <path>

Creates a new directory file according to the pathlist given. The pathlist must refer to a parent directory for which the user has write permission.

The new directory is initialized and initially does not contain files except for the "." and ".." pointers to its parent directory and itself, respectively (see 3.7.3). All access permissions are enabled (except sharable).

It is customary (but not mandatory) to capitalize directory names.

For more information see: 3.3, 3.4, 3.5, 3.7.3, 3.9.5

Examples:

```
makdir /d1/STEVE/PROJECT

makdir DATAFILES

makdir ../SAVEFILES
```

MDIR

Display Module Directory

Syntax: MDIR [e]

Displays the present module names in the system module directory, i.e., all modules currently resident in memory.

If the "e" option is given, a full listing of the physical address, size, type, revision level, and user count of each module is displayed. All numbers shown are in hexadecimal.

On Level Two systems, the extended physical address (block number and offset within the block) are given.

WARNING: not all modules listed by MDIR are executable as processes: always check the module type code before executing a module if you are not familiar with it to make sure it is executable.

For more information see: 5.4.1

Example:

```
OS9: mdir

   Module Directory at 14:44:35
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         CDS         HI
Sysgo       Clock       Shell       Tsmon       Mdir
```

MERGE

Copy and Combine Files to Standard Output

Syntax: MERGE <path> { <path> }

This command copies multiple input files specified by the pathlists given as parameters to the standard output path. It is commonly used to combine several files into a single output file. 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.

Examples:

```
OS9: merge file1 file2 file3 file4 >combined.file

OS9: merge compile.list asm.list >/printer
```

MFREE

Display Free System RAM

Syntax: MFREE

Displays a list of which areas of memory are not presently in use and available for assignment. The address and size of each free memory block are displayed.

In Level One systems, mfree shows the address and size of each contiguous area of unassigned RAM. The size is given as the number of 256-byte pages. This information is useful to detect and correct memory fragmentation (see 5.4.3).

In Level Two systems, mfree shows the block number, physical (extended) beginning and ending addresses, and size of each contiguous area of unassigned RAM. The size is given in number of blocks and in K bytes. The block size is usually 2K per block for systems equipped with MC6829 MMUs, or 4K bytes for most SS-50 buss systems. Free memory to be used for user data area need not be contiguous because the MMU can map scattered free blocks to be logically contiguous. Since OS-9 requires 56K of physically contiguous memory to load program modules, load operations can fail even if sufficient total free memory exists.

For more information see: 5.4, 5.4.3

Example (Level One MFREE):

```
OS9: mfree

 Address  pages
--------- -----
 700- 7FF     1
 B00-AEFF   164
B100-B1FF     1

Total pages free = 166
```

Example (Level Two MFREE):

```
  Blk Begin  End   Blks  Size
  --- ----- ----- ---- ------
   10 10000 10FFF    1     4K
   18 18000 1DFFF    6    24k
   20 20000 3FFFF   32   128k
                    ==== ======
            Total:  39   156k
```

OS9Gen

Build and Link a Bootstrap File

Syntax: OS9GEN <device name>

OS9Gen is used to create and link the "OS9Boot" file 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 (such as for a different system).

On OS-9 Level One systems, the "cobbler" command is usually a convenient way to make an exact copy of the existing boot file. On Level Two systems, OS9Gen is the only way a boot file can be made.

The name of the device on which the "OS9Boot" file is to be installed is passed to OS9Gen as a command line parameter. OS9Gen then creates a working file called "TempBoot" on the device specified. Next it reads file names (pathlists) from its standard input, one pathlist per line. Every file named is opened and copied to "TempBoot". This is repeated until end-of-file or a blank line is reached on OS9Gen's standard input. All boot files must contain the OS-9 component modules listed in section 6.1.

After all input files have been copied to "TempBoot", the old "OS9Boot" file, if present, is deleted. "TempBoot" is then renamed to "OS9Boot", and its starting address and size is 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 print a warning message indicated the disk cannot be used to bootstrap OS-9.

The list of file names given to OS9Gen can be entered from a keyboard, or OS9Gen's standard input may be redirected to a text file containing a list of file names (pathlists). If 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.

