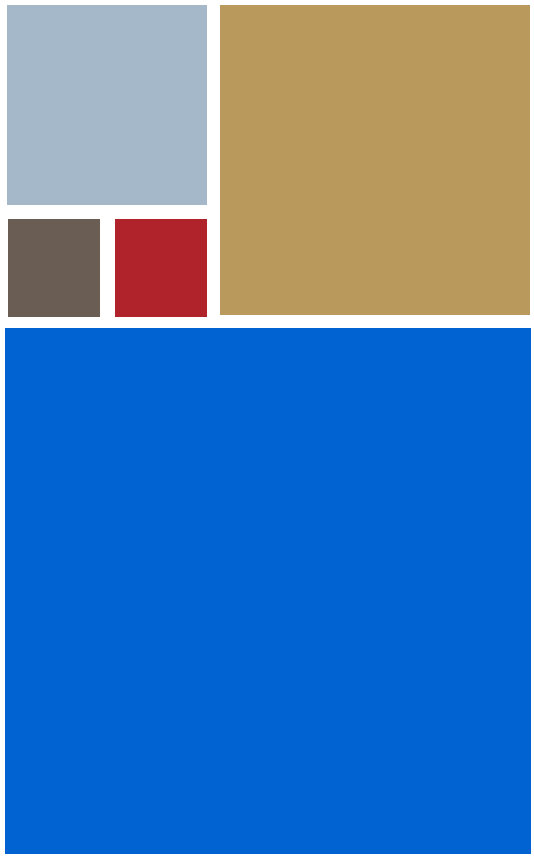


Home

Using OS-9 for 68K Processors

Version 3.3

RadiSys
THE POWER OF WE

www.radisys.com

Revision B • July 2006

Copyright and publication information

This manual reflects version 3.3 of OS-9 for 68K.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

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

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Table of Contents

Chapter 1: OS-9 for 68K Overview

This chapter is a general introduction to OS-9 for 68K. It introduces the concept of an operating system and explains some of OS-9's basic features. It includes the following topics:

What Is an Operating System?

An operating system is the master supervisor of the resources and functions of a computer system. Computer resources consist of:

- Memory
- CPU time
- Input/output devices such as terminals, disk drives, and printers

OS-9 is a sophisticated operating system for microcomputers. OS-9's basic functions are to:

- Provide an interface between the computer and the user
- Manage the input/output (I/O) operations of the system
- Provide for loading and executing programs
- Create and manage a system of directories and files
- Manage timesharing and multitasking
- Allocate memory for various purposes
- Allocate and manage interprocess communication services

Using OS-9 as Your Operating System

The most visible function of the operating system is its role as an interface between you and the complex internal hardware and software functions of the system. OS-9 was designed to make its powerful features easy to use, even by persons with limited technical knowledge.

Because an operating system provides only part of the overall software necessary to make the computer useful, application programs such as word processors and accounting packages tend to be the most frequently used programs. They are not part of the operating system, but they rely heavily on services such as input and output provided by the operating system. Most application programs are written by users or obtained from commercial software suppliers.

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

To help make it easy to use, OS-9 includes a set of programs called utilities. Utilities are not part of the basic operating system; they are small application programs providing essential housekeeping, management, customizing, and maintenance functions. Some utilities, such as the µMACS text editor, are useful, general-purpose application programs. Others, such as `procs`, provide information about a specific function of the operating system.

Using OS-9's Functions

There are two basic ways to use OS-9's many capabilities and functions:

- The first method uses the utility command set and the shell command interpreter program. This allows you to type OS-9 commands directly on your keyboard. These commands are translated into the more complex internal system calls actually required to carry out the desired operations.

For More Information

Refer to the ***Utilities Reference*** manual for descriptions of all OS-9 utilities.

- The second method uses system calls. System calls are requests made to OS-9 within programs written in assembler or a high-level language. These system calls are available to:
  - Load programs into memory
  - Create new tasks
  - Create or delete files
  - Read, write, open, or close files

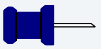
For More Information

System calls are largely of interest to advanced programmers and are covered in detail in the ***OS-9 for 68K Technical Manual***.

All OS-9 programming languages have statements causing the program to use OS-9 system calls, often in a hidden manner.

Storing Information

Another basic function of any operating system is storing information. Without some way to store and organize your programs, data, and text, working on a computer would be extremely complicated.



Note

OS-9 stores information in files and directories located on mass-storage devices such as floppy disks. It provides easy access methods for updating, storing, and retrieving files and directories through standard utilities.

OS-9 organizes all files into organizational structures called directories. A directory is actually a special file containing the names and locations of each file it contains. Directories can contain files and subdirectories. In turn, these subdirectories may contain other files and subdirectories. This is called a tree structure, or hierarchical, organization for file storage.

For More Information

For more information about the file structure, refer to **Chapter 4: The OS-9 File System**.

Multitasking and Multi-user Functions

OS-9 is a *multitasking* and *multi-user* operating system.

Multitasking, or multiprocessing, allows the computer to run many different programs at the same time. By rapidly switching from one program to the next, many times per second, programs appear to run at the same time.

Each program running on the system is called a task, or process. OS-9 allows you to have one or more tasks running in the background while a task is running in the foreground.

A foreground process is a task requiring your interaction. For example, if you are editing a file, it is a foreground process because you are actively using it. A program that prompts you for information is also a foreground process because you need to respond to it.

Note

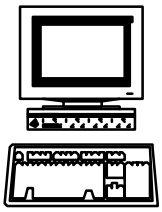
A *foreground process* requires your interaction.

A *background process* does not require your attention.

A background process is a task that does not require your attention. For example, if you are printing a text file, you do not have to supervise the printing process. Therefore, you can have the file printing in the background while you edit another file. This frees the computer from the limitation of doing only one thing at a time.

OS-9's multitasking capabilities make it possible for efficient memory use, CPU time, and I/O operations to be shared by all programs without conflict.

Figure 1-1 Typical Multitasking Use

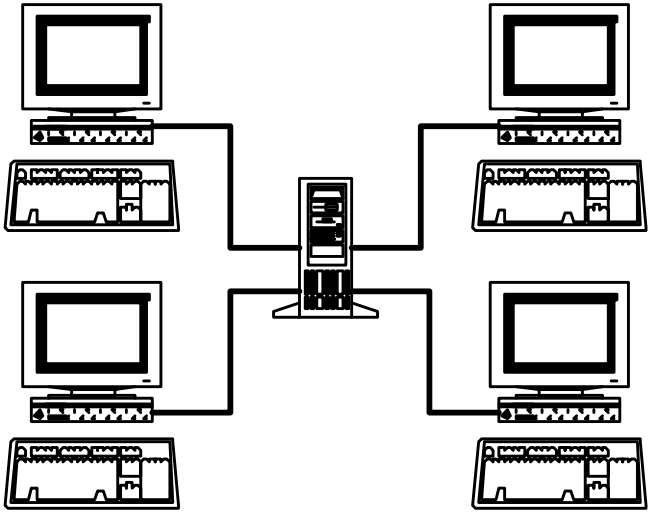


Typical Multitasking Use:

- Editing a file (foreground process)
- Listing a file to a printer (background process)
- Sorting and merging data files (background process)

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

Figure 1-2 Typical Multi-User System Configuration



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

The Memory Module and Modular Software

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

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

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

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

Another advantage of memory modules is frequently used functions can share common *library* modules. For example, a standard OS-9 module called `csl` provides a wide range of I/O processing for virtually all programming languages and programs. This eliminates the need for each program to include its own standard I/O package. In addition, you can split large and complex programs into smaller modules that are easier to test.

The MWOS Directory Structures

The directory structure introduced in OS-9 for 68K Version 3.0 represents a significant departure from its predecessor. Its design was influenced by a growing number of users developing not only under OS-9, but UNIX and DOS. Microware has adopted this general directory structure for all of its products.

The `MWOS` directory structure is designed to:

- Provide a consistent directory structure for all development platforms.
- Provide similar development environments for OS-9 and OS-9 for 68K.
- Allow code sharing between OS-9 and OS-9 for 68K.
- Make provisions for code and libraries optimized for 32 bit processors.
- Provide a clear division between the development and runtime directories.
- Allow for multiple ports from a common set of sources.
- Provide a means to create a disk-based runtime system without modifying makefiles.

About the Directory Structure

The new structure is built under the `MWOS` directory. As you descend through the directories, the files become progressively more OS, CPU, and hardware dependent. A simplified model appears in **Figure 1-3**. For a more detailed examination, we suggest recursively walking down the directory structure of your newly installed product.

Sources particular to an operating system (OS) are kept in `MWOS/<OS>/SRC`. Sources common between all OS's are located in `MWOS/SRC`. The same logic applies to C header files and assembler defs. Ports for particular boards are kept under the `<OS>/<Processor family>/PORTS` directories.

Figure 1-3 MWOS File Structure

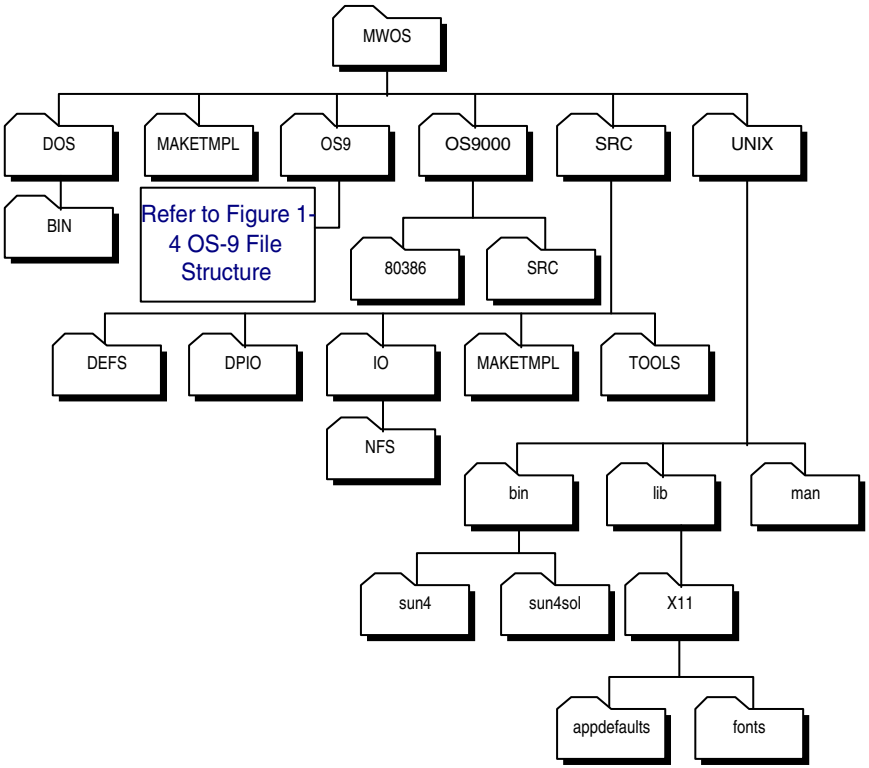
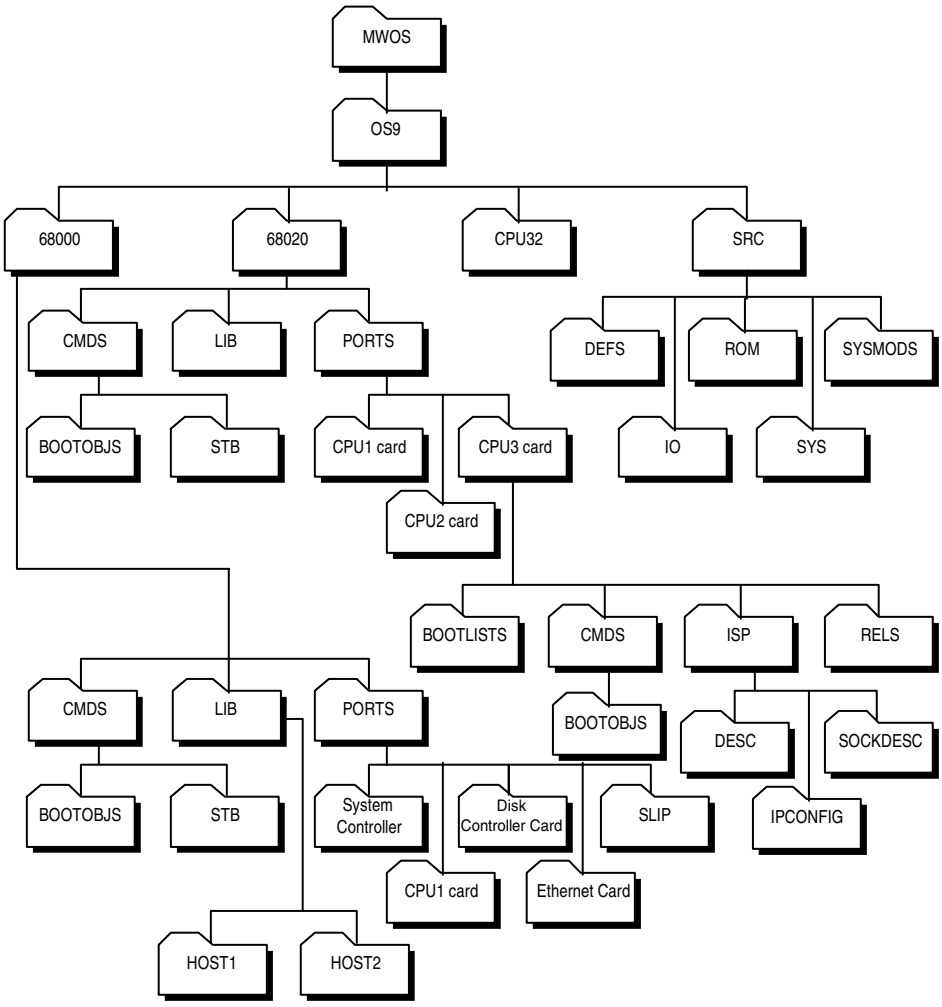


Figure 1-4 MWOS/OS9 File Structure



A few examples may be useful here. Where performance is not an issue, it is Microware's practice to compile OS-9 for 68K software products with a 68000 compiler, allowing execution on all Motorola 680X0 MPUs. Most utilities fall into this category and are found in `MWOS/OS9/68000/CMDS`. When there are significant performance benefits to be gained from compiling with a 32 bit compiler, such as with the ISP system modules, the executables are found in `MWOS/OS9/68020/CMDS`.

If you are an doing a port of OS-9 to a new 68040 CPU card, find the kernel and any other processor specific modules in the MWOS/OS9/68020/CMDS/BOOTOBJS directory. The remainder of the hardware-independent modules are in MWOS/OS9/68000/CMDS/BOOTOBJS. CPU card specific components of the OS are found in MWOS/OS9/<CPU>/PORTS/<YOUR CPU>.

Example boot driver source code is found in MWOS/OS9/SRC/ROM. Example high level driver sources are found in MWOS/OS9/SRC/IO/<FILE MANAGER>/DRVR.

Another useful example is for those doing cross development on a Windows host. An OS-9 targeted cross compiler resides in MWOS/DOS/BIN/ along with other cross-hosted utilities. Makefiles should target the appropriate OS and CPU.

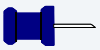
Development vs. Runtime

The MWOS directory structure is specifically oriented towards software development. Whether the development occurs on a resident OS-9 system or a cross development environment, once the executable modules have been created they are moved to their final locations on the target machine.

When you are developing an application on a resident development system, this might be simply a matter of copying a file from the MWOS/OS9/<CPU>/CMDS directory to the /H0/CMDS directory. It might involve downloading the modules into memory on a small target system, making a boot on a server to boot the target over ethernet, or creating a set of ROMs for a fully ROMed system.

Disk-based runtime systems are similar to their pre-V3.0 counterparts. Contents of system dependent directories are generally lifted to the root of the system drive, while (if desirable) a mirror image is kept within MWOS. For example, an OS-9 for 68K Version 2.4 MVME-147 development pack looked like this:

```
     Directory of . 17:10:43
C       CMDS    DEFS         IO         ISP
LIB     SYS     SYSCACHE     SYSMODS    init.ramdisk
startup
```



Note

Development (source) directories like `C`, `IO`, `SYSCACHE`, and `SYSMODS` appeared on the root.

An OS-9 for 68K Version 3.0 MVME-147 Extended Board Support Pak (our example's nearest counterpart) would look like this:

```
     Directory of . 17:10:43
CMDS    SYS     ETC     MWOS
```

All sources, header files, and libraries are now under the `MWOS` directory. Depending on the application, the executables found in `CMDS`, the system startup file(s) found in `SYS`, and the network (Internet and/or NFS) database files are found in `ETC`. At the system administrator's option, these files may also be duplicated in `MWOS` so they may be modified and tested prior to committing them for use on the development system.

Directories such as `USR`, `TFTPBOOT`, and other directories used on your OS-9 systems can continue to reside in their current location. The `SYS` and `LIB` directories may continue to reside on the root or on RAM disks if desired.

Please see the Ultra C documentation for additional information about the `MWOS` file structure. The sources provided in Microware products use pathlists for defs and libs that stay within the `MWOS` directory structure.

The sources and makefiles are designed to allow the relocating of the `MWOS` directory. Multiple `MWOS` directories may be created for different versions of OS-9 and OS-9 for 68K.

ISP, NFS, and Other Package's Directories

The ISP and NFS `cmds` and system modules are now located in the target system's `CMDS` and `CMDS/BOOTOBJS` directories. This simplifies the startup procedures for both systems and allows utilities to be loaded as they are needed without long `PATH` searching.

The startup procedures for these packages still allow the utilities to be loaded at startup, but the practice is no longer required. You may choose to move the systems modules to the boot so no loading is required.

Directories Contained on the System Disk

The following is a list of directories commonly distributed with OS-9 for 68K. They are all contained in the primary directory (the root directory) of your system disk.

Figure 1-5 Root Directories

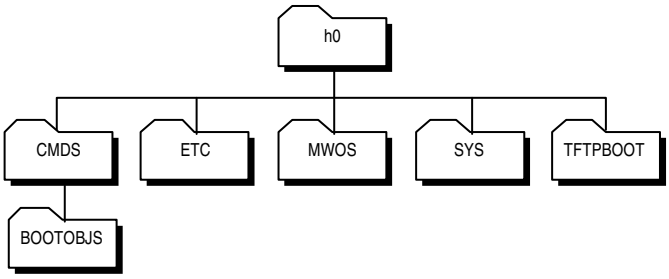


Table 1-1 Root Directories

Directory	Contains
CMDS	The standard OS-9 utilities for running the system. Additional user-created programs and OS-9 modules to be executed from a shell command line.
SYS	System files and startup scripts for use in bringing up the system, allowing logins, and others, including: Errmsg — Text for descriptions of error messages. An appendix listing the error messages is included with this manual set. password — A sample password file for timesharing systems. The password file contains information such as the user name, password, and initial process for each user. termcap — Descriptions of your terminal characteristics.
MWOS	Microware Operating System development directory structure. See the following pages for more information on the MWOS structure.
ETC	Contains the data files used to create the Inetdb and rpcdb configuration modules.

Table 1-1 Root Directories (continued)

Directory	Contains
`TFTPBOOT`	If the system is a `bootp` server, this is the default directory for bootfiles for the client machines.
`CMDS/BOOTOBJS`	This directory should contain any system modules that are to be loaded after the system is booted. If the `MWOS` directory is not otherwise needed on the target machine, you may choose to keep the modules required for remaking the system boot in this directory.

For More Information

For more information about:

- Each utility distributed with OS-9, refer to the ***Utilities Reference*** manual.
- Changing device descriptors, refer to **Chapter 8: OS-9 for 68K System Management**.
- The password file, refer to **Chapter 5: The Shell**, and the *login* utility in the ***Utilities Reference*** manual.

- The termcap file, refer to **Chapter 8: OS-9 for 68K System Management**.

Figure 1-6 MWOS Directories

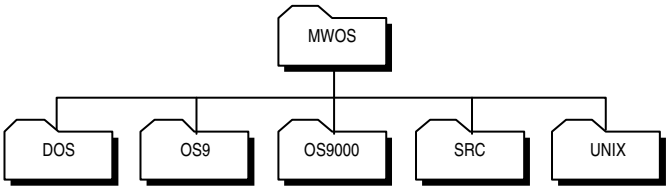


Table 1-2 MWOS Directories

Directory	Contains
`OS9`	All OS-9 for 68K object code is targeted under this directory. All OS-9 for 68K specific source code, defs files, libraries, processor family code, and ports reside here.
`OS9000`	All OS-9 directories (same as above).
`SRC`	All sources that are not specific to either OS-9 or OS-9 for 68K. C defs, common I/O systems, user tools, and Dual Ported I/O (DPIO) are examples of code found under the `MWOS/SRC` directory.
`UNIX`	Similar to other OS directories. Contains the objects for cross development tools for use on a variety of UNIX based development systems.
`DOS`	Similar to other OS directories. Contains the objects for cross development tools for use on a Windows®-based development system.

Figure 1-7 MWOS/OS9 Directories

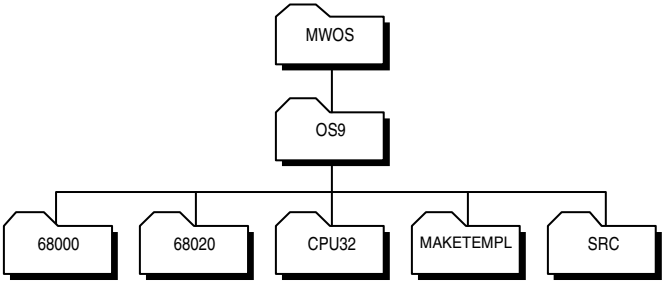


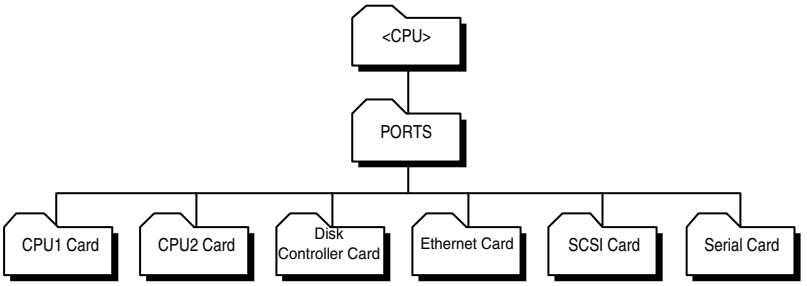
Table 1-3 MWOS/OS-9 Directories

Directory	Contains
68000	The object code and libraries specific to the 68000 family of processors or binaries created to run on all versions of the Motorola MC68xxx family of processors. Most OS-9 utilities are compiled to run on all processors. In some cases (such as the Internet utilities), speed concerns require having versions compiled to the 68020 or CPU32 families. The 68000 directory also contains code for the 68010, 68070, and 68302 processors.
68020	The cmds and libraries specific to the 68020/68030/68040 processors.
CPU32	Files specific to the CPU32 family, such as the 68332, 68340, and 68349 processors.
SRC	The source files for the OS-9 drivers, descriptors, system modules, defs, and macros. SRC is intended to be a source directory containing hardware-specific code written to be reusable from target to target. It is not intended to be the repository for final object modules built from this source, although intermediate object files may be found within its subdirectories.
MAKETEMPL	A directory for common makefile templates (include files for makefiles). In this release, any templates found in this directory apply only to makefiles for ISP and related products.

To create the CMDS and CMDS/BOOTOBJS directories on the root of a 68030 based system, first dsave the 68000 CMDS directory, then dsave the 68020 CMDS directory onto the target machine. This provides all of the base utilities and modules from the 68000 directory and the 68020/68030 modules from the 68020 directory.

Each CPU directory has a PORTS subdirectory. The ports subdirectory provides directories for a variety of target system boards.

Figure 1-8 Ports Directories



Generally, if you are going to use peripheral cards with a variety of CPU cards, you should locate them under the 68000 directories. Drivers and card ports specific to 68020 or CPU32 family processors would be located under their respective <CPU>/PORTS directory.

Each card subdirectory has a structure that includes CMDS and CMDS/BOOTOBJS directories. CPU card directories may additionally contain a BOOTLISTS subdirectory for use in creating boots from within the MWOS directory structure.

Figure 1-9 Source Directories

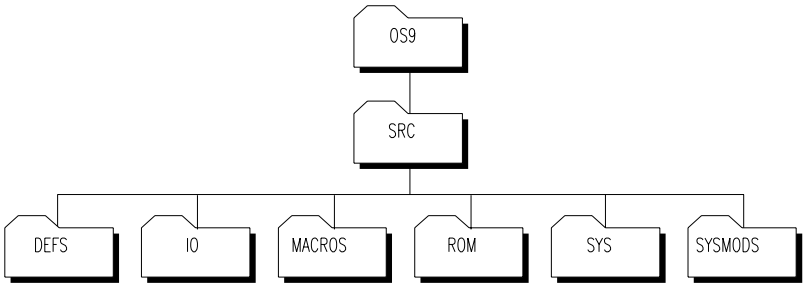
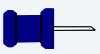


Table 1-4 Source Directories

Directory	Contains
DEFS	Files of definitions that apply system-wide, or are processor independent. These are both assembler `.d` and `C .h include` files.
IO	Sources for all OS-9 I/O subsystems including file managers, drivers, and descriptors. The file's subdirectories are organized by subsystem.
MACROS	Files of assembly language macro definitions that apply system-wide or are target independent.
ROM	Sources for rebuilding boot ROM components, except for a few that share source with SCSI drivers in IO.
SYS	A repository for files and scripts that end up residing in the OS-9 `SYS` directory on the root of the system disk.
SYSMODS	Sources for system extension modules.



Note

The level of source code available under the `SRC` directory depends on the type of package you purchased.

Chapter 2: Starting OS-9

This chapter describes how to get OS-9 up and running. This includes formatting and backup procedures. It includes the following topics:

- **Booting OS-9**
- **Backing Up the System Disk**

Booting OS-9

For More Information

Refer to **Chapter 1: OS-9 for 68K Overview** for a description of the directories commonly supplied with OS-9.

Before using OS-9 on your computer, you must *boot* the system. Booting is also called a *cold start* or *bootstrapping*. It involves the computer reading a portion of the system disk (or tape) into memory.

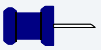
If your system is a standard disk-based computer, the system disk contains all the modules that make up OS-9. The system disk usually contains other files and directories frequently used during normal operations. This includes a directory for each user, a shared commands directory, and files used by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

For More Information

Chapter 8: OS-9 for 68K System Management contains information on changing the startup and `OS9Boot` files.

You should be familiar with two files, called `startup` and `OS9Boot`:

`startup`	is a shell procedure file processed immediately after the system starts running. `startup` may contain any legal OS-9 command or program.
`OS9Boot`	contains the OS-9 system modules that are read into memory.



Note

The boot procedure depends on the requirements of your specific hardware. The manufacturer supplies detailed instructions outlining the boot procedure for the specific system involved. Follow those instructions as specified.

Failure to Boot

If the system fails to boot:

Step 1. Recheck the hardware setup instructions, especially if you made any modifications to your computer.

Step 2. Make sure you inserted the disk (or tape) correctly, and try the boot sequence again.

Step 3. Make sure you followed the manufacturer's booting instructions.

If the boot sequence fails several times, contact your supplier.

Setting the System Time and Date

When the system boots correctly, a welcome message is displayed followed by the `setime` prompt. The `setime` utility starts the system clock and allows OS-9 to track the date and time of the creation of new files. The clock must be running for multitasking to take place.

For More Information

Refer to:

- The ***Utilities Reference*** manual for more information about `setime` and `date`.
- **Chapter 8: OS-9 for 68K System Management** for more information about the `Init` module and the system clock.

The `Init` module may inform the kernel to automatically start the clock from a battery-backed clock. If the clock is not started and you have a system with a battery-backed clock, type the following command to start the system clock:

```
$ setime -s
```

Otherwise, execute `setime` by typing:

```
$ setime
```

`setime` prompts with the following:

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

At the prompt, enter the year, month, day, hour, minutes, seconds, and optionally `am` or `pm`. Unless you specify `am` or `pm`, `setime` uses the 24 hour clock. For example, 15:20 is the same as 3:20 p.m. The input is one or two digit numbers with a space, colon, semicolon, comma, or slash used as a field delimiter. If you use a semicolon, the entire date string must be within quotes. For example, to set the time on May 14, 1993 at 1:24 p.m., type one of the following:

- `93/5/14/1/24/pm`
- `93 05 14 1 24 pm`
- `93,5,14,13,24`
- `93:5:14:13:24`

- `93/5/14/13/24`
- `"93;5;14;13;24"`

Checking the Date and Time

To find out if the system clock is running or if the date and time is correct, use the `date` utility. For example:

```
$ date
July 2, 1993 Monday 1:25:26pm
```

The System Prompt

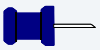
For More Information

For information on changing the shell prompt, refer to **Chapter 5: The Shell**.

Once you set the time and date, the system displays the following prompt:

```
$
```

The `$` prompt means the operating system is active and waiting for you to enter a command line. This is the default system prompt. This manual uses the `$` prompt for all examples.



Note

The following sections are specifically intended for systems distributed with floppy disk system disks. These sections are also of general interest in terms of formatting and backing up floppy disks. If you have a hard disk or are booting from a media other than a disk, refer to **Chapter 8: OS-9 for 68K System Management**.

Backing Up the System Disk

For More Information

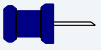
Refer to:

- The ***Utilities Reference*** manual for more information about `format` and `backup`.
- A list of naming conventions OS-9 uses is located in **Chapter 5: The Shell**.

Before experimenting with OS-9, make a backup of your master system disk. The backup procedure involves making an exact copy of a disk. If your system disk becomes damaged, it may become unreadable. For this reason, it is important to have another copy stored safely away.

Before you can backup your system disk, you need a properly formatted disk. OS-9 cannot read from or write to new disks until they have been formatted. The `format` utility initializes new disks for reading and writing. `backup`, the OS-9 utility that makes copies of disks, *requires* the backup disk to be the same size and format as the original disk.

The following section provides the steps to follow to backup a disk on a typical OS-9 system booting from a floppy drive (usually called `/d0`).



Note

Before formatting your first disk, we strongly recommend you read the entire section on formatting disks.

Formatting a Disk

The format of OS-9 system disks vary by the type of disk drive and by manufacturer. Usually, the format is set to the maximum capacity of the disk drive.

You can place several parameters on the command line with the `format` command:

`-sd` for single density disks.

`-dd` for double density disks.

`-ss` for single sided disks.

`-ds` for double sided disks.

For More Information

The `format` utility description in the ***Utilities Reference*** manual contains additional information about `format`.

Refer to your hardware documentation for the maximum capacity of your drives. Refer also to the label of your system disk for the proper format of your backup copy.

Multiple Drive Format

If your system has two disk drives, place the system disk in the first drive and the new disk in the second drive. The second drive is usually called `/d1`. At the `$` prompt, type `format`, the drive name of the new disk, any desired options, and press the `<return>` key to enter the command line:

```
$ format /d1
```

This command line specifies to format the disk in the second drive as a double-sided, double-density disk. If your disk is different, use different options.

Single Drive Format

If your system has only one disk drive, you must load the `format` utility into memory. The `load` utility puts a copy of a program into the computer's memory. Once `format` has been loaded into memory, you can remove your system disk from the drive. OS-9 can execute the copy of `format` residing in memory. You can load and execute any OS-9 utility in this fashion.

To load the `format` utility into memory, type the following command at the `$` prompt:

```
load format
```

After you load `format` complete the following steps:

Step 1. Remove the system disk from the drive.

Step 2. Place the disk to format into the drive.

Step 3. Enter the following at the `$` prompt to format the disk:

```
format /d0
```

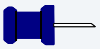
For More Information

Refer to the ***Utilities Reference*** manual for more information about the `load` utility.

Continuing the Formatting Process with Either a Single Drive or a Multiple Drive

In the case of both single and multiple drive systems, `format` displays the specific disk format settings, followed by a prompt:

```
Formatting device: <drive name>
proceed?
```



Note

`<drive name>` is replaced by the name of the device on which you are trying to format. For example, `/d0`.

WARNING

If the drive name in the prompt is not the name of the drive with the blank disk, type q to quit, or you may erase your only system disk.

If the drive name and parameters in the prompt are correct, type `y` for yes. If you type `y` at the prompt, there is a pause while the disk is being formatted. `format` then prompts for the name of the disk:

```
volume name:
```

After you enter the volume name, `format` prints:

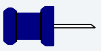
```
verifying media, building bitmap...
```

During the final phase of the process, the hexadecimal number of each track is displayed as each track is verified to see if any sectors are bad. If any bad sectors are found, an error message is displayed along with the number of the bad sector. The number of good sectors, the number of unusable sectors, and the total number of verified sectors are also displayed.

The Backup Procedure

After a disk is formatted, you can run `backup`. The `backup` utility makes an exact copy of the OS-9 system disk. There are other ways to make a copy of a disk, but this method is the least complicated. The backup process involves copying everything from your system disk to a formatted disk.

During the backup procedure, the system disk is referred to as the `source disk`. The backup disk is called the `destination disk`.



Note

This procedure makes copies of any disk, not just the system disk.

`backup` makes two passes.

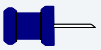
- The first pass reads a portion of the source disk into a buffer in memory and writes it to the destination disk.
- The second pass verifies everything was copied to the new disk correctly.

Generally, if an error occurs on the first pass, something is wrong with the source disk or the drive it is in.

If an error occurs during the second pass, the problem is with the destination disk. If `backup` repeatedly fails on the second pass, reformat the disk to make sure it has no bad sectors. If the disk reformats correctly, try the backup procedure again.

WARNING

Never backup a system disk to a disk that has any bad sectors reported by `format`.



Note

You may wish to *write protect* your source disk when using the *backup* procedure. This prevents any confusion in exchanging the source and destination disks.

Multiple Drive Backup

If your system has two disk drives complete the following steps:

Step 1. Place the source disk in the first drive (`/d0`).

Step 2. Place the destination disk in the second drive (`/d1`).

Step 3. Type `backup` at the `$` prompt.

Step 4. Press the `<return>` key.

The system assumes you want to backup the disk in `/d0`. It responds to `backup` with the following prompt:

```
ready to BACKUP /D0 to /D1?
```

Table 2-1 Responses to Backup Question

Type	If
`y`	The correct disks are in the correct drives.
`q`	The disks are not in the correct drives. You exit the backup procedure when you enter `q`.

If you type y, the system copies all information on the disk in /d0 onto the disk in /d1 and returns the $ prompt.

Single Drive Backup

If your system has only one drive, you need to load the backup utility into memory. Make sure your system disk is in /d0 and type the following command:

```
load backup
```

After you have loaded backup, you may proceed with the backup procedure. Type the following command:

```
backup /d0 -b=100k
```

This tells the system you are performing a single drive backup and you want to use a 100K buffer for the backup. If your system allows you to use a larger buffer, increase this number. The larger the buffer, the fewer swaps you have to make. The system responds with the following prompt:

```
ready to BACKUP /D0 to /D0?
```

Table 2-2 Responses to Backup Question

Type	If You Are
y	Ready to perform the backup.
q	Not ready to perform the backup. You exit the backup procedure when you enter q.

If you type y, the system begins a series of prompts to complete the backup procedure. This consists of swapping the source and destination disks in the disk drive as prompted by the system.

The first prompt is:

```
ready destination, hit a key
```

At this prompt do the following:

Step 1. Remove the source disk from the drive.

Step 2. Insert the destination disk.

Step 3. Press any key to continue the backup procedure.

The next system prompt is:

```
ready source, hit a key
```

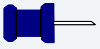
At this prompt, do the following:

Step 1. Remove the destination disk from the drive.

Step 2. Insert the source disk.

Step 3. Press any key to continue the backup procedure.

The exchanging of disks continues until the backup procedure is completed.



Note

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

Chapter 3: Basic Commands and Functions

This chapter helps you get started using the operating system. The more frequently used system commands are discussed. These are utilities every user should be familiar with.

This chapter includes the following topics:

Learning the Basics

Now that your system is up and running, it is time to learn about OS-9's basic features and utilities. This chapter and the chapter on the OS-9 file system provide an introduction to OS-9 to get you started quickly.

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

Logging on to a Timesharing System

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

Until you press the `<return>` key, idle terminals on multi-user systems do nothing but beep at you. Pressing the `<return>` key starts the log-on program called `login`. `login` maintains system security and starts each user with a personalized environment.

The system asks for your user name and the password the system manager assigned to you. The system echoes your user name, but for security purposes your password is not echoed. You have three chances to enter a valid user name and password.

The following is an example of the `login` procedure:

```
OS-9/68040 V3.0 Vite_MVME167 - 68040 93/10/24 14:51:12
User Name: smith
Password: [not echoed]
Process #10 logged on 93/10/24 14:51:20
Welcome!
$
```

Depending on how the system is set up, a system-wide *message of the day* may display on your screen. You can also automatically run one or more initial programs. In addition, you are normally set up in your own main working directory.

To log off, simply press the `<escape>` (end-of-file) key or type `logout` any time your main shell is active.

For More Information

For more information about `login` and `tsmon`, refer to the ***Utilities Reference*** manual.

An Introduction to the Shell

Every operating system has a command interpreter. A command interpreter is a translator between the commands you type in and the commands the operating system understands and executes. OS-9's command interpreter is called the `shell`.

The shell provides many functions and options. **Chapter 5: The Shell** is exclusively devoted to the available shell features. This section is intended to provide just enough familiarity with the shell for you to run basic OS-9 commands.

The `shell` is normally started as part of the system startup sequence on a single user system or after logging on to a timesharing system. It is the primary interface with the system. When you enter a command, the shell translates the command into something OS-9 can understand.

The shell functions in two ways:

1. Accepts interactive commands from your keyboard.
2. Reads a sequence of command lines from a special type of file called a *procedure file*. The shell executes each command line in the procedure file just as if the command lines had been typed in manually from the keyboard. Procedure files are a convenient way to eliminate typing frequently used, identical sequences of commands.

When the shell is ready for command input, it displays a `$` prompt. You can now enter a command line followed by a carriage return.

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

- An OS-9 utility
- An application program or programming language
- A procedure file

Most commands require or accept additional parameters or options. These parameters and options provide the program and/or the shell with additional information such as file names and directory names to search. Almost all options are preceded by a hyphen (`-`) character. All parameters are separated by space characters.

The shell follows a special searching sequence to locate the command in memory or on disk. If it cannot find the command you specified, the error `#000:216`, "`file not found`" is generally reported.

Here is an example of a simple shell command line:

```
$ list myfile
```

The name of the program is `list`. The file name `myfile` is passed to the program.

Using the Keyboard

Most input to OS-9, programming languages, and application programs is line oriented. This means as you type, the characters are collected but not sent to the program until you press the <return> key. This gives you a chance to correct typing errors before they are sent to the program.

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

Line Editing Control Keys

The line editing control keys are:

Table 3-1 Line Editing Control Keys

Key	Function
<control>a	Repeat the previous input line. The last line entered is redisplayed but not executed. The cursor is positioned at the end of the line. You may enter the line as it is or you can add more characters to it. You can edit the line by backspacing and typing over old characters.
<control>d	Redisplay the current input line. This is mainly used for hardcopy terminals that cannot erase deleted characters.
<control>h	Backspace to erase previous characters. Most keyboards have a special <backspace> key you can use directly without using the <control> key.

Table 3-1 Line Editing Control Keys (continued)

Key	Function
<control>q	Resume the input and output previously stopped by <control>s. The <control>q function is known as X-On.
<control>s	Halt input and output until <control>q is entered. The <control>s function is known as X-Off. Many serial I/O devices, such as printers, use this feature to control output speed.
<control>w	Temporarily halt output so you can read the screen before data scrolls off. Output resumes when any other key is pressed. See the section on the page pause feature.
<control>x	Delete line; erase the entire current line.
ESCAPE or <control>[	Indicate the end-of-file: all OS-9 I/O devices, including terminals, are accessed as files. This simulates the effect of reaching the end of a disk file.

Interrupt Keys

There are also two important control keys called *interrupt* keys. They work differently than the line editing keys because you can use them at any time, not just when a program requests input. They are normally used to halt or alter a running program.

Table 3-2 Interrupt Keys

Key	Function
`<control>c`	Send an interrupt signal to the most recent program. This functions differently from program to program. If a program does not make specific interrupt provisions, it aborts the program. If a program has provisions for interrupts, `<control>c` usually provides a way to stop the current function and return to a master menu or command mode. In the shell, you can use `<control>c` to convert the *foreground* program to a *background* program, if the program has not begun I/O to the terminal.
`<control>e`	Send a *program abort* signal to the program presently running. In most cases, this key prematurely aborts the current program and returns you to the shell.

For More Information

For more information about `tmode`, refer to **Chapter 8: OS-9 for 68K System Management** and the ***Utilities Reference*** manual.

These control keys are the key assignments commonly used in most OS-9 systems. You can change the correspondence between control keys and their functions, so your keys may be different. Use the `tmode` utility to redefine the function of control keys. This command allows you to customize OS-9 to the specific computer's keyboard layout.

The Page Pause Feature

The page pause feature eliminates the annoyance of having output scroll off the screen before you can read it. OS-9 counts output lines until a full screen has been displayed. It then halts output until you press any key. This is repeated for each screen of output.

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

You can use `tmode` to turn this feature on and off, or to change the number of lines per screen:

Table 3-3 Page Pause Feature

Key	Function
`tmode pause`	Turn the page pause mode on.
`tmode nopause`	Turn the page pause mode off.
`tmode pag=10`	Set the page length to ten lines.