For more information see: 6.0, 6.1, 6.6

Examples are given on the following page.

OS9Gen - Continued

Examples:

To manually install a boot file on device "d1" which is an exact copy of the "OS9Boot" file on device "d0":

```
OS9: os9gen /d1              (run OS9Gen)
/d0/os9boot                  (enter file to be installed)
[ESCAPE]                     (enter end-of-file)
```

To manually install a boot file on device "d1" which is a copy of the "OS9Boot" file on device "d0" with the addition of modules stored in the files "/d0/tape.driver" and "/d2/video.driver":

```
OS9: os9gen /d1              (run OS9Gen)
/d0/os9boot                  (enter main boot file name)
/d0/tape.driver              (enter name of first file to be added)
/d2/video.driver             (enter name of second file to be added)
[ESCAPE]                     (enter end-of-file)
```

As above, but automatically by redirecting OS9Gen standard input:

```
OS9: build /d0/bootlist      (use "build" to create file "bootlist")
? /d0/os9boot                (enter first file name)
? /d0/tape.driver            (enter second file name)
? /d2/video.driver           (enter third file name)
? [RETURN]                   (terminate "build")
OS9: os9gen /d1 </d0/bootlist   (run OS9gen with redirected input)
```

PRINTERR

Print Full Text Error Messages

Syntax: PRINTERR

This command replaces the basic OS-9 error printing routine (F$PERR service request) which only prints error code numbers, with a routine the reads and displays textual error messages from the file "/d0/SYS/errmsg". Printerr's effect is system-wide.

A standard error message file is supplied with OS-9. This file can be edited or replaced by the system manager. The file is a normal text file with variable length line. Each error message line begins with the error number code (in ASCII characters), a delimiter, and the error message text. The error messages need not be in any particular order. Delimiters are spaces or any character numerically lower then $20. Any line having a delimiter as its first character is considered a continuation of the previous line(s) which permits multi-line error messages.

WARNING: Once the printerr command has been used, it can not be undone. Once installed, the PRINTERR module should not be unlinked. PRINTERR uses the current user's stack for an I/O buffer, so users are encouraged to reserve reasonably large stacks.

For more information see: 4.7, 6.2

Example:

```
OS9: printerr
```

PROCS

Display Processes

Syntax: PROCS [e]

Displays a list of processes running on the system. Normally only processes having the user's ID are listed, but if the "e" option is given, processes of all users are listed. The display is a "snapshot" taken at the instant the command is executed: processes can switch states rapidly, usually many times per second.

On Level One systems, PROCS shows the user and process ID numbers, priority, state (process status), memory size (in 256 byte pages), primary program module, and standard input path.

On Level Two Systems, the process ID number, the parent process ID number User Index (ID), process priority, memory size (in 256 byte pages), current stack pointer address, and primary module (name of program being executed) are listed.

For more information see: 5.1, 5.2, 5.3

Level One Example:

```
Usr #  Id pty  state    Mem Primary module
----- --- --- -------- --- --------------
    0   2   0  active     2 Shell <TERM
    0   1   0  waiting    1 SysGo <TERM
    1   3   1  waiting    2 Tsmon </T1
    1   4   1  waiting    4 Shell </t1
    1   5   1  active    64 Basic09 </t1
```

Level Two Example:

```
   Parnt User     Mem Stack
ID  ID  Index Pty Siz  Ptr   Primary Module
--- --- ----- --- --- ----- ----------------
  2   1     0 255   1 $98E2 SysGo
  3   2     0 255   2 $96E2 Shell
  4   3     0 255  96 $94E2 Basic09
  5   4     0 255   2 $92E2 Shell
  6   5     0 255   4 $03F3 Procs
  7   2     0 128  48 $A0F0 Cobol
```

PWD
PXD
Print Working Directory
Print Execution Directory

Syntax: PWD
PXD

PWD displays a pathlist that shows the path from the root directory to the user's current data directory. It can be used by programs to discover the actual physical location of files, or by humans who get lost in the file system. PXD displays the pathlist from the root to the current execution directory.