Basic Utilities

OS-9 provides over seventy standard utilities and built-in shell commands. Most utilities are used rarely, if ever, by casual users. You will frequently use less than a dozen of them and less frequently use about a dozen more. Some of the most frequently used utilities are listed below.

Table 3-4 Frequently Used Utilities

attr	backup	build	chd
chx	copy	date	del
deldir	dir	dsave	echo
edt	format	free	help
kill	list	makdir	merge
mfree	pd	pr	procs
rename	set	setime	shell
w	wait		

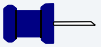
The help Utility and the -? Option

The most important command to learn when beginning to use the OS-9 utilities is `help`. The `help` utility is an on-line, quick reference. To use this utility, type `help`, a utility name, and a carriage return. The utility function, syntax, and available options are listed.

For example, if you cannot remember the function or syntax of the `backup` utility, you could type `help backup` after the `$` prompt:

```
$ help backup
Syntax: backup [<opts>] [<srcpath> <dstpath>] [<opts>]
Function: backup disks
Options:
-b=<size>   use larger buffer (default is 4k)
-r          don't exit if read error occurs
-v          do not verify
$
```

The descriptions are short and precise. Try it. This is a quick way to find information without looking up the utility in the documentation.



Note

Typing `help` by itself displays the syntax and use of the `help` utility.

The same information is also available by typing the utility name followed by a question mark (`-?`). Each utility has the `-?` option.

free and mfree

After booting your OS-9 system, you may wish to see how much memory and unused disk space is available. You can use the `mfree` and `free` utilities to do this. `mfree` is useful for all systems, `free` is useful for disk-based systems.

`mfree` displays the amount of unused memory available in the system. For example:

```
$ mfree
Current total free RAM: 164.00 K-bytes
```

For a complete list of information concerning the unused memory, you can use the `-e` option with `mfree`. For example:

```
mfree -e
Minimum allocation size:       4.00 K-bytes
Number of memory segments:   6
Total RAM at startup:       8192.00 K-bytes
Current total free RAM:     2084.00 K-bytes
Free memory map:
     Segment Address          Size of Segment
   -----------------   --------------------------
       $5B000          $1000          4.00 K-bytes
       $5F000          $2000          8.00 K-bytes
       $99000          $1E3000     1932.00 K-bytes
       $29C000         $3000         12.00 K-bytes
       $2A1000         $1F000       124.00 K-bytes
       $2C5000         $1000          4.00 K-bytes
```

`free` displays the amount of unused disk space in the number of sectors and in the number of bytes. It also displays the disk name, its creation date, and the cluster size of the device. For example:

```
$ free
"Tazz: /H0 Wren V" created on: Aug 17, 1993
Capacity: 2347860 sectors (256-byte sectors, 8-sector clusters)
1477296 free sectors, largest block 1356000 sectors
378187776 of 601052160 bytes (360.66 of 573.20 Mb) free on media (62%)
347136000 bytes (331.05 Mb) in largest free block
```

`free` uses a 4K buffer by default. To increase the buffer size, use the `-b` option. For example, to use a 10K buffer you could type one of the following:

- `$ free -b=10`
- `$ free -b10`

Chapter 4: The OS-9 File System

This chapter is a detailed explanation of the tree-structured file and directory system of OS-9. It includes the following topics:

- **OS-9 File Storage**
- **The OS-9 File System**
- **Current Directories**
- **Accessing Files and Directories: The Pathlist**
- **Basic File System Oriented Utilities**

OS-9 File Storage

All information stored on an OS-9 computer system is organized into files and directories. Files and directories provide a way for you to organize your information:

- A file may contain a program, data, or text.
- A directory is a file containing the names and locations of the files and directories it contains.

This allows you to organize your files by topic, work group, or any other method.

When a file is created, the information is stored as an ordered sequence of *bytes*. These bytes are organized into *sectors*. A sector is a pre-defined group of bytes. For example, a sector may be composed of 256 bytes. This means every 256 bytes are grouped together as a sector.

During the format procedure, each sector is marked as being unused. The allocation map keeps track of each sector. If a sector is in use, it is marked in the allocation map located at the beginning of each disk as being in use.

Table 4-1 Sectors in the Allocation Map

When a File Is	Action
Created	The information is stored in sectors.
Expanded	The new information is stored in sectors.
Shortened/ Deleted	The previously used sectors are unmarked in the allocation map and are available for use by other files.

Within a text file, each byte contains one character. Data is written to a file in the order it is provided. Data is read from a file exactly as it is stored in the file.

The File Pointer

When a file is created or opened, a file pointer is also created and maintained for it. The file pointer holds the address of the next byte to write or read (see **Figure 4-1**). As data in the file is read or written, the file pointer is automatically moved. Therefore, successive read or write operations transfer data sequentially (see **Figure 4-2**).

You can use an OS-9 system call (`seek`) to directly access any part of a file by positioning the file pointer to any location in the file.

You can access the `seek` system call through the various languages available for OS-9 or directly with the macro assembler command: `I$SEEK`.

For More Information

For more information about `I$SEEK`, refer to the ***OS-9 for 68K Technical Manual***.

Figure 4-1 shows when creating or opening a file, the file pointer is positioned to read from or write to the first component.

Figure 4-1 File Pointer at File Creation or File Open

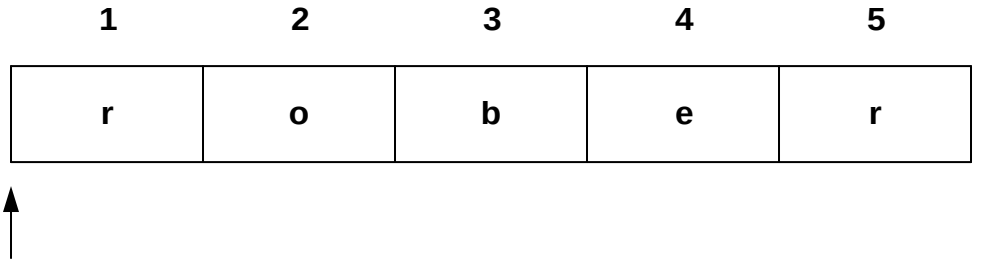


Figure 4-2 shows after reading or writing the first component of a file, the file pointer points to the second component.

Figure 4-2 Pointer After Reading First Component

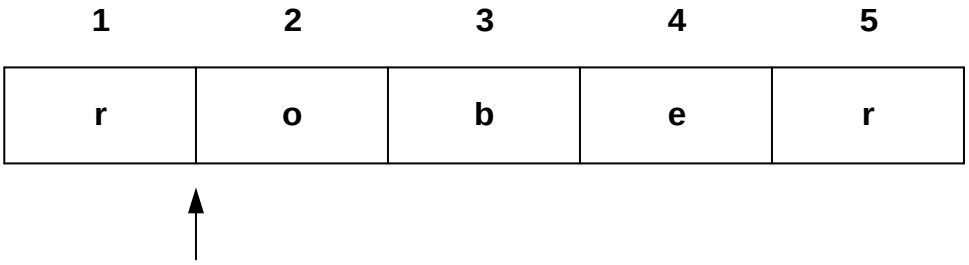


Figure 4-3 shows the file pointer is pointing to the current end-of-file. Attempting another read operation causes an end-of-file error. Another write operation increases the size of the file.

Figure 4-3 File Pointer Pointing to Current End-of-File

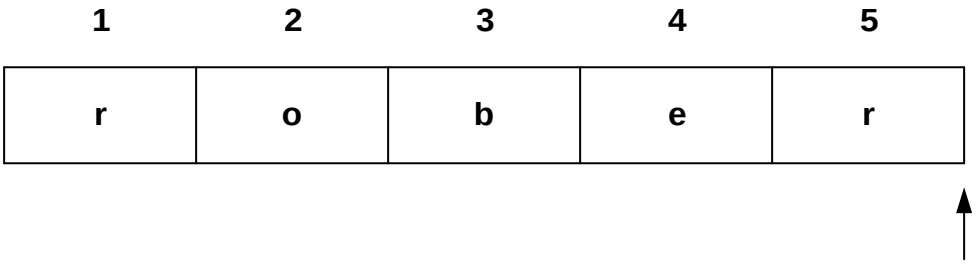
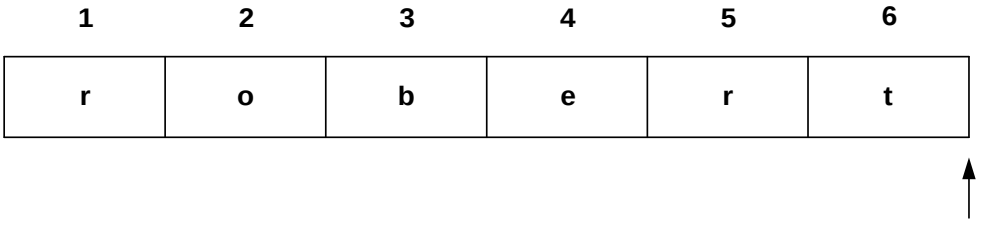


Figure 4-4 shows the next write operation adds a new component to the file and moves the file pointer to the new end-of-file.

Figure 4-4 Adding A New Component



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

Because all OS-9 files have the same physical organization, you can generally use file manipulation utilities on any file regardless of its logical use. The main logical types of files used by OS-9 are:

- Text files
- Executable program module files
- Data files
- Directories

Directory files are an exception and are covered separately.

Text Files

Text files contain variable length lines of ASCII characters. Each line is terminated by a carriage return (`hex $OD`). Text files typically contain documentation, procedure files, and program source code. You can create text files with any text editor or the `build` utility.

Executable Program Module Files

Executable program modules store programs that assemblers and compilers generate. Each file may contain one or more modules with standard OS-9 module format. The ***OS-9 for 68K Technical Manual*** contains more information about modules.

Random Access Data Files

A random access data file is created and used primarily by high level languages such as C, Pascal, and BASIC. The file is organized as an ordered sequence of records of varying sizes. If each record has exactly

the same length, its beginning address within the file can be computed to allow you to access records in any order. OS-9 does not directly deal with records other than providing the basic file manipulation functions high level languages that support random access records require.

File Ownership

When you create a file or directory, OS-9 automatically stores a *group.user ID* with it. The group.user ID is formed from your group number and your user number.

- The group number allows people working on the same project or working in the same department to share a common group identification.
- The user number identifies a specific user.

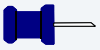
Therefore, a `group.user` ID identifies a specific user in a specific group or department.

The group.user ID determines file ownership. OS-9 users are divided into two classes:

- owner
- public

The owner is any user with the same group number as the person who created the file. The super-user group (`0.x`) is also considered the owner of the file.

The public is any person with a group ID differing from the person who created the file.



Note

A user with a `group.user` ID of `0.0` is referred to as a *super user*. A super user can access and manipulate any file or directory on the system regardless of the file's ownership.

On multi-user systems, the system manager generally assigns the group.user ID for each user. This number is stored in a special file called a *password file*. A super user on a multi-user system is generally the system manager, although other people such as group managers or project leaders may also be super users.

For More Information

For more information about password files, refer to **Chapter 5: The Shell**.

On single-user systems, users have super user status by default.

Attributes and the File Security System

File use and security are based on file attributes. Each file has eight attributes. These attributes are displayed in an eight character listing.

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

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

```
dsewrewr
```

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

Table 4-2 Attributes

Attribute	Abbreviation	Description
Owner Read	`r`	The owner can read the file. When off, this denies any access to the file.
Owner Write	`w`	The owner can write to the file. When off, this attribute can be used to protect files from accidentally being deleted or modified.
Owner Execute	`e`	The owner can execute the file.
Public Read	`pr`	The public can read the file.
Public Write	`pw`	The public can write to the file.
Public Execute	`pe`	The public can execute the file.
Single User	`s`	When set, only one user at a time can open the file.
Directory	`d`	When set, indicates a directory.

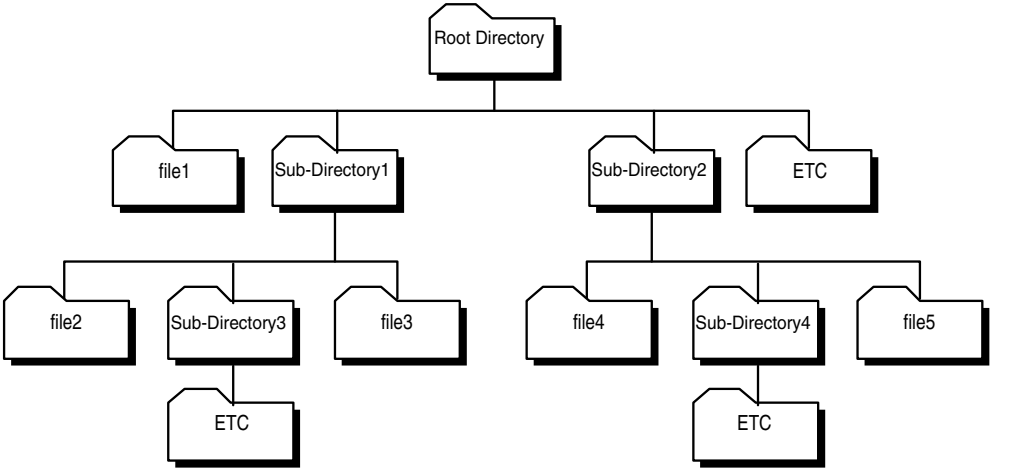
The OS-9 File System

OS-9 uses a *tree-structured*, or hierarchical, organization for its file system on mass storage devices such as disk systems (see **Figure 4-5**). Each mass storage device has a master directory called the *root directory*.

The root directory is created automatically when a new disk is formatted. It contains the names of the files and the subdirectories on the disk. Every file is listed in a directory by name, and each file has a unique name within a directory.

An OS-9 directory can contain both files and subdirectories. Each subdirectory can contain more files and subdirectories. This allows you to embed subdirectories within other subdirectories. The only limit to this division is the amount of available disk space.

Figure 4-5 The File System



With the exception of the root directory, each file and directory in the system has a *parent* directory. A parent directory is the directory directly above the file or directory being discussed. For example in **Figure 4-5**, the parent directory of `file2` is `SUB-DIRECTORY1`. Likewise, the parent directory of `SUB-DIRECTORY1` is the root directory.

Current Directories

Two working directories are always associated with each user or process. These directories are called the *current data directory* and the *current execution directory*.

Note

A *data directory* is where you create and store your text files.

An *execution directory* is where executable files such as utilities and programs you have created are located.

The current directory concept allows you to organize your files while keeping them separate from other users on the system. The word *current* is used because you can use the `chd` and `chx` commands to move through the tree structure of the OS-9 file system to a different directory. This new directory then becomes your current data or execution directory.

For More Information

For more information about `chd`, refer to the ***Utilities Reference*** manual. `chd` is also covered later in this chapter.

On Single-User Systems

On a single user system, OS-9 chooses the root directory of your system disk as your initial current data directory. Your initial current execution directory is the `CMDS` directory. The `CMDS` directory is located in the root directory of the system disk.

On Multi-User Systems

On a multi-user system, your current data and execution directories are established for you as part of the initial login sequence. When you login, your initial directories are set up according to your password file entry. A password entry is established for each user on a multi-user system. This entry lists information such as the user's password and current directories.

For More Information

For more information about password files, refer to **Chapter 5: The Shell** and the `login` utility description in the ***Utilities Reference*** manual.

Your execution directory on a multi-user system is usually the `CMDS` directory, which is shared with other users. `CMDS` contains OS-9 utilities and other executable files. If all users had their own copy of all OS-9 commands, a great deal of disk space would be wasted. Private execution directories are also possible and are covered later in this chapter.

The Home Directory

On typical multi-user systems, all users have their own data directory, but share an execution directory. The private data directory allows you to organize your own files by project, function, or any other method without affecting other user's files. The data directory specified in the password file entry is known as your *home directory*. When you first login to the system, you are placed in this directory. Using the `chd` utility with no parameters also places you in this directory.

On single user systems, you may establish a home directory by setting the `HOME` environment variable.

For More Information

For more information about:

- `chd`: refer to the ***Utilities Reference*** manual. `chd` is also covered later in this chapter.
- The `HOME` environment variable: refer to **Chapter 5: The Shell**.

Directory Characteristics

Some important characteristics relating to directory files are:

- Directories have the same ownership and attributes as regular files. However, directories always have the `d` attribute set.
- Each file name within a directory must be unique. For example, you cannot store two files named `trial` in the same directory. Files can have identical names, as long as they are stored in different directories.
- All files are stored on the same device as the directory in which they are listed.
- The only limit to the number of files you can store in a directory is the amount of free disk space.

Accessing Files and Directories: The Pathlist

You can access all files or directories in your current data directory by specifying the name of the file or directory after the proper command. When only a file or directory name is given, OS-9 does not look outside your current data directory to find it.

If you want to access a file that is not in your current data directory or run a program that is not in your current execution directory, you must either change your current directory or specify a *pathlist* through the file system for OS-9 to follow.

There are two types of pathlists:

- **Full Pathlists**
- **Relative Pathlists**

Full Pathlists

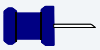
A full pathlist starts at the root directory and follows the directory names in the list down the file structure to a specific file or directory. A full pathlist must begin with a slash character (/). Slashes separate names within the pathlist.

The following example is a full pathlist from the root directory, `/d1`, through two subdirectories, `PASCAL` and `TESTS`, to the file `futureval`.

```
/d1/Pascal/tests/futureval
```

The next example specifies a path from the root directory, `/h0`, through the `USR` subdirectory to the `NICHOLLE` subdirectory.

```
/h0/usr/nicholle
```



Note

A full pathlist begins at the root directory regardless of where your current data directory is located. It lists each directory located between the root directory and a specific file or subdirectory.

Relative Pathlists

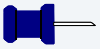
A *relative* path starts at the current directory and proceeds up or down through the file structure to the specified file or directory. A relative pathlist does not begin with a slash (/). Slashes separate names within a relative pathlist.

When you use a relative pathlist and the desired destination requires going up the directory tree, you can use special naming conventions to make moving around the pathlist easier:

- A single period (.) refers to the current directory.
- Two periods (..) refer to the current directory's parent directory.
- Add a period for each higher directory level.

For example, to specify a directory two levels above the current directory, three periods are required. Four periods refer to a directory three levels above the current directory.

You can also use a UNIX-style pathlist such as ../../../.



Note

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

The following example is a relative pathlist beginning in your current directory and goes through the subdirectories `DOC` and `LETTERS` to the file `jim`.

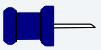
```
doc/letters/jim
```

The next pathlist goes up to the next directory above your current directory and then through the subdirectory `CHAP` to the file `page`.

```
../chap/page
```

The next pathlist specifies a file within your current directory. No directories are searched other than the current directory.

```
accounts
```



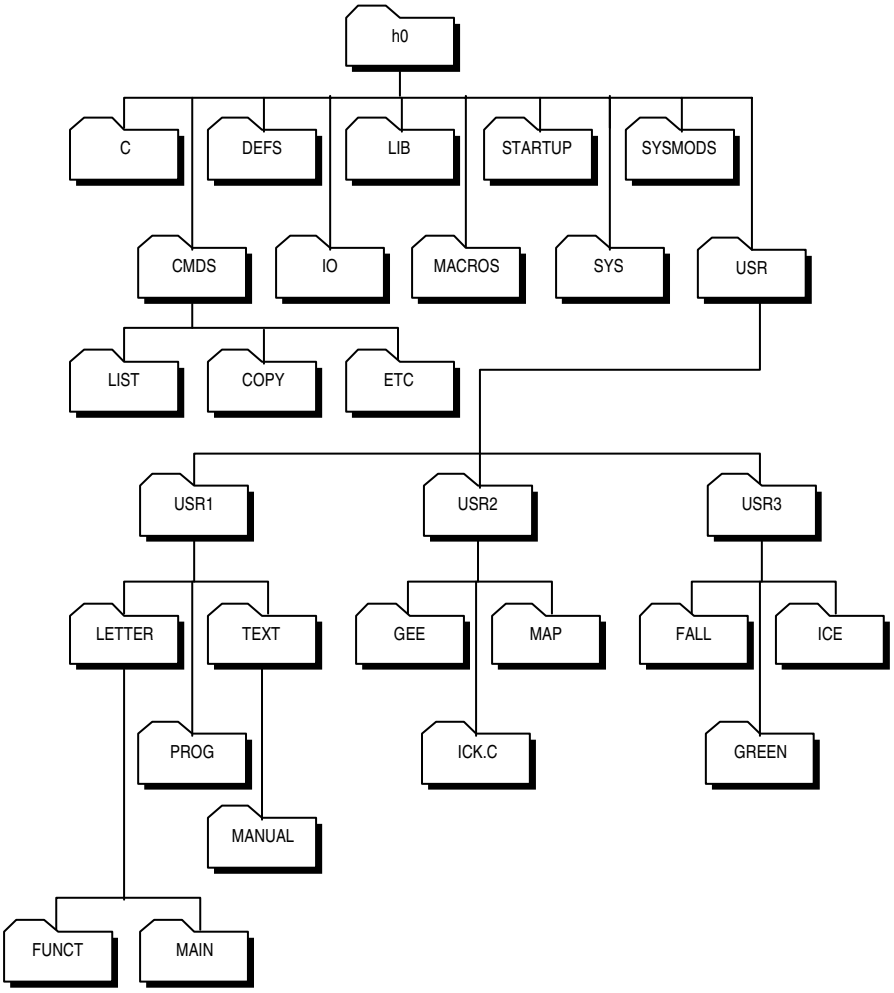
Note

A relative pathlist begins at your current directory regardless of its location in the overall file structure.

Basic File System Oriented Utilities

This section explains some of the OS-9 utility commands that manipulate the file system. The utilities include `dir`, `chd`, `chx`, `pd`, `build`, `makdir`, `list`, `copy`, `dsave`, `del`, `deldir`, and `attr`. The examples given refer to an example file system (**Figure 4-6**).

Figure 4-6 Diagram of a Typical File System



dir: Display Directory Contents

The `dir` utility displays the contents of directories. Typing `dir` by itself displays the contents of your current data directory. For the following example, the current data directory is `/h0` in **Figure 4-6**:

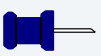
```
$ dir

directory of .  13:56:58
C       CMDS    DEFS       IO    LIB
MACROS  SYS     SYSMODS    USR   startup
```

To look at directories other than your current data directory, you must either provide a pathlist to the desired directory or change your current data directory.

For example, if you are in the root directory and you want to see what is in the `DEFS` directory, type:

```
dir defs
```

`dir` now displays the names of the files in the `DEFS` directory. The name `defs` is a relative pathlist. You can type `dir defs` because `DEFS` is in your current data directory. You can also use the full pathlist, `dir /h0/defs`, and get the same result.



Note

To display the contents of another directory without changing your current data directory, type `dir` and the pathlist to the directory.

For More Information

More information about changing directories is provided later in this chapter.

Wildcards and dir

To display the contents of your current execution directory, type `dir -x`.

You may also use wildcards with `dir` and with most other utilities as well. OS-9 recognizes two wildcards:

An asterisk (*)	An asterisk is replaced by any number of letter(s), number(s), or special characters. Consequently, an asterisk by itself expands to include all of the files in a given directory.
A question mark (?)	A question mark is replaced by a single letter, number, or special character.

For example, the command `dir *` lists the contents of all directories located in the current data directory. The command `dir /h0/cmds/d*` lists all files and directories in the `CMDS` directory beginning with the letter `d`. The command `dir prog_?` lists all files in your current directory having a file name with `prog_` followed by a single character.

For More Information

Chapter 5: The Shell contains more information about the use of wildcards.

dir Options

`dir` has several options that are fully documented in the ***Utilities Reference*** manual. The `-e` and `-r` options are discussed here. Try each option and see what information is displayed.

The `-e` option gives an *extended directory listing*. An extended directory listing displays all files within the specified directory with their attributes, the size of the file, and the sector where the file is stored. The following example uses the file structure shown in **Figure 4-6**.

```
$ dir usr/usr1 -e
Directory of USR/USR1 12:30:00
 Owner   Last Modified  Attributes Sector Bytecount Name
-------  -------------  ---------- ------ --------- ----
 12.4    93/06/17 1601   ------wr    3458      5744 letter
 12.4    93/07/03 1148   d-----wr   104A0     15944 PROG
 12.4    93/05/13 1417   d-----wr    DODO     11113 TEXT
```

The `-r` option displays the contents of the specified directory and any files contained within its subdirectories. Using **Figure 4-6** as an example, typing `dir usr/usr1 -r` lists the following:

```
Directory of . 12:30:15
  PROG    TEXT   letter
Directory of PROG 12:30:15
  funct   main
Directory of TEXT 12:30:15
  manual
```

You can use the `dir` options with each other. Typing `dir -er` displays all files within the current data directory, all files within its subdirectories, and provides an extended listing of their attributes, sizes, etc.

chd and chx: Moving Around in the File System

The `chd` and `chx` utilities allow you to travel around the file system:

- `chd` changes your current data directory.
- `chx` changes your current execution directory.

Using chd

To change your current data directory, type `chd` followed by a full or relative pathlist.

For example, if your current data directory is `/h0` and you want your current data directory to be `USR`, you would type `chd` and the pathlist of `USR`.

- Using a relative pathlist, type:

```
chd usr
```

- Using a full pathlist, type:

```
chd /h0/usr
```

Your current data directory is now `USR`. If you type `dir`, you see the contents of `USR`:

```
           directory of . 14:04:32
USR1    USR2    USR3
```

If you want to see which files are in the `USR1` directory, type `dir usr1`. Or change directories by typing `chd usr1` and after the new prompt, type `dir`.

If you want to return to your home directory, which in this case is `/h0`, type `chd` without a pathlist. After changing directories, `dir` displays the contents of `/h0`.

Using chx

The `chx` command allows you to redefine an existing directory as a personal execution directory. This may be important if you have programs you do not want other people to execute. To use this command, type `chx`, followed by a full or relative pathlist to the directory. When using a relative pathlist with `chx`, the pathlist is relative to your current execution directory.

If your current data directory is `USR` and you want to change your current execution directory from `CMDS` to `USR2`, you could type the relative pathlist `chx ../usr/usr2` or the full pathlist `chx /h0/usr/usr2`. When you type a command after you have changed your current execution directory, OS-9 searches `USR2` instead of `CMDS`.

Typing `dir -x` displays the contents of your current execution directory, `USR2`:

```
          directory of . 14:05:06
gee  ick.c   map
```

Climbing Directory Trees

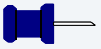
You can use OS-9's special naming conventions to move around the file system. As a reminder, the naming conventions are periods specifying the current directories and directories higher in the file structure. For example:

`.` refers to the current directory

`..` refers to the parent directory

`...` refers to two directory levels higher

When used as the first name in a path, you can use these naming conventions with relative pathlists.



Note

If you plan to port your code to other operating systems, remember most operating systems only use this convention as it refers to the current and parent directories. For example, if you use `...` to refer to the directory above a parent directory, most operating systems require you to use `../..` instead.

The following examples relate to the file structure in **Figure 4-7**. The examples assume your initial current data directory is `PROG`.

The following example displays the contents of `PROG`. It is functionally the same command as `dir`:

```
dir .
     directory of . 14:04:32
funct   main
```

The next command displays the contents of `PROG`'s parent directory, `USR1`.

```
dir ..
     directory of ..  14:05:58
PROG    TEXT    letter
```

This example displays the contents of TEXT by specifying a path starting with the parent directory (..):

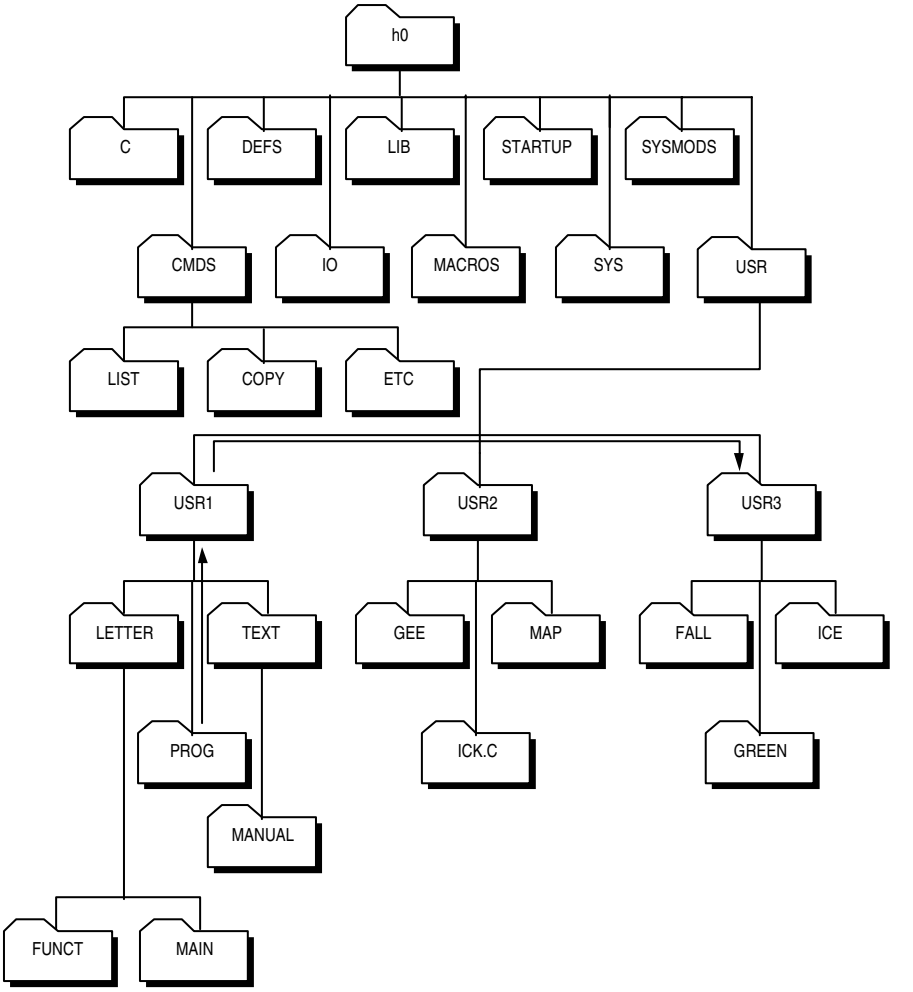
```
dir ../text
     directory of ../text 14:06:47
manual
```

The following command changes the current data directory from PROG to USR3:

```
chd .../usr3
```

In **Figure 4-7**, USR3 is accessed from PROG using the relative path .../usr3.

Figure 4-7 Accessing Directories Using a Relative Path



You can use any number of periods (.) to access higher directories. One period is added for each additional level. An error is not returned if you specify a greater number of directory levels above your current data directory than actually exist. Instead, this indicates the root directory on your system. For example, this command displays the contents of the root directory:

```
dir .........
```

This may be helpful if you are not sure how far down you are in the directory structure. The next example changes your current data directory from PROG to MACROS:

```
chd ...../macros
```

Using the pd Utility

When the file system becomes complex, you may become confused as to where the directory you are currently working in is located in relation to the overall file system. You can use the pd utility to display the complete pathlist from the root directory to your current data directory.

For example, if your current data directory is USR2:

```
pd
/h0/USR/USR2
```

Using makdir to Create New Directories

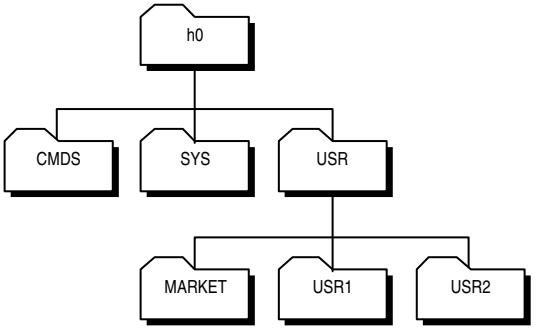
Likewise, if you forget which directory is your current execution directory, type pd -x to display the pathlist to the current execution directory.

To create new directories, use the makdir utility. For example, to create a directory called MARKET, type:

```
makdir MARKET
```

The makdir /h0/usr/MARKET command creates a new directory called MARKET in the USR directory.

Figure 4-8 Creating the /h0/USR/MARKET Directory



`MARKET` now is a new entry in your current directory.

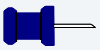
If you want the new directory created somewhere other than your current directory, you must specify a pathlist. For example, `makdir /h0/usr/MARKET` creates the new directory in `USR`.

Rules for Constructing File Names

When creating files and directories, you must follow certain rules. Any file name can contain from 1 to 28 upper or lower case letters, numbers, or special characters as listed below. While the file name may begin with any of the following characters or digits, each file name must contain at least one letter or number. Within these limitations, a name can contain any combination of the following:

Upper case letter:	`A - Z`	Underscore:	`_`
Lower case letter:	`a - z`	Period:	`.`
Decimal digits:	`0 - 9`	Dollar sign:	`$`

File names may not contain spaces. Instead, use an underscore (_) or a period (.) to improve the readability of file and directory names. OS-9 does not distinguish upper case letters from lower case letters. The names `FRED` and `fred` are considered the same name.



Note

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

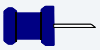
Here are some examples of legal names:

```
raw.data.2    project_review_backup
X6809         $SHIP.DIR
...c          12345
```

Here are some examples of illegal names:

Table 4-3

Name	Reason
`Max*min`	* is not a legal character
`open orders`	name cannot contain a space
`this.name.has.more.than.28.characters`	too long



Note

File names starting with a period are not displayed by `dir` unless you use the `-a` option. This allows you to hide files within a directory.

Creating Files

You can create files in many ways. Text files are generally created with the `build` utility, the `edt` utility, or the µMACS text editor. These file building tools are provided with the OS-9 package for your convenience.

For More Information

For more information about `build` and `edt`, refer to the ***Utilities Reference*** Manual.

The build Utility

Use the `build` utility to create short text files. To use `build`, type `build`, followed by the name of the file you want to create. `build` responds with the prompt:

```
?
```

This tells you `build` is waiting for input. To terminate `build`, type a carriage return at the *?* prompt. For example:

```
$ build test
? Some programmers have been known to
? howl at full moons.
?
$
```

You cannot edit files with `build`.

The edt Utility

You can also use the `edt` utility to create files. `edt` is a line-oriented text editor allowing you to create and edit source files. To use `edt`, type `edt` and the desired pathlist. `edt` displays a question mark (`?`) prompt and waits for an edit command. If the file is found, `edt`:

- Opens it
- Displays the last line
- Displays the `?` prompt

µMACS

For More Information

For more information about `µmacs`, refer to the ***Utilities Reference*** manual.

The preferred method of creating and editing files is with µMACS. µMACS is a screen-oriented text editor designed for creating and modifying text files and programs. Through the use of multiple buffers, µMACS allows you to display different files or different portions of the same file on the same screen. In addition, extensive formatting commands allow you to:

- Reformat paragraphs with new user-defined margins
- Transpose characters
- Capitalize words
- Change words or sections into upper or lower case

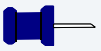
Examining File Attributes with attr

When you create a file using `build` or µMACS, only the owner read and owner write permissions are set. When you create a directory, it initially has all the permissions set except the single user permission.

To examine file attributes, use the `attr` utility. To use this utility, type `attr`, followed by the name of a file. For example:

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

The file `newtest` has the permissions set for owner reading and owner writing. Access to this file by anyone other than the owner is denied.



Note

Users with the same group.user ID as the person who created the file are considered owners. However, if the file is created by a group 0 user, only users in the super group can read, write, or execute the file.

If you use `attr` with a list of one or more attribute abbreviations, the file's attributes are changed accordingly, provided you have the proper write permission to access the file. You do not need to list the attribute abbreviations in any particular order. The letter `n` preceding an attribute removes that permission.

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

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

If you are the owner of a file, you can change the access permissions regardless of what the permissions indicate. Thus, the owner always has the right to delete a file, change the user privileges, etc. Users in the same group have the same permissions as the owner.

The directory attribute is somewhat different than the other attributes. It could be dangerous to be able to change directory files to normal files or a normal file to a directory. For this reason, you cannot use `attr` to

turn the directory (d) attribute on; use makdir to turn this attribute on. Furthermore, you can only use attr to turn the directory attribute off if the directory is empty.

Listing Files

Use the list utility to display the contents of files. By default, list displays the lines of text on your terminal screen. To examine a file, type list, followed by the name of the file. For example:

```
$ list test
Some programmers have been known to
howl at full moons.
$
```

For More Information

For more information about list, refer to the ***Utilities Reference*** manual.

It is important to remember you cannot list a directory. If you type the command list USR, the following error message and error number are returned:

```
list: can't open "USR". Error# 000:214.
```

This means you cannot access USR because it is a directory.

list displays text files. All distributed files in CMDS are executable program module files. If you try to list the contents of a random access data file or an executable program module file, you see what appears to be random data displayed on your screen. This may also include unprintable characters, such as escape or delete, that could change your terminal's operating parameters. If the operating characteristics of your terminal are affected, first try turning the terminal off and on. If this does not re-initialize the terminal, consult your terminal operating manual.

Copying Files

Use the `copy` utility to make a duplicate of a file. To copy a file, type `copy`, followed by the name of the file to be copied, followed by the name of the duplicate file. For example:

```
$ copy test newtest
```

If you `list` the file `newtest`, it is an exact copy of `test`.

The file you are copying and the duplicate file can be located in any directory; they do not have to be in your current data directory. For files located outside of your current data directory, use full or relative pathlists. The following example uses Figure 4-8. The first command copies the file *gee* in the `USR2` directory to a file named `new.info` in the `TEXT` directory:

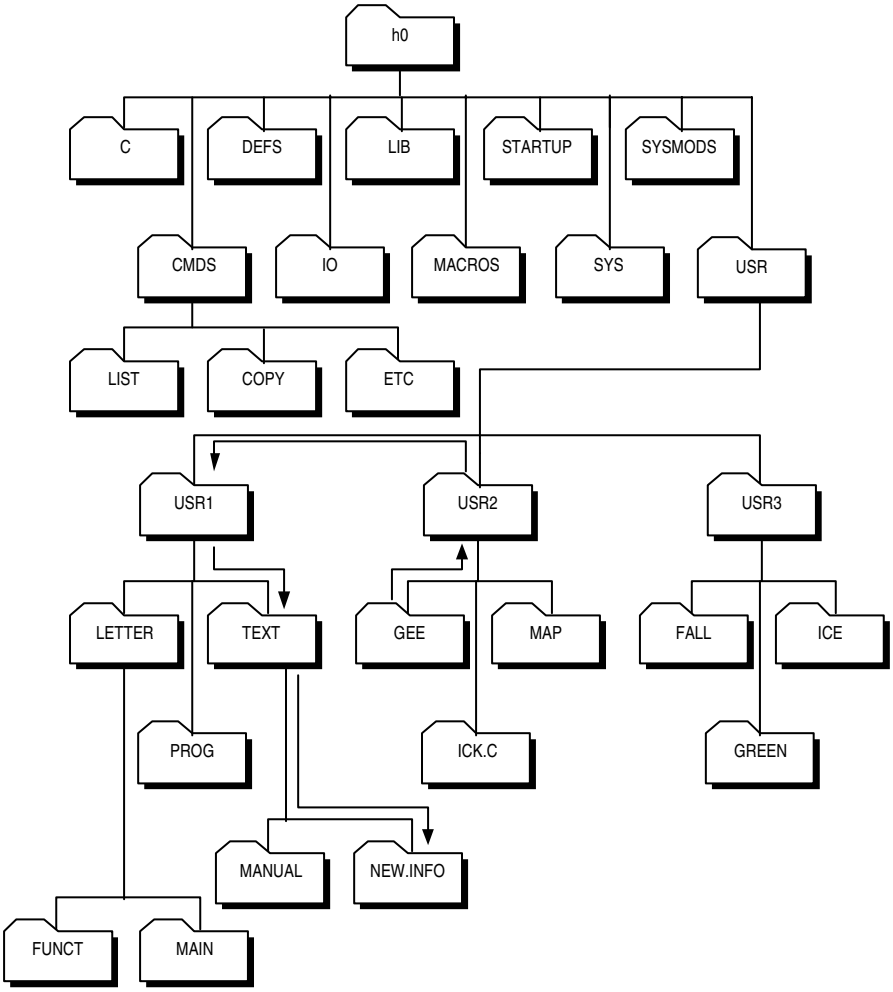
```
copy /h0/usr/usr2/gee /h0/usr/usr1/text/new.info
```

Assuming your data directory is `USR`, the following commands have the same effect:

```
copy /h0/usr/usr2/gee usr1/text/new.info
copy usr2/gee usr1/text/new.info
```

In **Figure 4-9**, `gee` is copied from `USR2/gee` to `USR1/TEXT/new.info` using the command `copy usr2/gee usr1/text/new.info`.

Figure 4-9 Copying Files



Copying a File into an Existing File

If you try to copy the contents of one file into an existing file, you receive `Error #000:218 Tried to create a file that already exists`. If you know the file exists but you want to overwrite it anyway, use the `-r` option. For example, the following command replaces the contents of `green` with the contents of `fall`.

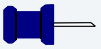
```
$ copy fall green -r
```

If you list the contents of both files, you see they are identical.

Copying Multiple Files

At some point, you may want to copy more than one file at a time into another directory. By using the `-w=<dir>` option of `copy`, you can copy more than one file with a single command. For example, if your current directory is `PROG` and you want to copy all of the files in `PROG` into the `TEXT` directory, you could type the following command line:

```
$ copy * -w=../text
```



Note

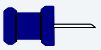
An asterisk is a wildcard. For more information about wildcards, refer to the section on wildcards in **Chapter 5: The Shell**.

This option prints the name of the file after each successful copy. If an error occurs, the prompt `continue (y/n)` is displayed.