Example:

```
OS9: chd /D1/STEVE/TEXTFILES/MANUALS
OS9: pwd
/D1/STEVE/TEXTFILES/MANUALS
OS9: chd ..
OS9: pwd
/D1/STEVE/TEXTFILES
OS9: chd ..
OS9: pwd
/D1/STEVE
OS9: pxd
/D0/CMDS
```

RENAME

Change file name

Syntax: RENAME <path> <new name>

Gives the mass storage file specified in the pathlist a new name. The user must have write permission for the file to change its name. It is not possible to change the names of devices, ".", or ".."

Examples:

```
rename blue purple

rename /D3/user9/test temp



OS9: dir

   Directory of .  16:22:53
myfile          animals

OS9:rename animals cars
OS9:dir

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

SAVE

Save memory module(s) on a file

Syntax: SAVE <path> <modname> {<modname>}

Creates a new file and writes a copy of the memory module(s) specified on to the file. The module name(s) must exist in the module directory when saved. The new file is given access permissions for all modes except public write.

Note: SAVE's default directory is the current data directory. Executable modules should generally be saved in the default execution directory.

Examples:

```
save wordcount wcount

save /d1/math_pack add sub mul div
```

SETIME

Activate and set system clock

Syntax: SETIME [y,m,d,h,m,s]

This command sets the system date and time, then activates the real time clock. The date and time can be entered as parameters, or if no parameters are given, SETIME will issue a prompt. Numbers are one or two decimal digits using space, colon, semicolon or slash delimiters. OS-9 system time uses the 24 hour clock, i.e., 1520 is 3:20 PM.

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 year need be given, the date and time will be read from the clock.

Examples:

```
OS9: setime 82,12,22,1545     (Set to: Dec. 12, 1981, 3:45 PM)

OS9: setime 821222 154500     (Same as above)

OS9: setime 82                (For system with battery-backup clock)
```

SETPR

Set Process Priority

Syntax: SETPR <procID> <number>

This command changes the CPU priority of a process. It may only be used with a process having the user's ID. The process number is a decimal number in the range of 1 (lowest) to 255. The "procs" command can be used to obtain process ID numbers and present priority.

NOTE: This command does not appear in the CMDS directory as it is built-in to the SHELL.

For more information see: 5.1, PROCS

Examples:

```
setpr 8 250      (change process #8 priority to 250)
```

```
OS9: procs

Usr #  Id pty  state    Mem Primary module
----- --- --- -------- --- --------------
    0   3   0  waiting   2 Shell <TERM
    0   2   0  waiting   2 Shell <TERM
    0   1   0  waiting   1 Sysgo <TERM
```

```
OS9: setpr 3 128
OS9: procs

Usr #  Id pty  state    Mem Primary module
----- --- --- -------- --- --------------
    0   3 128  active    2 Shell <TERM
    0   2   0  waiting   2 Shell <TERM
    0   1   0  waiting   1 Sysgo <TERM
```

SLEEP

Suspend process for period of time

Syntax: SLEEP <tick count>

This command puts the user's process to "sleep" for a number of clock ticks. It is generally used to generate time delays or to "break up" CPU-intensive jobs. The duration of a tick is system-dependent but is typically 100 milliseconds on Level One systems and 10 milliseconds on Level Two systems.

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).

Example:

```
OS9: sleep 25
```

SHELL - OS-9 Command Interpreter

Syntax: SHELL <arglist>

The Shell is OS-9's command interpreter program. It reads data from its standard input path (the keyboard or a file), and interprets the data as a sequence of commands. The basic function of the shell is to initiate and control execution of other OS-9 programs.

The shell reads and interprets one text line at a time from the standard input path. After interpretation of each line it reads another until an end-of-file condition occurs, at which time it terminates itself. A special case is when the shell is called from another program, in which case it will take the parameter area (rest of the command line) as its first line of input. If this command line consists of "built in" commands only, more lines will be read and processed; otherwise control will return to the calling program after the single command line is processed.

The rest of this description is a technical specification of the shell syntax. Use of the Shell is described fully in Chapters 2 and 4 of this manual.

Shell Input Line Formal Syntax:

```
<pgm line> := <pgm> {<pgm>}
<pgm>  :=  [<params>]  [ <name> [<modif>] [<pgm params>] [<modif>] ]
{<sep>]
```

Program Specifications

```
<name>  := <module name>
        := <pathlist>
        := ( <pgm list> )
```

Parameters

```
<params>:= <param> { <delim> <param> }
<delim> := space or comma characters
<param> := ex <name> [<modif]     chain to program specified
        := chd <pathlist>         change working directory
        := kill <procID>          send abort signal to process
        := setpr<procID> <pty>    change process priority
        := chx <pathlist>         change execution directory
        := w                      wait for any process to die
        := p                      turn "OS9:" prompting on
        := -p                     turn prompting off
        := t                      echo input lines to std output
        := -t                     don't echo input lines
```

SHELL (continued)

```
          := -x                    don't abort on error
          :=  x                    abort on error
          := * <text>              comment line: not processed

<sep>     :=  ;    sequential execution separator
          :=  &    concurrent execution separator
          :=  !    pipeline separator
          :=  <cr> end-of-line (sequential execution separator)
```

Modifiers

```
<modif> := <mod> { <delim> <mod> }
<mod>   := < <pathlist>     redirect standard input
          := > <pathlist>     redirect standard output
          := >> <pathlist>    redirect standard error output
          := # <integer>      set process memory size in pages
          := # <integer> K    set program memory size
                              in 1K increments
```

TEE

Copy standard input to multiple output paths

Syntax: Tee {<path>}

This command is a filter (see 4.3.3) that copies all text lines from its standard input path to the standard output path and any number of additional output paths whose pathlists are given as parameters.

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 four terminals:

```
echo WARNING System down in 10 minutes ! tee /t1 /t2 /t3 /t4
```

TMODE

Change terminal operating mode

Syntax: TMODE [.<pathnum>] [<arglist>]

This command is used to display or change the operating parameters of the user's terminal.

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. A period and a number can be used to optionally specify the path number to be affected. If none is given, the standard input path is affected.

NOTE: If this command is used in a shell procedure file, the option ".<path num>" must be used to specify one of the standard output paths (0, 1 or 2) to change the terminal's operating characteristics. The change will remain in effect until the path is closed. To effect a permanent change to a device characteristic, the device descriptor must be changed.

This command can work only if a path to the file/device has already been opened. You may alter the device descriptor to set a device's initial operating parameter (see the System Programmer's Manual).

upc	Upper case only Lower case characters are automatically converted to upper case.
-upc	Upper case and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	no erase on backspace: echoes single backspace only
bsl	Backspace over line: lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are "deleted" by printing a "new line" sequence (for hard-copy terminals).
echo	Input characters "echoed" back to terminal (default)
-echo	No echo
lf	Auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.

TMODE (Continued)

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.

-pause Screen pause mode off.

null=n Set null count: number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal. default = 0.

pag=n Set video display page length to n (decimal) lines. Used for "pause" mode, see above.

bsp=h Set input backspace character. Numeric value of character in hexadecimal. Default = 08.

bse=h Set output backspace character. Numeric value of character in hexadecimal. Default = 08.

del=h Set input delete line character. Numeric value of character in hexadecimal. Default = 18.

bell=h Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07

eor=h Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D

eof=h Set end-of-file input character. Numeric value of character in hexadecimal. Default = 1B.

type=h ACIA initialization value: sets parity, word size, etc. Value in hexadecimal. Default = 15

reprint=h Reprint line character. Numeric value of character in hexadecimal.

dup=h Duplicate last input line character. Numeric value of character in hexadecimal.

psc=h Pause character. Numeric value of character in hexadecimal.

abort=h Abort character (normally control C). Numeric value of character in hexadecimal.

quit=h Quit character (normally control Q). Numeric value of character in hexadecimal.

xon=h DC1 resume output character (normally control Q). Numeric value of character in hexadecimal.

xoff=h DC2 suspend output character (normally control S). Numeric value of character in hexadecimal.

Examples:

```
tmode -upc lf null=4 bse=1F pause

tmode pag=24 pause bsl -echo bsp=8 bsl=C

tmode xon xoff quit=5
```

NOTE: If you use TMODE in a procedure file. it will be necessary to specify one of the standard output paths (.1 or .2) since the shell's standard input path will have been redirected to the disk file (TMODE can be used on an SCFMAN-type devices only).

Example:

```
tmode .1 pag=24              (set lines/page on standard output)
```

TSMON

Timesharing monitor

Syntax: TSMON [<pathlist>]

This command 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 the login fails because the user could not supply a valid user name or password, it will return to TSMON.

Note: The LOGIN command and its password file must be present for TSMON to work correctly (see the LOGIN command description).

Logging Off the System

Most programs will terminate when an end of file character (escape) is entered as the first character on a command line. This will log you off of the system and return control to TSMON.

For more information see: 2.5, LOGIN

Examples:

```
OS9:tsmon /t1&
&005
```

UNLINK

Unlink memory module

Syntax: UNLINK <modname> { <modname>}

Tells OS-9 that the memory module(s) named are no longer needed by the user. The module(s) may or may not be destroyed and their memory reassigned, depending on if in use by other processes or user, whether resident in ROM or RAM, etc.

It is good practice to unload modules whenever possible to make most efficient use of available memory resources.

Warning: never unlink a module you did not load or link to.

For more information see: 5.4, 5.4.1, 5.4.2

Example:

```
unlink pgm1 pgm5 pgm99



OS9: mdir

   Module Directory at 11:26:22
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         Sysgo       Clock
Shell       Tsmon       Edit

OS9: unlink edit
OS9: mdir

   Module Directory at 11:26:22
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         Sysgo       Clock
Shell       Tsmon
```

VERIFY

Verify or update module header and CRC

Syntax: VERIFY [U]

This command is used to verify that module header parity and CRC value of one or more modules on a file (standard input) are correct. Module(s) are read from standard input, and messages will be sent to the standard error path.

If the U (update) option is specified, the module(s) will be copied to the standard output path with the module's header parity and CRC values replaced with the computed values. A message will be displayed to indicate whether or not the module's values matched those computed by VERIFY.

If the option is NOT specified, the module will not be copied to standard output. VERIFY will only display a message to indicate whether or not the module's header parity and CRC matched those which were computed.

Examples:

```
OS9: verify <EDIT  >NEWEDIT

Module's header parity is correct.
Calculated CRC matches module's.

OS9: verify <myprogram1 >myprogram2

Module's header parity is correct.
CRC does not match.

OS9: verify <myprogram2

Module's header parity is correct.
Calculated CRC matches module's.