Copying Large Files

If you have a large file, the copy procedure may be slow because the system has to perform multiple read and write statements. You can use the `-b` option to increase the buffer size. This would make the `copy` procedure faster for large files. To use the `-b` option, type `copy`, the original file name, the new file name, and `-b=<num>k`.

For example, typing `copy gee mine -b=20k` allocates a 20K buffer for copying the file `gee` into the file `mine`.



Note

`copy` uses a 4K memory buffer by default. This means only 4K of information is read from the original file and written to the new file at one time.

For More Information

For more information about `copy`, refer to the ***Utilities Reference*** manual.

You must have permission to copy the file. That is, you must be the owner of the file to be copied or the public read permission must be set in order to copy the file. You must also have permission to write in the directory you specify. In either case, if the copy procedure is successful, the new file has your group.user number unless you are the super user. If you are the super user, the new file has the same group.user number as the original file.

dsave: Using Procedure Files to Copy Files

Use the `dsave` utility to copy all files and directories within a specified directory by generating a procedure file. The procedure file is either executed later to actually perform the copy or, by specifying the `-e` option, executed immediately.

A procedure file is a special OS-9 file. It contains OS-9 commands. Each command is specified on a line, one command per line. When the procedure file is executed, the OS-9 commands it contains are executed in the order they are listed in the procedure file.

Note

To use the `dsave` utility, type `dsave` followed by the pathlist of the directory into which the files are copied, followed by any options you wish to use.

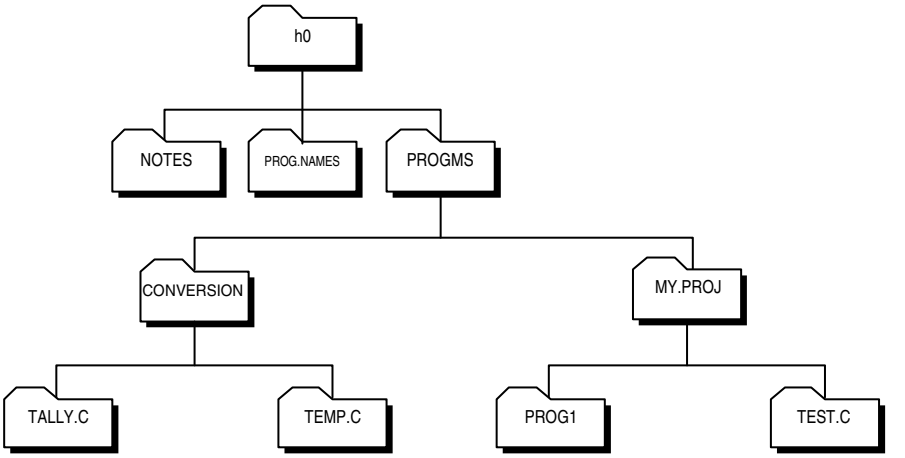
For More Information

For more information about procedure files, refer to **Chapter 5: The Shell**.

If no pathlist is specified for the destination, the files are copied to the current data directory when the procedure file is executed. If you do not specify the `-e` option or redirect the output to a file, `dsave` sends the output to the terminal.

The example below uses the following directory structure:

Figure 4-10 Dsave Example Directory Structure



If PROGMS is your current data directory and you type dsave ../notes, the following appears on your screen:

```
$ dsave ../notes
-t
chd ../notes
tmode -w=1 nopause
load copy
Makdir MY.PROJ
Chd MY.PROJ
Copy -b=10 /h0/PROGMS/MY.PROJ/prog1
Copy -b=10 /h0/PROGMS/MY.PROJ/test.c
Chd ..
Makdir CONVERSION
Chd CONVERSION
Copy -b=10 /h0/PROGMS/CONVERSION/temp.c
Copy -b=10 /h0/PROGMS/CONVERSION/tally.c
Chd ..
unlink copy
tmode -w=1 pause
$
```

Because the output was not redirected to a procedure file and the -e option was not used, the above commands were not executed. They were just echoed to your screen.

If you now type dsave ../notes -e, the commands are again echoed to the screen. However, the contents of the PROGMS directory are copied into the NOTES directory.

Copying Multiple Files

You can also redirect the output of dsave to a file. When you redirect the output, the commands that are output from dsave are essentially captured in a file. You can later execute this file to actually perform the dsave operation.

To redirect the output from dsave to a file, use the redirection modifier for standard output. The standard output modifier is the > symbol.

For example, from the PROGMS directory, you can redirect the output from dsave into a file called make.bckp by typing:

```
dsave >make.bckp
```

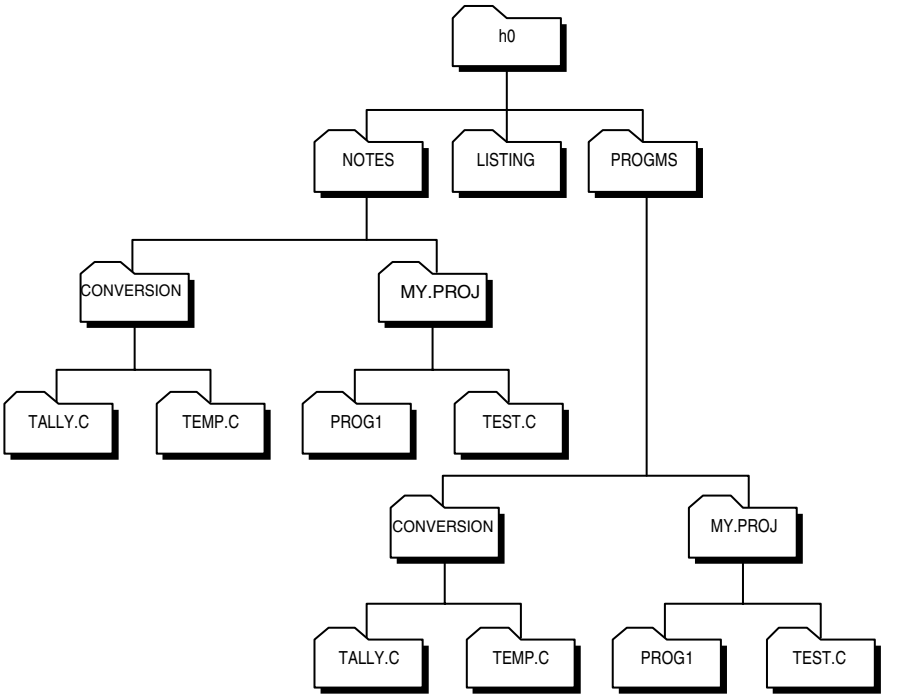
This command creates `make.bckp` in the current data directory. To perform the `dsave`, type `make.bckp` at the command line.

Redirecting the output to a file is helpful when you want to save most, but not all, of the files in the directory or directories being saved. You can edit `make.bckp` before performing the `dsave`. This allows you to save only selected files.

Regardless of how you decide to perform the `dsave`, if `dsave` encounters a directory file, it automatically creates a new directory and changes to that directory before generating `copy` commands for files in the subdirectory.

In the `dsave` example, the directory structure looks like the following after `dsave` has finished:

Figure 4-11 dsave Example Directory Structure



If the current working directory is the root directory of the disk, `dsave` creates a file that backups the entire disk, file by file. This is useful when you need to copy many files from different format disks or from a floppy disk or a hard disk.

Errors During dsave

If an error occurs during the `dsave` process, the following prompt is displayed:

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

Table 4-4 Responses During dsave

Response	Indicates you
`y`	Want to continue with `dsave`.
`n`	Do not want to continue with `dsave`.
`a`	Want all possible files copied and you do not want the prompt displayed on error.
`q`	Want to exit `dsave`.

You can use the `-s` option to turn off the prompt. This skips any file that cannot be copied and continues the `dsave` routine without the error prompt.

Indenting for Directory Levels

When you copy several subdirectories, you can use the `-i` option to indent for directory levels. This helps to keep track of which files are located in which directories.

Keeping Current Directory Backups

You can use `dsave` to keep current directory backups. Use the `-d` or `-d=<date>` options to compare the date of the file to be copied with a file of the same name in the directory where it is to be copied. The `-d` option copies any file with a more recent date. The `-d=<date>` option copies any file with a date more recent than that specified. The following example shows the use of `dsave` with the `-d` option:

```
$ chd /d0/BACKUP
$ dir
Directory of . 14:14:32
 Owner    Last Modified  Attributes Sector Bytecount Name
-------   -------------  ---------- ------ --------- ----
 12.4     92/11/12 1417   ------wr    20CO     11113 program.c
 12.4     92/10/05 1601   ------wr    313D      5744 prog.2
$ chd /d0/WORKFILES
$ dir
Directory of . 14:14:32
 Owner    Last Modified  Attributes Sector Bytecount Name
-------   -------------  ---------- ------ --------- ----
 12.4     92/11/12 1417   ------wr    DODO     11113 program.c
 12.4     92/11/12 1601   ------wr    3458      5780 prog.2
$ dsave -deb32 /d0/BACKUP
$ chd /d0/BACKUP
$ dir
Directory of . 14:14:32
 Owner    Last Modified  Attributes Sector Bytecount Name
-------   -------------  ---------- ------ --------- ----
 12.4     92/11/12 1417   ------wr    5990     11113 program.c
 12.4     92/11/12 1601   ------wr    A12B      5780 prog.2
```

Only `prog.2` was copied to the `BACKUP` directory because the date was more recent in the `WORKFILES` directory.

For More Information

For more information about `dsave`, refer to the ***Utilities Reference*** manual.

del and deldir: Deleting Files and Directories

Use the `del` and `deldir` utilities to eliminate unwanted files and directories:

- `del` deletes a file
- `deldir` deletes a directory

If you no longer need a file, deleting the file frees disk space. You *must* have permission to write to the file or directory in order to delete it.

Deleting Files

To delete a file, type `del`, followed by the name of the file you want deleted. For example, to delete the file `test` you created with `build`, you would type:

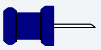
```
del test
```

If you execute `dir`, you see `test` is no longer displayed.

For More Information

For more information about wildcards, refer to **Chapter 5: The Shell**.

When deleting files, you may use wildcards. For example, if you have three files, `trial`, `trial1`, and `trial.c` in a directory and you want to use wildcards to delete `trial` and `trial1`, you may be tempted to type `del trial*`, but this would also delete `trial.c`, a file you want to keep.



Note

Use caution when you use wildcards with utilities like `del` and `deldir`. It is easy to unintentionally delete files you want to save.

The `del -p` option displays the following prompt before deleting a file:

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

Table 4-5 Responses When Deleting Files

Response	Action
`y`	Delete the file.
`n`	Do not delete the file.
`a`	Delete specified files without further prompts.
`q`	Exit the deleting process.

This helps prevent deleting files you want to keep.

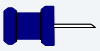
Deleting Directories

Deleting a directory is a little different. Use the `deldir` utility to delete directories. `deldir` first deletes all the files and directories in the given directory, and then, if no errors occur, finally deletes the directory name. For example:

```
$ deldir USER2
Deleting directory: USER2
Delete, List, or Quit (d, l, or q) ?
```

Table 4-6 Responses When Deleting a Directory

Response	Action
`d`	Delete the directory.
`l`	List the directory contents.
`q`	Quit without deleting any files.



Note

Never delete a file or directory unless you are sure you do not need it. Files and directories deleted with the `del` and `deldir` commands are permanently removed.

Chapter 5: The Shell

This chapter is a detailed description of the shell; the OS-9 user interface. It includes the following topics:

Shell Functions

The shell is the OS-9 command interpreter program. The shell translates the commands you enter into commands the operating system understands and executes. This allows you to use commands such as `dir`, `copy`, and `procs` without knowing the complex machine language OS-9 understands.

The shell also provides a user-configurable environment to personalize the way OS-9 works on your system. You can use the shell to change the shell prompt, send error messages to a file, or backup your disk before you log out.

The `shell` command starts the shell program. This command is automatically executed following system startup or after logging on to a timesharing terminal. When the shell is ready for commands, it displays the prompt:

```
$
```

This prompt indicates the shell is active and waiting for a command from your keyboard. You can now type a command line followed by a carriage return.

Shell Options

A number of options are available to the shell. By default, some are automatically turned on following startup or log on. The available shell options are:

Table 5-1 Shell Options

Option	Description
`-e=<file>`	Print error messages from `<file>`. If no file is specified, `/dd/sys/errmsg` is used. Without this option, the shell prints only error numbers with a brief message description. The ***OS-9 for 68K Technical Manual*** contains error code descriptions.
`-ne`	Print no error messages. This is the default option.
`-l`	The `logout` built-in command is required to terminate the login shell. `<eof>` does not cause the shell to terminate.
`-nl`	`<eof>` terminates the login shell. `<eof>` is normally caused by pressing the `<esc>` key. This is the default option.
`-p`	Display prompt. The default prompt is a dollar sign (`$`).
`-p=<string>`	Set the current shell prompt equal to `<string>`.
`-np`	Do not display the prompt.
`-t`	Echo input lines.
`-nt`	Do not echo input lines. This is the default option.

Table 5-1 Shell Options (continued)

Option	Description
`-v`	Verbose mode. Display a message for each directory searched when executing a command.
`-nv`	Turn off verbose mode. This is the default option.
`-x`	Abort process on error. This is the default option.
`-nx`	Do not abort process on error.

Changing Shell Options

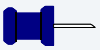
You can change shell options with either of two methods:

1. Type the option on the command line or after the `shell` command. For example:

`$ -np`	Turns off the shell prompt.
`$ shell -np`	Creates a new shell that does not prompt. When you exit the new shell, the original shell prompts.

2. Use `set`, a special shell command. To set shell options, type `set`, followed by the options desired. When using the `set` command, a hyphen (`-`) is unnecessary before the letter option. For example:

`$ set np`	Turns off the shell prompt.
`$ shell set np`	Creates a new shell that does not prompt. When you exit the new shell, the original shell prompts.



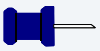
Note

The two methods described here accomplish the same function and are provided for your convenience. Use the method that is clearest to you.

The Shell Environment

The shell maintains a unique list of *environment* variables for each user on an OS-9 system. These variables affect the operation of the shell or other programs subsequently executed and can be set according to your preference.

You can access all environment variables by any process called by the environment's shell or by descendant shells. This essentially allows you to use the environment variables as *global* variables.



Note

If a subsequent shell redefines an environment variable, the variable is only redefined for that shell and its descendents. The environment variable is not redefined for the parent shell.

Environment variables are case sensitive. OS-9 does not recognize a variable if you do not use the proper case.

Four environment variables are automatically set up when you log on to a time-sharing system:

Table 5-2 Environment Variables Automatically Set Up

Environment Variable	Specifies
PORT	The name of the terminal. An example of a valid name is /t1. The tsmon utility automatically sets PORT.
HOME	Your *home* directory. The home directory is specified in your password file entry and is your current data directory when you first log on the system. This is also the directory used when you execute the command chd with no parameters.
SHELL	The first process executed when you log on to the system.
USER	The user name you typed when prompted by the login command.

On single user systems, you can set these variables with the setenv command. You can also set up a procedure file with your normal configuration of these variables. This procedure file could then be executed each time you startup your terminal.

For More Information

For more information about setenv, refer to the ***Utilities Reference*** manual.

There are four other important environment variables:

Table 5-3 Additional Environment Variables

Environment Variable	Specifies
PATH	Any number of directories. A colon (:) must separate directory paths. The shell uses PATH as a list of commands directories to search when executing a command. If the default commands directory does not include the file/module to execute, each directory specified by PATH is searched until the file/module is found or the list is exhausted.
PROMPT	The current prompt. By specifying an *at* sign (@) as part of your prompt, you may easily keep track of how many shells you have running under each other. @ is a replaceable macro for the shell level number. The environment variable _sh sets the base level.

Table 5-3 Additional Environment Variables (continued)

Environment Variable	Specifies

_sh	The base level for counting the number of shell levels. For example, set the shell prompt to `@howdy:` and `_sh` to `0`:
	`$ setenv _sh 0`
	`$ -p="@howdy: "`
	`howdy: shell`
	`1.howdy: shell`
	`2.howdy: eof`
	`1.howdy: eof`
	`howdy:`

TERM	The type of terminal. `TERM` allows word processors, screen editors, and other screen dependent programs to know what type of terminal configuration to use.

Changing the Shell Environment

Three commands are available to use with environment variables:

Table 5-4 Environment Variables Commands

Command	Description
`setenv`	Declare the variable and set the value of the variable.
`unsetenv`	Clear the value and remove the variable from storage.
`printenv`	Print the variables and their values to standard output.

setenv

`setenv` declares the variable and sets its value. The variable is put in an environment storage area accessed by the shell. For example:

```
$ setenv PATH ..:/h0/cmds:/d0/cmds:/dd/cmds
$ setenv _sh 0
```

These variables are only known to the shell in which they are defined and any descendant processes from that shell. This command does not change the environment of the parent process of the shell issuing `setenv`.

unsetenv

`unsetenv` clears the value of the variable and removes it from storage. For example:

```
$ unsetenv PATH
$ unsetenv _sh
```

printenv

`printenv` prints the variables and their values to standard output. For example:

```
$ printenv
PATH=..:/h0/cmds:/d0/cmds:/dd/cmds
PROMPT=howdy
_sh=0
```

For More Information

For more information about `setenv`, `unsetenv`, and `printenv`, refer to the ***Utilities Reference*** manual.

Built-In Shell Commands

The shell has a special set of commands, or option switches, built-in to the shell. You can execute these commands without loading a program and creating a new process. They are executable regardless of your current execution directory.

The built-in commands and their functions are:

Table 5-5 Built-in Shell Commands

Command	Description
`* <text>`	Indicate a comment: `<text>` is not processed. This is especially useful in procedure files.
`chd <path>`	Change the current data directory to the directory specified by the path.
`chx <path>`	Change the current execution directory to the directory specified by the path.
`ex <name>`	Directly execute the named program. This replaces the shell process with a new execution module.
`kill <proc ID>`	Abort the process specified by `<proc ID>`.
`logout`	Terminate the current shell. If the login shell is to be terminated, the `.logout` file in the home directory is executed and then the login shell is terminated.
`profile <path>`	Read input from a named file and then return to the shell's original input source.

Table 5-5 Built-in Shell Commands (continued)

Command	Description
`set <options>`	Set options for the shell.
`setenv <env var> <value>`	Set environment variable to the specified value.
`setpr <proc ID> <priority>`	Change the process' priority.
`unsetenv <env var>`	Delete an environment variable from the environment.
`w`	Wait for a child process to terminate.
`wait`	Wait for all child processes to terminate.

Shell Command Line Processing

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

Table 5-6 Command Line Parts

Command	Description
keyword	A name of a program, procedure file, built-in command, or pathlist.
parameters	**Optional:** The names of files, programs, values, variables, or constants to pass to the program being executed.
execution modifiers	**Optional:** These modify a program's execution by redirecting I/O or changing the priority or memory allocation of a process.
separators	**Optional:** When multiple commands are placed on the same command line, separators specify whether they should execute sequentially or concurrently.

After it identifies the keyword, the shell processes any execution modifiers and separators. The shell assumes any text not yet processed are parameters; they are passed to the program called.

The keyword must be the first word in the command line. If the keyword is a built-in command, it is immediately executed.

If the keyword is not a built-in command, the shell assumes it is a program name and attempts to locate it. The shell searches for the command in the following sequence:

1. The shell checks the memory to see if the program is already loaded into the module directory. If it is already in memory, there is no need to load another copy. The shell then calls the program to be executed.
2. If the program was not in memory, your current execution directory is searched. If it is found, the shell attempts to load the program. If this fails, the shell tries to execute it as a procedure file. If this fails, the shell attempts the same procedure using the next directory specified in the `PATH` environment variable. This continues until the command is successfully executed or the list of directories is exhausted.
3. The shell searches your current data directory. If it finds the specified file, it is processed as a procedure file. Procedure files are assumed to contain one or more shell command lines. These command lines are processed by a newly created, or *child*, shell as if they had been typed in manually. After all commands from the procedure file execute, control returns to the old, or *parent*, shell. Because the child shell processes the commands, all built-in commands in the procedure file such as `chd` and `chx` only affect the child shell.

The shell returns an error if the program is not found. If the program is found and executed, the shell waits until the program terminates. When the program terminates, it reports any errors returned. If there are more input lines, the shell gets the next line and the process is repeated.

This sample command line calls a program:

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

In this example:

`prog`	Is the keyword.
`#12K`	Is a modifier requesting an alternate memory size be assigned to this process. In this case, 12K is used as memory.

`sourcefile -l -j`	Are parameters passed to `prog`.
`>`	Is a modifier redirecting output to a file or device. In this case, `>` redirects the output to the printer (`/p`).
`/p`	Is the system's printer.

Special Command Line Features

In addition to basic command line processing, the shell facilitates:

- Memory allocation
- I/O redirection, including filters
- Process priority
- Wildcard pattern matching
- Multitasking: concurrent execution

These functions are accessed through execution modifiers, separators, and wildcards. There are virtually unlimited combinations of ways to use these capabilities.

Characters comprising execution modifiers, separators, and wildcards are stripped from the part(s) of the command line passed to a program as parameters. You cannot pass the following characters as parameters to programs unless you enclose them in quotes:

Table 5-7 Command Line Modifiers, Separators, and Wildcards

Name	Symbol	Description
Modifiers	#	Additional memory size
	^	Process priority
	>	Redirect output
	<	Redirect input
	>>	Redirect error output
Separators	;	Sequential execution
	&	Concurrent execution
	!	Pipe construction
Wildcards	*	Matches any character
	?	Matches a single character

Execution Modifiers

The shell processes execution modifiers before the program is run. If an error is detected in any of the modifiers, the run is aborted and the error reported.

Additional Memory Size Modifiers

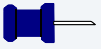
For More Information

For more information about module headers, refer to the ***OS-9 for 68K Technical Manual***.

Every executable program is converted to machine language for storage. During the conversion process, a *module header* is created for the program. A module header is part of all executable programs and holds information such as the program's name, size, and memory requirements.

When the shell processes an executable program, it allocates the minimum amount of working memory specified in the program's module header. To increase the default memory size, you can assign memory in 1K increments using the pound sign modifier (#), followed by a number of allocated kilobytes: `#10k` or `#10`. If the specified memory allocation is smaller than would otherwise be used, the modifier is ignored.

The increase in memory allocation only affects one command. If you want to increase the allocation for the next command, you must add the modifier (#) again.



Note

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

I/O Redirection Modifiers

Redirection modifiers redirect the program's standard I/O paths to alternate files or devices. Usually, programs do not use specific file or device names. This makes redirecting standard I/O to any file or device fairly simple without altering the program.

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

Standard Input Path	Normally passes data from a terminal's keyboard to a program.
Standard Output Path	Normally passes output data from a program to a terminal's display.
Standard Error Path	Can be used for either input or output, depending on the nature of the program using it. This path is commonly used to output routine status messages such as prompts and errors to the terminal's display. By default, the standard error path uses the same device as the standard output path.

A new process can only be created by an existing process. The new process is known as the *child process*. The process creating the child process is known as the *parent process*. Each child process inherits the parent process' standard I/O paths.

When the shell creates a new process, it inherits the shell's standard I/O paths. Upon startup or logging in, the shell's standard input is the terminal keyboard. The standard output and standard error are directed to the terminal's display. Consequently, the child's standard input is the terminal keyboard. The child's standard output and standard error are directed to the terminal's display.

When a redirection modifier is used on a shell command line, the shell opens the corresponding paths and passes them to the new process as its standard I/O paths.

The three redirection modifiers are:

Table 5-8 Redirection Modifiers

Name	Redirects the Standard
`<`	Input path
`>`	Output path
`>>`	Error path

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

Physical I/O Device Names

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

Table 5-9 Standard Device Names

Device	Description
`term`	Primary system terminal
`t1`, `t2`	Other serial terminals
`p`	Parallel printer
`p1`	Serial printer
`dd`	Default disk drive

Table 5-9 Standard Device Names (continued)

Device	Description
d0	Floppy disk drive unit 0
d1, d2	Other floppy disk drives
h0, h1	Hard disk drives (format-inhibited)
h0fmt, h1fmt	Hard disk drives (format-enabled)
n0, n1	Network devices
mt0, mt1	Tape devices
r0	RAM disk
pipe	Pipe device
nil	Null device

The h0fmt, h1fmt, etc. device descriptors have a bit set allowing you to use the format and os9gen utilities on them. To avoid accidentally formatting a hard disk, you should normally use the device names h0, h1, etc.

You may only use device names as the first name of a pathlist. The device name must be preceded by a slash (/) to indicate the name is an I/O device. If the device is not a mass storage multi-file device, such as a disk drive, the device name must be the only name in the path. This restriction is true for devices such as terminals and printers.

For example, you can redirect the standard output of list to write to the system printer instead of the terminal:

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

Using I/O Redirection Modifiers

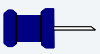
The shell automatically opens or creates, and closes (as appropriate) files referenced by I/O redirection modifiers. In the next example, the output of `dir` is redirected to the path `/d1/savelisting`:

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

If `list` is used on the path `/d1/savelisting`, output from `dir` is displayed as follows:

```
$ List /d1/savelisting
  directory of .   10:15:00
file1            myfile          savelisting
```

You can use redirection modifiers before and/or after the program's parameters, but you can use each modifier only once in a given command line. You can use redirection modifiers together to redirect more than one standard path. For example, `shell <>>>/t1` redirects all three standard paths to `/t1`.



Note

You may not place spaces between redirection operators and the device or file path.

You can use the addition and hyphen characters (`+` and `-`) with redirection modifiers:

- `>-` redirects output to a file. If the file already exists, the output overwrites it.
- `>+` adds the output to the end of the file.

The following example overwrites `dirfile` with output from the execution directory listing:

```
dir -x >-dirfile
```

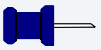
To add the listing of `newfile` to the end of `oldfile`, type:

```
list newfile >+oldfile
```

Process Priority Modifier

On multi-user systems or when multitasking, many processes seem to execute simultaneously. Actually, OS-9 uses a scheduling algorithm to allocate execution time to activate processes.

All active processes are sorted into a queue based on the *age* of the process.



Note

The age is a number between 0 and 65535 based on how long a process has waited for execution and its initial priority.

On a timesharing system, the system manager assigns the initial priority for processes started by each user. The password file contains the priority for the initial process. The initial process is usually the shell.

For More Information

Password files are covered later in this chapter.

On a single user system, processes have their priority set in the `Init` module.

All child processes inherit their parent process's priority.

When a process enters the active queue, it has an age set to its initial priority. Every time a new active process is submitted for execution, all earlier processes' ages are incremented. The process with the highest age executes first.

Raising the Process' Priority

If you want a program to run at a higher priority, use the caret modifier (^). By specifying a higher priority, a process is placed higher in the execution queue. For example:

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

In this example, the process `format` is assigned the priority of 255. By assigning a lower number, you can specify a lower priority.

WARNING

Specifying too high of a priority for a process can lock out all other processes until their ages mature. For example, if you specify a priority of 2000 for a large program and all the other processes have an age of less than 100, your program is the only process executed on the system until either your program terminates or another process' age reaches 2000. If another process' age reaches 2000, it runs once and is entered back in the queue at its initial priority. Once again, your program either runs until it terminates or until another process' age reaches 2000.

Wildcard Matching

The shell uses some alternate ways to identify file and directory names. It accepts wildcards in the command line. The two recognized wildcard characters are:

- An asterisk (`*`) matches any group of zero or more characters.
- A question mark (`?`) matches any single character.

The shell searches the current data directory or the directory given in a path for matching file names.

Examples found throughout this chapter use a directory containing the following files:

```
directory of FILES 14:45:20
diarydiary2 form form.backup
forms login.names logistics logs
old oldstuff setime.c shellfacts
sizes sizes.backup utils1
```

The Asterisk (*)

The command `list log*` lists the contents of `login.names`, `logistics`, and `logs`. The pattern `log*` matches all file names beginning with `log` followed by zero or more characters. The following commands demonstrate the function of this wildcard:

`list s*`	Lists all files in the current data directory beginning with `s`: `Shellfacts`, `setime.c`, and `sizes`.
`del *`	Deletes every file in the directory `FILES`.
`dir ../*.backup`	Lists all files in the parent directory ending with `.backup`.

The Question Mark (?)

The question mark (`?`) matches any single character in the wildcard's position. For example, the command line `list log?` only lists the contents of the file `logs`. The following commands demonstrate the function of this wildcard:

`del form?`	Deletes the file `forms` but not `form`.
`list s????`	Lists the contents of `sizes`, but not `setime.c` or `shellfacts`.

In both examples, the shell only searches for names with five characters.

Using Wildcards Together

You can also use wildcards together. For example, the command `list *.?` lists any files ending in a period followed by any letter, number, or special character, regardless of what comes before the period. In this case, `list *.?` lists the contents of the file `setime.c`.

Note

The shell disregards wildcard characters enclosed in double quotes. For example:

```
echo "*"
```

This echoes an asterisk (`*`) to standard output—usually the terminal. If you left out the double quotes around the asterisk, the shell would expand the wildcard to include every file name in the current directory and output each name to the terminal.

The shell only attempts to expand a character string containing a wildcard if the character string could be a pathlist. The shell does not expand wildcards used in the keyword of a command line. For example, the shell does not expand the asterisk in the following:

```
d* forms
```

WARNING

You must be careful when using wildcards with utilities such as `del` and `deldir`. You should not use wildcards with the `-x` or `-z` options of most utilities.

Command Separators

A single shell input line can include more than one command line. You can execute these command lines sequentially or concurrently:

Sequential Execution — Causes one program to complete its function and terminate before the next program is allowed to begin execution. Specify sequential execution with a semicolon (`;`).

Concurrent Execution — Allows several command lines to begin execution and run simultaneously. Specify concurrent execution with an ampersand (`&`)

Sequential Execution

When you enter one command per line from the keyboard, programs execute one after another (sequentially). All programs executed sequentially are individual processes created by the shell. After initiating execution of a program to be executed sequentially, the shell waits until the program it created terminates. The command line prompt does not return until the program has finished.

For example, the following command lines are executed sequentially. The `copy` command is executed first, followed by the `dir` command.

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

Specify more than one program on a single shell command line for sequential execution by separating each program name and its parameters from the next one with a semicolon (`;`). For example:

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

The shell first executes `copy` and then `dir`. No command line prompt appears between the execution of the `copy` and `dir` commands. The command line executes exactly as the previous two command lines, unless an error occurs.

If any program returns an error, subsequent commands on the same line are not executed regardless of the `-nx` option. In all other regards, a semicolon (`;`) and a carriage return act as identical separators.

Examples of Sequential Execution

To copy the contents of `oldfile` into `newfile`, delete `oldfile` when the `copy` command is finished, and then `list` the contents of `newfile`, type:

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

To redirect the output from `dir` into `myfile` in the `d1` directory, redirect the output from `list` to the printer, and delete `temp`, type:

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

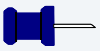
Concurrent Execution

Use the ampersand (`&`) separator to execute programs, including the shell, concurrently. The shell does not wait to complete a process before processing the next command. You use concurrent execution to start a background program.

For example:

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

The `dir`, `list`, and `copy` utilities run concurrently because they are separated by an ampersand (`&`). `del` does not run until `copy` terminates because sequential execution (`;`) was specified.



Note

Use the concurrent execution separator (`&`) for multitasking. The number of programs that can run at the same time depends on the amount of free memory in the system and each program's memory requirements.

For More Information

If you have several processes running at once, you can display a *status summary* of all your processes with the `procs` utility. `procs` lists your current processes and pertinent information about each process.

The `procs` utility is covered later in this chapter.

If you add an ampersand (`&`) to the end of a command line, regardless of the type of execution specified, the shell immediately:

1. Returns command to the keyboard.
2. Displays the `$` prompt.
3. Waits for a new command.

This frees you from waiting for a process or sequence of processes to terminate. This is especially useful when making a listing of a long text file on a printer. Instead of waiting for the listing to print to completion, you can use the concurrent execution separator to use your time more efficiently.

Pipes and Filters

A third kind of separator is the exclamation point (`!`) used to construct *pipelines*. Pipelines consist of two or more concurrent programs whose standard input and/or output paths connect to each other using *pipes*.

A pipe is simply a way to connect the output of a process to the input of another process, so the two run as a sequence of processes: a pipeline. Pipes are one of the primary means to transfer data from process to process for interprocess communications. Pipes are first-in, first-out buffers and may hold up to 90 bytes of data at a time.

All programs in a pipeline execute 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 ensures data cannot flow through a pipeline any faster than the slowest program can process it.

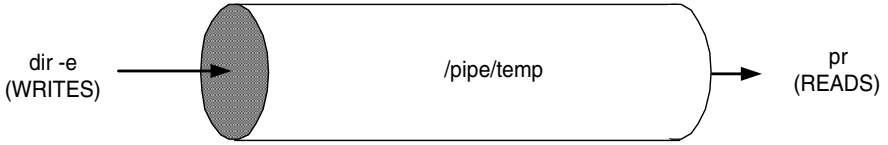
Any program that reads data from standard input can read from a pipe. Any program that writes data to standard output can write data to a pipe. Several utilities are designed so you can pipe the standard output of one to the standard input of another. For example:

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

This example pipes the standard output of `dir` to the standard input of the `pr` utility instead of on the terminal screen. `pr` reads the output of `dir` even though `pr` reads standard input by default. `pr` then displays the result.

As **Figure 5-1** shows, the standard output of the `dir -e` command is piped to the standard input of the `pr` command through an un-named pipe. The `pr` utility displays the results of the `dir -e` command.

Figure 5-1 Example Pipe



OS-9 provides two types of pipes:

- **Named Pipes**
- **Un-named Pipes**

Un-named Pipes

The shell creates un-named pipes when it processes an input line with one or more exclamation point (`!`) separators. For each exclamation point, the standard output of the program named to the left of the

exclamation point is redirected by a pipe to the standard input of the program named to the right of the exclamation point. Individual pipes are created for each exclamation point present. For example:

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

In this example, the input for the program `update` is redirected from `master_file`. `update`'s standard output becomes the standard input for the program `sort`. `sort`'s output, in turn, becomes the standard input for the program `write_report`. `write_report`'s standard output is redirected to the printer.

Named Pipes

Named pipes are similar to un-named pipes with one exception: a named pipe works as a holding buffer that another process can open at a different time.

To create a named pipe, redirect output to `/pipe/<file>`, where `<file>` is any legal OS-9 file name. For example:

```
$ list letters >/pipe/letters
```

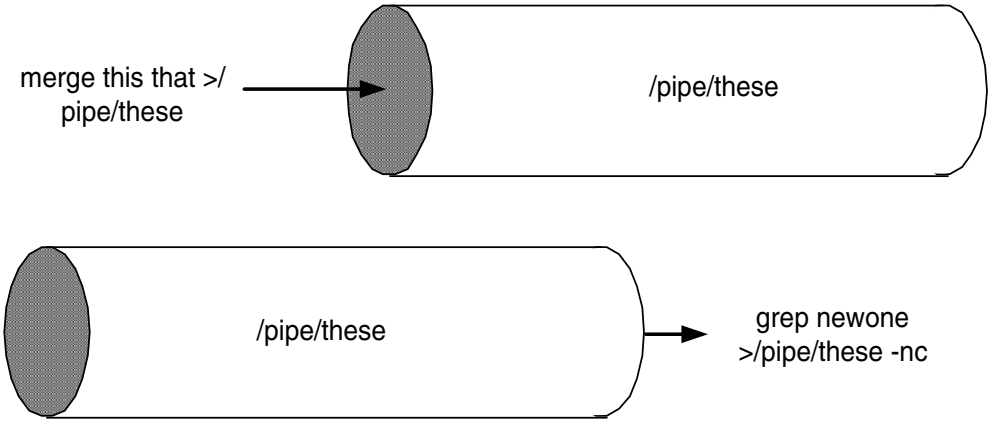
The output from the `list` command is redirected into a named pipe, `/pipe/letters`. The information remains in the pipe until it is listed, copied, deleted, or used in some other manner.

As **Figure 5-2** shows, the output from the `merge` command is redirected to the named pipe, `/pipe/these`. `/pipe/these` remains open until the contents are used in some way. In this example, another user could later `grep` for a word in the file `/pipe/these`:

```
grep newone /pipe/these -nc
```

Once the file has been used, the named pipe is deleted.

Figure 5-2 Named Pipes



You can also create named pipes by writing to the named pipe from a program. Named pipes are similar to mass-storage files. Named pipes have attributes and owners. They may be deleted, copied, or listed using the same syntax one would use to delete, copy, or list a file. You may change the attributes of a named pipe just as you would change the attributes of a file.

`dir` works with `/pipe`. This displays all named pipes in existence. A `dir -e` command may be deceiving. If any utility other than `copy` creates a named pipe, the pipe size equals 90 bytes. `copy` expands the size of the pipe to the size of the file. This indicates the first 90 bytes of the output are in the named pipe. However, if the `procs` utility is executed, you see a path remains open to `/pipe`. If you were to `copy` or `list` the pipe, for example, the pipe would continue to receive input and pass it to its output path until the input process is finished. When the pipe is empty, the named pipe is deleted automatically.

Some of the most useful applications of pipelines are character set conversion, data compression/decompression, and text file formatting. Programs designed to process data as components of a pipeline are often called *filters*.

Command Grouping

You can enclose sections of shell input lines in parentheses ( ). This allows you to apply modifiers and separators to an entire set of programs. The shell processes them by calling itself recursively as a new process to execute the enclosed program list. For example, the following commands produce the same result:

```
$ (dir /d0; dir /d1) >/p
$ dir /d0 >/p; dir /d1 >/p
$ dir& (procs; del it)
```

It is important to remember OS-9 processes commands from left to right. In the following example, the `dir` command executes before the `procs` and `del` commands located inside the parentheses.

However, one subtle difference exists. The printer is continuously controlled by one user in the first example, while in the second case, another user could conceivably use the printer in between the `dir` commands.

You can use command grouping to execute a group of programs sequentially with respect to each other and concurrently with respect to the shell initiating them. For example:

```
$ (del *.backup; list stuff_* >p)&
```

This command begins to sequentially delete all files ending in `.backup` and then list to the printer the contents of any files starting with `stuff_`. At the same time, a `$` prompt appears, indicating the shell is waiting for a new command.

Command Grouping and Pipelines

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

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

This command line outputs the `dir` listings of `CMDS` and `SYS`, in that order, through a pipe to the program `makeuppercase`. The total output from `makeuppercase` is then piped to the program `transmit`.

Shell Procedure Files

A procedure file is a text file containing one or more command lines that are identical to command lines manually entered from the keyboard. The shell executes each command line in the exact sequence given in the procedure file.

A simple procedure file could consist of `dir` on one line and `date` on another. When the name of this procedure file is entered from the command line, OS-9 runs `dir`, followed by `date`.

Procedure files have a number of valuable applications. They can:

- Eliminate repetitive manual entry of commonly used command sequences.
- Allow the computer to execute a lengthy series of programs in the background unattended or while you are running other programs in the foreground.
- Initialize your environment when you first log in.

In addition, you can use a procedure file to redirect the standard input, standard output, and standard error paths from programs and utilities to procedure files. This has many useful purposes. For example, instead of receiving the sometimes annoying output of shell messages to your terminal at random times, you could redirect the shell's output to a procedure file and review the messages at a more convenient time.

You can also run procedure files in the background by adding the ampersand (`&`) operator:

```
$ procfile&
+4
```

WARNING

If a procedure file is run in the background, it should not contain any terminal I/O. Any terminal I/O caused by a background procedure file minimally causes confusion as two or more processes try to control the same I/O path.

Notice the `+4` returned by the shell in the example above. This is the process number assigned to the shell running `procfile`. You could achieve the same effect with the `<control>c` interrupt:

```
$ procfile
[<control>C is typed]
+4
```

Using `<control>c` to place a procedure in the background only works if the procedure has not yet performed I/O to the terminal. Another limitation of the `<control>c` interrupt occurs when the shell has not had time to set up the command for execution. If the shell has not loaded files from the disk or established pipelines, the `<control>c` causes the shell to abort the operation and return the shell prompt. For this reason, you should usually use the ampersand to place a procedure in the background.

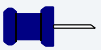
OS-9 does not have any limit on the number of procedure files that can execute simultaneously, as long as memory is available.

Note

Procedure files themselves can cause sequential or concurrent execution of additional procedure files.

The Login Shell and Two Special Procedure Files: .login and .logout

The *login shell* is the initial shell created by the login sequence to process the user input commands after logging in.



Note

The `.login` and `.logout` procedure files provide a way to execute desired commands when logging on to and leaving the system. To make use of these files, they must be located in the home directory.

The .login File

`.login` is executed each time the login command is executed. This allows you to run a number of initializing commands without remembering each and every command. The login shell processes `.login` as a command file immediately after successfully logging on to a system. After processing all commands in the `.login` file, the shell prompts you for more commands. The main difference in handling `.login` is the login shell itself actually executes the commands rather than creating another shell to execute the commands.

You can issue commands such as `set` and `setenv` within `.login` and have them affect the login shell. This is especially useful for setting up the environment variables `PATH`, `PROMPT`, `TERM`, and `_sh`.