OS9: verify u <module >temp
```

This page is intentionally blank

This page is intentionally blank

OS-9 ERROR CODES

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

HEX	DEC	
$C8	200	PATH TABLE FULL - The file cannot be opened because the system path table is currently full.
$C9	201	ILLEGAL PATH NUMBER - Number too large or for non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL
$CB	203	ILLEGAL MODE: attempt to perform I/O function of which the device or file is incapable.
$CC	204	DEVICE TABLE FULL - Can't add another device.
$CD	205	ILLEGAL MODULE HEADER - module not loaded because its sync code, header parity, or CRC is incorrect.
$CE	206	MODULE DIRECTORY FULL - Can't add another module
$CF	207	MEMORY FULL - Level One: not enough contiguous RAM free. Level Two: process address space full
$D0	208	ILLEGAL SERVICE REQUEST - System call had an illegal code number.
$D1	209	MODULE BUSY - non-sharable module is in use by another process.
$D2	210	BOUNDARY ERROR - Memory allocation or deallocation request not on page boundary.
$D3	211	END OF FILE - End of file encountered on read.
$D4	212	RETURNING NON-ALLOCATED MEMORY - attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT - device has damaged file structure.
$D6	214	NO PERMISSION - file or device attributes do not permit access requested.

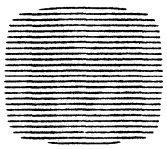
$D7	215	BAD PATH NAME - syntax error in pathlist (illegal character, etc.).
$D8	216	PATH NAME NOT FOUND - can't find pathlist specified.
$D9	217	SEGMENT LIST FULL - file is too fragmented to be expanded further.
$DA	218	FILE ALREADY EXISTS - file name already appears in current directory.
$DB	219	ILLEGAL BLOCK ADDRESS - device's file structure has been damaged.
$DD	221	MODULE NOT FOUND - request for link to module not found in directory.
$DF	223	SUICIDE ATTEMPT - request to return memory where your stack is located.
$E0	224	ILLEGAL PROCESS NUMBER - no such process exists.
$E2	226	NO CHILDREN - can't wait because process has no children.
$E3	227	ILLEGAL SWI CODE - must be 1 to 3.
$E4	228	PROCESS ABORTED - process aborted by signal code 2.
$E5	229	PROCESS TABLE FULL - can't fork now.
$E6	230	ILLEGAL PARAMETER AREA - high and low bounds passed in fork call are incorrect.
$E7	231	KNOWN MODULE - for internal use only
$E8	232	INCORRECT MODULE CRC - module has bad CRC value
$E9	233	SIGNAL ERROR - receiving process has previous unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE - unable to locate module
$EB	235	BAD NAME - illegal name syntax
$ED	237	RAM FULL - no free system RAM available at this time
$EE	238	UNKNOWN PROCESS ID - incorrect process ID number
$EF	239	NO TASK NUMBER AVAILABLE - all task numbers in use

DEVICE DRIVER ERRORS

The following error codes are generated by I/O device drivers, and are somewhat hardware dependent. Consult manufacturer's hardware manual for more details.

$F0	240	UNIT ERROR - device unit does not exist.
$F1	241	SECTOR ERROR - sector number is out of range.
$F2	242	WRITE PROTECT - device is write protected.
$F3	243	CRC ERROR - CRC error on read or write verify.
$F4	244	READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR - hardware error during disk write operation.
$F6	246	NOT READY - device has "not ready" status.
$F7	247	SEEK ERROR - physical seek to non-existent sector.
$F8	248	MEDIA FULL - insufficient free space on media.
$F9	249	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
$FA	250	DEVICE BUSY - non-sharable device is in use.
$FB	251	DISK ID CHANGE - Media was changed with files open.
$FC	252	RECORD IS LOCKED-OUT - Another process is accessing the requested record.
$FD	253	NON-SHARABLE FILE BUSY - Another process is accessing the requested file.

This page is intentionally blank



MICROWARE®

Microware Systems Corporation
5835 Grand Avenue, Box 4865, Des Moines, Iowa 50304
515-279-8844

LIMITED WARRANTY

Microware Systems Corporation ("Microware") warrants its software product(s) to be free from media or operational defects for a period of ninety (90) days from the date of shipment. Microware may correct any such defects by furnishing replacements or corrections, at its option, without cost to the purchaser.

Microware makes no other warranties, express or implied, of any kind including, but not limited to, merchantability or fitness of the product(s) for any particular purpose. It is the responsibility of the purchaser to determine the suitability and usefulness of the products for any particular application. Microware shall not be responsible for any damages, direct, indirect, or consequential, in connection with the use of its product(s). In any event, Microware's liability shall be limited to the purchase price of the product(s).

This warranty specifically excludes product(s) modified by the purchaser, or if used with equipment not specified by Microware as being suitable for use with the product(s).

SOFTWARE LICENSE AGREEMENT

Microware hereby grants a Limited Software License to the purchaser for use of the software product(s) on a single computer system. Reproduction of the product(s) in any manner, except for a reasonable number of backup copies, is expressly prohibited. Distribution of the products, in part or whole, to any other party constitutes misappropriation of valuable trade secrets and processes which are property of Microware and/or other parties, and causes damages far in excess of the value of the copies involved. This license shall remain in effect until all copies of the product(s) are destroyed by purchaser, or returned to Microware for disposal.

Acceptance or use of the product(s) shall be deemed implied consent to the terms of the license agreement. If the terms of this agreement are not acceptable to the purchaser, the product(s) can be returned for a full refund within ten days of the date of shipment, providing that the products are returned in good condition with all seals intact.

If you have questions concerning this agreement or warranty, or have a problem with a product, contact Microware by mail at P.O. Box 4865, Des Moines, IA 50304; by phone at (515) 279-8898; or TWX/Telex at 910-520-2535.