Here is an example `.login` file:

```
setenv PATH ..:/h0/cmds:/d0/cmds:/dd/cmds:/h0/doc/spex
setenv PROMPT "@what next: "
setenv _sh 0
setenv TERM abm85h
querymail
date
dir
```

The .logout File

You execute `logout` to exit the login shell and leave the system. `.logout` is processed before the login shell terminates. It only processes the `.logout` file when given to the login shell; other subsequent shells simply terminate. You could use `.logout` to execute any clean up procedures performed on a regular schedule. This might be anything from instigating a backup procedure of some sort to printing a reminder of things to do. Here is an example `.logout` file:

```
procs
wait
echo "all processes terminated"
* basic program to instigate backup if necessary *
disk_backup
echo "backup complete"
```

The Profile Command

You can use the `profile` built-in shell command to cause the current shell to read its input from the named file and then return to its original input source—usually the keyboard. To use the `profile` command, enter `profile` and the name of a file:

```
profile setmyenviron
```

The specified file (in this case `setmyenviron`) may contain any utility or shell commands, including commands to set or unset environment variables or to change directories. These changes remain in effect after the command has finished executing. This is in contrast to calling a normal procedure file by name only. If you call a normal procedure file without using the `profile` command, the changes would not affect the environment of the calling shell.

You can nest `profile` commands. That is, the file itself may contain a `profile` command for another file. When the latter `profile` command is completed, the first one resumes.

A particularly useful application for `profile` files is within a user's `.login` and `.logout` files. For example, if each user includes the following line in the `.login` file, then system-wide commands (such as common environments and news bulletins) can be included in the file `/dd/SYS/login_sys`:

```
profile /dd/SYS/login_sys
```

You can use a similar technique for `.logout` files.

The Startup Procedure File

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. This procedure file initiates the timesharing monitor for each terminal. It begins by starting the system clock and initiating concurrent execution of a number of processes having their I/O redirected to each timesharing terminal.

`tsmon` is a special program that monitors a terminal for activity. Typically, `tsmon` is executed as part of the start-up procedure when the system is first brought up and remains active until the system shuts down.

For More Information

For more information about `tsmon`, refer to the ***Utilities Reference*** manual.

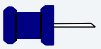
`tsmon` is normally used to monitor I/O devices capable of bi-directional communication, such as CRT terminals. However, you can also use `tsmon` to monitor a named pipe. If you do this, `tsmon` creates the named pipe and then waits for some other process to write data to it.

You can run several `tsmon` processes concurrently, each one watching a different group of devices. Because `tsmon` can monitor up to 28 device name pathlists, you must run multiple `tsmon` processes when you need more than 28 devices monitored. Multiple `tsmon` processes can be useful for other reasons. For example, you may want to keep modems or terminals suspected of hardware trouble isolated from other devices in the system.

Here is a sample procedure file for a timesharing system with terminals named `T1`, `T2`, `T3`, and `T71`:

```
* system startup procedure file
echo Please Enter the Date and Time
setime
```

```
tsmon /t1 /t2 /t3&
tsmon /t71* This terminal has been misbehaving
```



Note

This `login` procedure requires a file called `password`, with the appropriate entries, exists in the `SYS` directory of the system's initial device.

The Password File

A password file is found in the `SYS` directory. Each line in the password file is a login entry for a user. The line has several fields, each separated by a comma. The fields are:

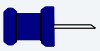
User name	The user name may contain up to 32 characters including spaces. If this field is empty, any name matches.
Password	The password may contain a maximum of 32 characters including spaces. If this field is omitted, no password is required for the specified user.

For More Information

For more information about password files, refer to the login description in the ***Utilities Reference*** manual.

Group.user ID number	Both the group and the user portion of this number may be from 0 to 65535. 0.0 is the super user. The file security system uses this number 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.
Initial process priority	This number may be from 1 to 65535. It indicates the priority of the initial process.
Initial execution directory	This field is usually set to `/h0/CMDS`. Specifying a period (`.`) for this field defaults the initial execution directory to the `CMDS` file.
Initial data directory	This is usually the specific user directory. Specifying a period (`.`) for this directory defaults to the current directory.
Initial Program	This field contains the name and parameters of the program to be initially executed. This is usually `shell`.



Note

Fields left empty are indicated by two consecutive commas.

The following is a sample password file:

```
superuser,secret,0.0,255,.,.,shell -p="@howdy"
suzy,morning,1.5,128,.,/d0/SUZY,shell
don,dragon,3.10,100,.,/d0/DON,Basic
```

Creating a Temporary Procedure File

You can create temporary procedure files to perform tasks requiring a sequence of commands. The `cfp` utility creates a temporary procedure file in the current data directory and calls the shell to execute it. After the task is complete, `cfp` automatically deletes the procedure file, unless you use the `-nd` option to specify you do not want the procedure file deleted.

The following is the syntax for the `cfp` utility:

```
cfp [<opts>] [<path1>] {<path2>} [<opts>]
```

To use the `cfp` utility, type `cfp`, the name of the procedure file (`<path1>`), and the file(s) (`<path2>`) used by the procedure file. If you use the `-s=<string>` option, you may omit the name of the procedure file.

For More Information

For more information about `cfp`, refer to the ***Utilities Reference*** manual.

All occurrences of an asterisk (`*`) in the procedure file are replaced by the given pathlist(s) unless preceded by the tilde character (`~`). For example, `~*` translates to `*`. The command procedure is not executed until all input files have been read.

For example, if you have a procedure file in your current data directory called `copyit` consisting of a single command line: `copy *`, you could put all of your C programs from two directories, `PROGMS` and `MISC.JUNK`, into your current data directory by typing:

```
$ cfp copyit ../progms/*.c ../misc.junk/*.c
```

If you do not have a procedure file, you can use the `-s` option. The `-s` option causes `cfp` to read the string surrounded by quotes instead of a procedure file. For example:

```
$ cfp -s="copy *" ../progms/*.c ../misc.junk/*.c
```

In this case, `cfp` creates a temporary procedure file to copy every file ending in `.c` in both `PROGMS` and `MISC.JUNK` to the current data directory. The procedure file created by `cfp` is deleted when all the files have been copied.

Using the `-s` option is convenient because you do not have to edit the procedure file if you want to change the copy procedure. For example, if you are copying large C programs, you may want to increase the memory allocation to speed up the process. You could allocate the additional memory on the `cfp` command line:

```
$ cfp "-s=copy -b100 *" ../progms/*.c ../misc.junk/*.c
```

Reading the File Names from Standard Input or a File

You can use the `-z` and `-z=<file>` options to read the file names from either standard input or a file. The `-z` option is used to read the file names from standard input. For example, if you have a procedure file called `count.em` containing the command `count -l *` and you want to count the lines in each program to see how large the programs are before you copy them, you could type the following command line:

```
$ cfp -z count.em
```

The command line prompt does not appear because `cfp` is waiting for input. Type in the file names on separate command lines. For example:

```
$ cfp -z count.em
../progms/*.c
../misc.junk/*.c
```

When you have finished typing the file names, press the carriage return a second time to get the shell prompt.

If you have a file containing a list of the files you want copied, you could type:

```
$ cfp -z=files "-s=copy *"
```

Multiple Shells

Like all OS-9 utilities, the shell can be simultaneously executed by more than one process. This means in addition to each user having their own shell, an individual user can have multiple shells.

You can use procedure files to create new shells. For example, to execute a shell whose standard input is obtained from `procfile`, type:

```
$ shell <procfile
```

The new shell automatically accepts and executes the command lines from the procedure file instead of a terminal keyboard. This technique is sometimes called *batch* processing.

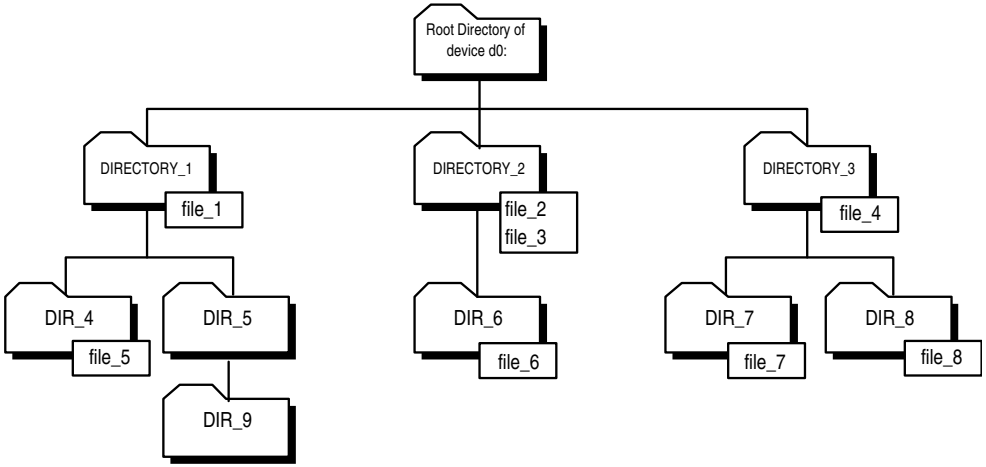
Shells can also fork new shells by simply processing the procedure file:

```
$ procfile
```

Basically, both of the above commands execute the commands found in the `procfile` file.

By creating new shells, you can also move around the file system more efficiently. To demonstrate this concept, the directory system in **Figure 5-3** is used.

Figure 5-3 An Example Directory



If your current data directory is DIR_9 and you want to work on file_8, you would normally change your current data directory to DIR_8 and access the file by typing:

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

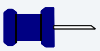
To return to DIR_9, you would execute a similar command. This is somewhat inconvenient and involves always knowing the path to each directory.

Instead, you can create a shell and change directories:

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

This makes your current directory DIR_8, but you can return to DIR_9 by pressing the <escape> (esc) key. By this method, you may use any directory as a base directory and *fork* a shell out to any other directory.

You may continue to embed as many shells as you like. Each time you press the <escape> key, you are taken to the previous shell. In this fashion you could conceivably escape from DIRECTORY_2 to DIR_8 to DIR_6 to DIR_9.



Note

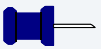
Because of the nature of jumping from shell to shell, it is easy to get lost. pd displays a complete pathlist from the root directory to your current data directory. Likewise, when running multiple shells, it is easy to forget how many shells are running. If the _sh environment variable is set to 1 and the shell prompt includes an *at* sign (@), the number of shells replaces the @ in the prompt. For example, if three shells are run under each other, the prompt might look like this:

```
3.what next:
```

You should experiment with the multiple shell aspects to fully use OS-9.

The Procs Utility

Because of OS-9's multitasking abilities, you often have more than one process executing at a time. You may become confused as to which processes are still running and which processes have run to completion. The `procs` utility displays a list of processes running on the system you own. This allows you to keep track of your current processes.



Note

Processes can switch states rapidly, usually many times per second. Therefore, the `procs` display is a snapshot taken at the instant the command is executed and shows only those processes running at that exact moment.

`procs` displays the following information for each process:

Table 5-10 Information Displayed By procs

Name	Description
`Id`	The process ID.
`PId`	The parent process ID.
`Grp.usr`	The group and user number of the process owner.
`Prior`	The initial priority of the process.
`MemSiz`	The amount of memory the process is using.
`Sig`	The number of any pending signals for the process.

Table 5-10 Information Displayed By procs (continued)

Name	Description
S	State of the process *CPU = Process is currently in the CPU. This will always be the procs command since it has to be running when it takes the snapshot of the process table. a = Active. Process wants CPU time, but is having to wait because another process is in the CPU already. d = Debug. Process is currently being debugged. e = Event. Process is blocked waiting on an event. p = Semaphore. Process is blocked waiting on a semaphore. s = Sleeping. Process is blocked waiting on a signal or time value to elapse. w = Waiting. Process is blocked waiting on a child process to terminate. - = Zombie. Process has been terminated, but the parent has not performed a wait to read the exit status.
CPU Time	The amount of CPU time the process has used.
Age	The elapsed time since the process started.
Module & I/O	The process name and standard I/O paths: < Standard input > Standard output >> Standard error output If several of the paths point to the same pathlist, the identifiers for the paths are merged.

The following is an example of `procs`:

```
$ procs
Id PId Grp.Usr Prior MemSiz Sig S   CPU Time   Age Module & I/O
 2   1   0.0   128    0.25k   0 w       0.01   ??? sysgo  <>>>term
 3   2   0.0   128    4.75k   0 w       4.11 01:13 shell  <>>>term
 4   3   0.0     5    4.00k   0 a   12:42.06 00:14 xhog   <>>>term
 5   3   0.0   128    8.50k   0 *       0.08 00:00 procs   <>term
 6   0   0.0   128    4.00k   0 s       0.02 01:12 tsmon  <>>>t1
 7   0   0.0   128    4.00k   0 s       0.01 01:12 tsmon  <>>>t2
```

`procs -a` displays nine pieces of information: the process ID, the parent process ID, the process name and standard I/O paths, and six new pieces of information:

Table 5-11 Information Displayed by procs -a

Name	Description
`Aging`	The age of the process based on the initial priority and how long it has waited for processing.
`F$calls`	The number of service request calls made.
`I$calls`	The number of I/O requests made.
`Last`	The last system call made.
`Read`	The number of bytes read.
`Written`	The number of bytes written.

The following is an example of `procs -a`:

```
$ procs -a
Id PId  Aging  F$calls I$calls Last      Read Written Module & I/O
 2   1   129        5       1 Wait         0       0 sysgo <>>>term
 3   2   132      116     127 Wait       282     129 shell <>>>term
 4   3    11        1       0 TLink        0       0 xhog  <>>>term
 5   3   128        7       4 GPrDsc       0       0 procs <>>>term
 6   0   130        2       7 ReadLn       0       0 tsmon <>>>t1
 7   0   129        2       7 ReadLn       0       0 tsmon <>>>t2
```

The `-b` option displays all information from `procs` and `procs -a`. The `-e` option displays information for all processes in the system.

For More Information

For more information on `procs`, see the ***Utilities Reference*** manual.

Waiting for the Background Procedures

If you use OS-9's multitasking ability, a number of procedures may be running in the background. If it is important to wait for these tasks to finish before running a new procedure, use the `w` or `wait` built-in shell command.

Note

`w` waits for a child process to be executed to finish.

`wait` waits for all child processes running in the background to finish.

REMINDER: A child process is a process that the current shell or a child of the shell is executing.

For example, if you need to create a document from three different files and each file has to be sorted by different fields, you can use the following procedure files to create the same result:

```
*start of first procedure file*
qsort -f=1 file1&
qsort -f=2 file2&
qsort -f=3 file3&
wait
merge file1 file2 file3 >report
*start of second procedure file*
```

```
qsort -f=1 file1
qsort -f=2 file2
qsort -f=3 file3
merge file1 file2 file3 >report
```

The first procedure file is much quicker because each of the files are processed concurrently.

Stopping Procedures

You can use two methods to stop a procedure:

1. The <control>c or <control>e signals.
2. The kill utility.

The shell handles the keyboard generated signals in the following manner. If OS-9 receives either of these signals while the shell is waiting for keyboard input, it issues the following messages:

```
$ Read I/O error - Error #000:002    [ ^E typed ]
$ Read I/O error - Error #000:003    [ ^C typed ]
```

These are the standard messages given whenever an I/O error occurs when reading command input data. These keyboard signals are useful to get the shell's attention while it is waiting for a process to terminate.

If the shell is waiting for keyboard input and <control>e is typed, the shell forwards the keyboard abort signal to the current process and immediately prompts for command input:

```
$ sleep 500
[ ^E is typed]
abort
$
```

The shell uses the abort message to acknowledge the interrupt's receipt.

If the shell is waiting for keyboard input and <control>c is typed, the shell stops waiting for the current process to terminate and prompts for command input. This action is similar to using an ampersand (&) on the command line. For example:

```
$ sleep 500
[ ^C is typed]
8
$
```

You must remember you can only use <control>c in this fashion if the command in question has not yet performed I/O to the terminal. The signal is only received by the last process to perform I/O. If the shell has not yet finished setting up the command for execution, the signal causes the shell to abort the operation and return the prompt.

Note

<control>c Stops the shell from waiting for the process to terminate and returns a prompt for a command.

<control>e Forwards the keyboard abort signal to the process and immediately prompts for input.

You can also use the kill utility to terminate background processes by specifying the process number of the process to kill. Obtain the process number of the process to kill from procs. kill is used in the following manner:

```
kill <proc num>
```

For example, if you want to terminate a process called xhog:

Step 1. Execute a procs:

```
$ procs
Id PId Grp.Usr Prior MemSiz Sig S   CPU Time   Age Module & I/O
 3   2   7.03   128    4.75k   0 w       4.11 01:13 shell <>>>term
 4   3   7.03     5    4.00k   0 a   12:42.06 00:14 xhog <>>>term
 5   3   7.03   128    8.50k   0 *       0.08 00:00 procs <>>term
```

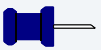
From procs, you can see the process number for xhog is 4.

Step 2. Type:

```
$ kill 4
```

When you execute `procs` again, `xhog` is no longer shown.

Either of these methods terminate any process running in the background with one exception: if a process is waiting for I/O, it may not die until the current I/O operation is complete. Therefore, if you terminate a process and `procs` shows it still exists, it is probably waiting for the output buffer to be flushed before it can die.



Note

You must either own the procedure or be the super user to kill a specified process.

Error Reporting

Many programs, including the shell, use OS-9's standard error reporting function. This displays a brief description of the error and an error number on the standard error path. An appendix listing all of the standard error codes is included with this manual.

If a longer description of errors is desired, set the `-e` shell option. This prints error messages from `/dd/SYS/errmsg` on standard output.

Running Compiled Intermediate Code Programs

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

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

```
$ BASIC adventure
$ adventure
```

Chapter 6: The make Utility

This chapter explains the `make` utility in detail. This utility is used to maintain and regenerate software from a group of files.

This chapter includes the following topics:

Introduction

The `make` utility simplifies multi-source-file project maintenance by determining what source files need to be recompiled. `make` examines the dates associated with the target file and its dependents and regenerates any out-of-date dependents and then the target. A special procedure file (generally named `makefile`) is used to specify the target file and other dependencies involved with recreating the target file.

Note

It is important for you to know two terms:

- The *target file* is the final product of one or more compiled source files.
- *Dependents* are the files that make up a target file.

This chapter contains sections about:

- Makefile entries
- How a makefile is processed
- Implicit dependencies
- Macro recognition
- `make` generated command lines
- `make` options

The chapter concludes with some examples of how you might use `make`.

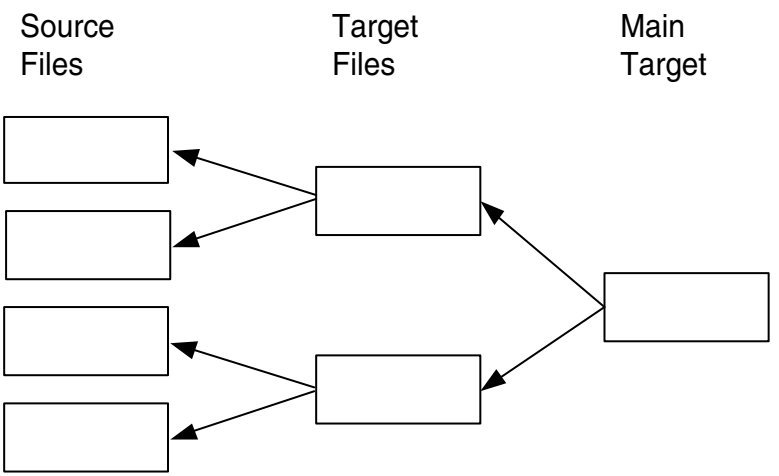
The make Utility

The `make` utility automates the process of maintaining and re-creating your target file. The executable target file is the final product of one or more compiled source files. It may depend on other files for information and instructions. These other files are known as dependents. If you update dependency files, the target file becomes out-of-date.

A *makefile* is a special type of procedure file describing the relationship between the final product and the files that make up the final product. `make` uses the makefile to:

- Create a list of file dependencies and resolves implicit dependencies (refer to the **Implicit Dependencies** section).
- Compare each file's date to the main target's date. Files with dates equal to or after the main target's date are considered changed.
- Recompile and relink the changed files to update the executable main target file.

Figure 6-1 The Make Process



A makefile can contain any of the following types of entries:

- **Dependency Entry**
- **Command Entry**
- **Comment Entry**
- **Include Entry**
- **Macro Entry**

Each type of entry is described below.

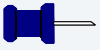
Dependency Entry

A makefile *dependency entry* specifies the relationship of a target file and the dependents used to build the target file. Dependency entries have the following syntax:

```
<target>: [<dependent>] {[<dependent>]}
```

For example, in the following dependency entry, `program` is the target, and its dependents are two files: `xxx.r` and `yyy.r`.

```
program: xxx.r yyy.r
```



Note

Syntax notes:

`[ ]` = Enclosed items are optional.

`{ }` = Enclosed items may be used 0, 1, or many times.

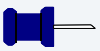
`< >` = Enclosed item is a description of the parameter to use.

The list of files following the target file is known as the *dependency list*. You can list any number of space separated dependents in the dependency list and any number of dependency entries in a makefile. A

dependent in one entry may also be a target file in another entry. There is, however, only one main target file in each makefile. The main target file is by default specified in the first dependency entry in the makefile.

For More Information

Refer to the **Implicit Dependencies** section for an explanation of what happens when the main target file is not a program to compile.



Note

To continue any makefile entry on the following line, place a space followed by a backslash (\) at the end of the line to continue. You must continue all entries longer than 256 characters on another line.

Each continuation line must adhere to the rules for its type of entry. For example, if a command line is continued on a second line, the second line must begin with a space or a tab:

```
FILE: aaa.r bbb.r ccc.r ddd.r eee.r \

fff.r ggg.r
            touch aaa.r bbb.r ccc.r \
            ddd.r eee.r fff.r ggg.r
```

`make` ignores spaces and tabs preceding non-command continuation lines.

Command Entry

A makefile *command entry* specifies the command you must execute to update, if necessary, a particular target file. `make` updates a target file only if its dependents are newer than itself. If no instructions for update are provided, `make` creates a command entry to perform the operation. The command entry created depends on the mode (`compat`, `c89`, or `ucc`).

For More Information

Refer to the **Implicit Dependencies** section of this chapter for more information about modes and generated command entries.

`make` recognizes a command entry by a line beginning with one or more spaces or tabs. Any legal OS-9 command line is acceptable. You can give more than one command entry for any dependency entry. Each command entry line is assumed to be complete unless it is continued from the previous command with a backslash (\). Do not intersperse comments with commands.

An example of command entry line syntax is:

```
<target>:[[<file>],<file>]
<OS-9 command line>
<OS-9 command line>\
<continued command line>
```

In the following example, the first line is the dependency entry and the second line is the command entry:

```
program: xxx.r yyy.r
cc xxx.r yyy.r -f=program
```

Comment Entry

A makefile *comment entry* is any line beginning with an asterisk (*) or a pound sign (#). In addition, all characters following a pound sign within another makefile line are considered comments. The one exception to this is when a pound sign is followed by a digit (0-9). In this case, it is considered a stack space command line modifier. All blank lines are ignored. For example:

```
<macro name> = <value>
<target> : [<file>] {[<file>]}
* the following commands are executed if a
* dependent file is newer than the target file
<OS-9 command line>    # this is also a comment
```

Include Entry

A makefile *include entry* tells `make` to use a file that has entries common to more than one makefile. `make` processes the lines of the *included* file as if they were in the current makefile. This makes it easier to change information because you can change it in one common file rather than each individual makefile.

The syntax for the `include` entry is:

```
include <pathname>
```

`include` opens the specified file (`<pathname>`) and processes the lines from that file as if they appeared in the current makefile.

Note

You can nest included makefiles up to seven times.

Here is an example of a file you might include in your makefiles:

```
File: make_os9.tpl
OSROOT = /dd/MWOS
DEFS = -v=$(OSROOT)/SRC/DEFS -v=$(OSROOT)/OS9/SRC/DEFS
```

```
LIBS = -w=$(OSROOT)/OS9/68000/LIB \
       -w=$(OSROOT)/OS9/68000/LIB/HOST1
```

Here is an example of including the `make_os9.tpl` common file in a makefile:

```
File: makefile.os9
include ../make_os9.tpl

CFLAGS = $(DEFS)
LFALGS = $(LIBS)

test:
```

Macro Entry

A *macro definition line* has the following syntax:

```
<name> = [<value>]
```

This defines a macro called `<name>` with `<value>`. `<value>` can contain references to other previously defined macros.

For More Information

Refer to the **Macro Recognition** section for more information about macros.

Here are examples of macro entries:

```
PROG = util
ODIR = /h0/CMDS
TARGET = $(ODIR)/$(PROG)
```

If a macro name has already been used, redefinitions have no effect. This allows a macro definition from the `make` command line to override macro definitions within the makefile.

A macro does not need to have a value specified in the makefile. In other words,

```
LOPTS =
```

is a valid line for a makefile. This type of macro is often used when a command line definition of a macro is expected.

Summary

Dependency entry

Specify the relationship of the target file and the dependents used to build the target file.

```
<target>: [<dependent>] {[<dependent>]}
```

Command entry

An OS-9 command line. Must begin with one or more spaces or tabs.

```
<OS-9 command line>
<OS-9 command line>\
<continued command line>
```

Comment entry

Any line beginning with an asterisk (*) or a pound sign (#).

Exception: A pound sign followed by a digit (0-9) is considered a stack space command line modifier.

```
<target>: [<dependent>] {[<dependent>]}
* comment
* comment
<command entry> # this is also a comment
```

Include entry

Open the specified file and process the lines as if they appear in the current makefile.

```
include <pathname>
```

Macro entry

Define a name and its corresponding value.

```
<name> = <value>
```

Processing the Make File

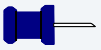
`make` processes the makefile three times:

1. Examines makefile, sets up dependency table, and determines the target file.
2. Resolves implicit dependencies.
3. Compares file dates. Re-makes files as needed.

During the first pass, `make` examines the makefile and sets up a table of dependencies. This table of dependencies stores the target file and the dependency files exactly as they are listed in the makefile. When `make` encounters a name on the left side of a colon, it first checks to see if it has encountered the name before. If it has, `make` connects the lists and continues.

After reading the makefile, `make` determines the target file on the list. It then makes a second pass through the dependency table. During this pass, `make` tries to resolve any existing *implicit dependencies*. (Implicit dependencies are covered in the next section.)

`make` does a third pass through the list to get and compare the file dates. When `make` finds a file in a dependency list that is newer than its target file, it executes the specified command(s). If no command entry is specified, `make` generates a command based on the implicit dependencies and re-makes the file.



Note

Because OS-9 only stores the time down to the closest minute, `make` re-makes a file if its date matches one of its dependents.

When a command is executed, it echoes to standard output. `make` normally stops if an error code is returned when a command line is executed.

Implicit Dependencies

When `make` generates a command line, it assumes the target file is a program to compile. If the target file *is not* a program to compile, you must specify any necessary command entries for each dependency list. `make` uses the specified mode (or the default) and following definitions/rules when forced to create a command line.

For More Information

Refer to the **Processing the Make File** section for more information about how make generates command lines.

Command Line Rules

Depending on the type of file, `make` uses the following rules:

Table 6-1 Command LIne Rules

Files	Definition/Rule
Object	Files with no suffix. An object file is made from a relocatable file and linked when it needs to be made.
Relocatable	Files appended by either the `.r` suffix (`compat` and `ucc` modes) or the `.o` suffix (`c89` mode). Relocatable files are made from source files and assembled or compiled if they need to be made.

Table 6-1 Command LIne Rules (continued)

Files	Definition/Rule

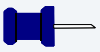
Source	Files with one of the following suffixes:
	`ucc` mode:
	`.a .o .i .pp .c .f`
	`compat` mode:
	`.a .c .f`
	`c89` mode:
	`.s .be .ic .i .c .f`

Defaults

`make` uses the following defaults:

Table 6-2 Defaults

Description:	Default:
Compiler	`cc`
Assembler	`r68`
Linker	`cc`
Mode	`ucc`
Directory for all files	The current data directory (.)



Note

Only use the *default linker* with programs using `Cstart`.

Modes

Three modes are built into `make`:

Table 6-3 Modes Built Into make

Mode	Description
`compat`	The `compat` files reflect the `compat` mode of the Ultra C executive. This is also the mode you use if you are still using the V3.2 C Compiler.
`c89`	The `c89` rules reflect the `c89` mode of the Ultra C executive.
`ucc`	The `ucc` rules reflect the `ucc` mode of the Ultra C executive

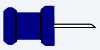
For More Information

Refer to the ***Using Ultra C Manual***, the Using the C Executive chapter for more information about Ultra C modes.

Because Ultra C has several intermediate phases between compilations, additional rules in the `c89` and `ucc` rule tables bring each intermediate compile phase to its upper levels of compilation.

For example, in `ucc` mode, `make` has built-in knowledge of bringing an I-code file (`.i`) to:

- A back-end file (`.o`)
- An assembly optimized file (`.a`)
- A relocatable object file (`.r`)
- An executable object file



Note

To view the built-in rules for the current mode, use the `-r` option. For example, `make -rn` shows only the rules for `makefile`.

Set Mode

There are three ways to tell `make` to use a particular mode:

1. Set the `MWMAKEOPTS` environment variable to the desired mode. For example:

   ```
   setenv MWMAKEOPTS -mode=c89
   ```

2. Use the `-mode=<mode>` option on the command line. This method overrides the environment variable. For example:

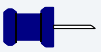
   ```
   make -mode=compat
   ```

3. Use the `-mode=<mode>` option within your makefile. This overrides both the environment variable and command line option.

Macro Recognition

In addition to recognizing compilation rules and definitions, make recognizes certain macros. make recognizes a macro by the dollar sign ($) character in front of the name. If a macro name is longer than a single character, you must enclose the entire name within parentheses. For example:

$R	refers to the macro R.
$(PFLAGS	refers to the macro PFLAGS.
$(B) and $B	refer to the macro B.
$BR	is interpreted as the value for the macro B followed by the character R.



Note

If you define a macro in your makefile and then redefine it on the command line, the command line definition overrides the makefile definition. This feature is useful for compiling with special options.

You can define your macros in the makefile for convenience or on the command line for flexibility. Macros are defined with the form <macro name> = <expansion>. The expansion is substituted for the macro name whenever the macro name appears. For example:

```
IDIR = /h0/MWOS/dir1
```

To increase make's flexibility, you can supply a value for special macros in the makefile. If you do not supply a value, make uses the default value. make uses these macros when you must make assumptions to generate command lines or search for unspecified files. For example, if no source file is specified for program.r, make searches either the directory specified by SDIR or the current data directory for program.a (or .c, .p, .f).

Special Macros

`make` recognizes the following special macros:

Table 6-4 Special Macros

Macro	Description
`IDIR=<path>`	`make` searches the directory specified by `path` for all I-code files not specified by a full path list. If `IDIR` is not defined in the makefile, `make` searches the current directory by default.
`ODIR=<path>`	`make` searches the directory specified by `<path>` for all files with no suffix or relative pathlist. If `ODIR` is not defined in the makefile, `make` searches the current directory by default.
`SDIR=<path>`	`make` searches the directory specified by `<path>` for all source files not specified by a full pathlist. If `SDIR` is not defined in the makefile, `make` searches the current directory by default.
`RDIR=<path>`	`make` searches the directory specified by `<path>` for all relocatable files not specified by a full pathlist. If `RDIR` is not defined, `make` searches the current directory by default.
`CFLAGS=<opts>`	These compiler options are used in implicit compiler command lines. `make` uses the specified options when it generates a compiler command line.
`RFLAGS=<opts>`	These assembler options are used in implicit assembler command lines. `make` uses the specified options when it generates an assembler command line.

Table 6-4 Special Macros (continued)

Macro	Description
LFLAGS=<opts>	These linker options are used in implicit linker command lines. make uses the specified options when it generates a linker command line.
CC=<comp>	make uses this compiler when generating command lines. The default is cc.
RC=<asm>	make uses this assembler when generating command lines. The default is r68.
LC=<link>	make uses this linker when generating command lines. The default is cc.

Reserved Macros

Some reserved macros are expanded when a command line associated with a particular file dependency is forked. You can only use these macros on a command line. They can be useful when you need to be explicit about a command line but have a target program with several dependencies. In practice, they are wildcards with the following meanings:

Table 6-5 Reserved Macros

Macro	Means to use

$@	The current target, including its full path and any suffix.
	For example:
	`/h0/CMDS/prog : RELS/prog.r RELS/prog2.r`
	`cc RELS/prog.r RELS/prog2.r -fd=$@`
	Generates:
	`cc RELS/prog.r RELS/prog2.r -fd=/h0/CMDS/prog`

$*	The base name of the target. That is, the target name minus any pathlist or extension.
	For example:
	`/h0/CMDS/prog : RELS/prog.r RELS/prog2.r`
	`chx /h0/CMDS ; cc RELS/prog.r RELS/prog2.r -fd=$*`
	Generates:
	`chx /h0/CMDS ; cc RELS/prog.r RELS/prog2.r -f=prog`

Table 6-5 Reserved Macros (continued)

Macro	Means to use

`$?`	The list of files found to be newer than the target on a given dependency line.
	For example:
	`print.new : main.c file1.c file2.c`
	`spl -njnh $?`
	`touch print.new`
	Generates the following if `main.c` and `file2.c` were modified since the last time the target `print.new` was made:
	`spl -njnh main.c file2.c`
	`touch print.new`
	Note: `file1.c` was not remade because its date was not later than `print.new`.

make Generated Command Lines

`make` can generate three types of command lines:

- **Compiler Command Lines**
- **Assembler Command Lines**
- **Linker Command Lines**

Compiler Command Lines

Make generates *compiler command lines* if a source file with a source file suffix needs to be recompiled. `make` generated compiler command lines have the following syntax:

`compat` and `ucc` mode:

```
$(CC)  $(CFLAGS)  $(SDIR)/<file> -f=$(ODIR)/<file>
```

`c89` mode:

```
$(CC)  $(CFLAGS)   $(SDIR)/<file> -f $(ODIR)/<file>
```

For example, after macro replacement the command line might look like:

`compat` and `ucc` mode:

```
cc -td=/r0 -i test.c -f=test
```

`c89` mode:

```
cc -td /r0 -i test.c -f test
```

Assembler Command Lines

Make generates *assembler command lines* if an assembly language source file needs to be re-assembled. `make` generated assembler command lines have the following syntax:

`compat` and `ucc` mode:

```
$(RC)  $(RFLAGS)  $(SDIR)/<file>.a -o=$(RDIR)/<file>.r
```

`c89` mode:

```
$(RC)  $(RFLAGS)  $(SDIR)/<file>.s -o=$(RDIR)/<file>.o
```

For example, after macro replacement the command line might look like:

`compat` and `ucc` mode:

```
r68 -q -bt ../ASM/test.a -o=RELS/test.r
```

`c89` mode:

```
r68 -q -bt ../ASM/test.s -o=RELS/test.o
```

Linker Command Lines

Make generates *linker command lines* if an object file needs to be relinked to remake the program module. `make` generated linker command lines have the following syntax:

`ucc` and `compat` mode:

```
$(LC)  $(LFLAGS)  $(RDIR)/<file>.r -f=$(ODIR)/<file>
```

`c89` mode:

```
$(LC)  $(LFLAGS)  $(RDIR)/<file>.o -f $(ODIR)/<file>
```

For example, after macro replacement the command line might look like:

`compat` and `ucc` mode:

```
cc -i -l=mylib.l RELS/prog.r -f=/h0/CMDS/RIC/prog
```

c89 mode:

```
cc -i -l mylib.l RELS/prog.o -f /h0/CMDS/RIC/prog
```

WARNING

When make is generating a command line for the linker, it looks at its list and uses the first relocatable file it finds, but only the first one. For example:

```
prog: x.r y.r z.r
```

Generates:

cc x.r, not cc x.r y.r z.r or cc prog.r

make Options

`make` has several options you can use to maintain files/modules. You may include these options on the command line when you run `make` or, for convenience, put them in the makefile.

When a command is executed, it is echoed to standard output, unless you use the `-s` (silent) option or start the command line with an *at* sign (`@`). Use the `-n` option to echo commands to standard output but not execute them. This is useful when building your original makefile.

`make` normally stops if an error code is returned when a command line is executed. To ignore errors, use the `-i` option or begin the command line with a hyphen (`-`).

Sometimes it is helpful to see the file dependencies and their associated dates. The `-d` option turns on the `make` debugger and supplies:

- A complete listing of the macro definitions.
- A listing of the files as it checks the dependency list
- All the file modification dates. If the `make` debugger cannot find a file to examine its date, it assumes a date of -1/00/00 00:00, indicating you should update the file.

For More Information

Refer to the **Implicit Dependencies** section for information about rules.

To update the date on a file without remaking it, use the `-t` option. `make` merely opens the file for update and then closes it, thus making the date current.

If you are quite explicit about your makefile dependencies and do not want `make` to assume anything, use the `-b` option to turn off the built-in rules governing implicit file dependencies.

Table 6-6 make Options

Option	Description
`-?`	Display the options, function, and command syntax of `make`.
`-b`	Do not use built-in rules.
`-bo`	Do not use built-in rules for object files.
`-d`	Print the dates of the files in the makefile (debug mode).
`-dd`	Double debug mode. Very verbose.
`-f-`	Read the makefile from standard input.
`-f=<path>`	Specify `<path>` as the makefile. If you do not specify `-f- or -f=<path>`, `make` looks for a file named `makefile`. If `<path>` is specified as a hyphen (`-`), `make` commands are read from standard input.
`-i`	Ignore errors.
`-mode`	Set the mode (`compat`, `ucc`, or `c68`). If used on the command line, this option overrides the environment variable (`MWMAKEOPTS`). If the option is set in the makefile, it overrides both the environment variable and the command line option.
`-n`	Display commands but do not execute them.

Table 6-6 make Options (continued)

Option	Description

-o	Do not assume object files need ROF files. Disable make's assumption that a dependency line with an object file as the target must have an ROF file in the dependency list. For example:
	`clean : del $(RDIR) /* .r $ (ODIR) / $ (PROG)`
	Without the -o option it generates:
	`make: can't find source file to make "clean.r"`
	With -o, the make line executes as expected.

-r	View the built-in rules for the current mode. For example, make -rn lists the current rules without running the makefile.
-s	Silent Mode. Execute commands without echo.
-t	Update the file dates without executing commands.
-u	Do the make regardless of the file dates.
-x	Use the cross-compiler/assembler.
-z	Read a list of make targets from standard input.
-z=<path>	Read a list of make targets from <path>.

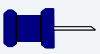
MWMAKEOPTS Environment Variable

The MWMAKEOPTS environment variable is used by make to set options and macro definitions. The following example shows how you might use MWMAKEOPTS to turn on debugging, turn off built-in rules, and set the DEBUG macro to 1:

```
setenv MWMAKEOPTS "-d -b DEBUG=1"
```

make parses MWMAKEOPTS before processing command line arguments.

Another feature of the MWMAKEOPTS is its ability to contain quoted strings to set a macro to multiple whitespaced, separated words. An example of this is the setting, -d "RUN_CMD=exec -n". This setting enables the -d option and sets the macro RUN_CMD to "exec -n".



Note

Do not use target names in the MWMAKEOPTS environment variable because make ignores them.

Examples

The rest of this chapter contains examples of make files. These examples are not meant to be totally inclusive of the ways in which you can use make.

Compiling C Programs

In this example, make is used to compile high level language modules. Each command and dependency is specified.

```
program: xxx.r yyy.r
    cc xxx.r yyy.r -f=program
xxx.r: xxx.c /d0/defs/oskdefs.h
    cc xxx.c -eas
yyy.r: yyy.c /d0/defs/oskdefs.h
    cc yyy.c -eas
```

This makefile specifies program is made up of two .r files: xxx.r and yyy.r. These files depend on xxx.c and yyy.c, respectively, and both depend on the oskdefs.h file.

- If either xxx.c or /d0/defs/oskdefs.h has a date more recent than xxx.r, the command cc xxx.c -eas is executed.
- If either yyy.c or /d0/defs/oskdefs.h is newer than yyy.r, then cc yyy.c -eas is executed.
- If either of the former commands are executed, the command cc xxx.r yyy.r -f=program is also executed.

In this example, make specifies each command it must execute. Often this is unnecessary as make uses specific definitions, macros, and built-in assumptions to facilitate program compilation and generate its own commands.

Refining the C Compiler Example

Knowing how make works and understanding the built-in rules can simplify coding immensely:

```
program: xxx.r yyy.r
   cc xxx.r yyy.r -f=program
xxx.r yyy.r: /d0/defs/oskdefs
```

This makefile now exploits make's awareness of file dependencies. No mention is made of the C language files; therefore, make looks in the directory specified by the macro definition SDIR = <path> and adjusts the dependency list accordingly. In this case, make looks in the current directory by default. make also generates a command line to compile xxx.r and yyy.r if one or both needs to be updated.

Further simplification would be possible if program was made up of only one source file:

```
program:
```

make assumes the following from this simple command:

- program has no suffix. It is an object file and therefore needs to rely on relocatable files to be made.
- No dependency list is given; therefore, make creates an entry in the table for program.r.
- After creating an entry for program.r, make creates the entry for a source file connected to the relocatable file.

Assuming it found program.a, make checks the dates on the various files and generates one or both of the following commands if required:

```
r68 program.a -o=program.r
cc program.r -f=program
```

Make File that Uses Macros

Using these inherent features of make can be especially helpful if you have several object files you want make to check:

```
* beginning
ODIR = /d0/cmds
RDIR = rels
UTILS = attr copy load dir backup dsave
SDIR = ../utils/sources
utils.files: $(UTILS)
   touch utils.files
* end
```

make looks in rels for attr.r, copy.r, etc. and looks in ../utils/sources for attr.c, copy.c, etc. make then generates the proper commands to compile and/or link any of the programs that need to be made. If one of the files in UTILS is made, the command touch utils.files is forked to maintain a current overall date.

Putting It All Together

The following example is a makefile to create make:

```
* beginning
ODIR = /h0/cmds
RDIR = rels
CFILES = domake.c doname.c dodate.c domac.c
RFILES = domake.r doname.r dodate.r
PFLAGS = -p64 -nh1
R2 = ../test/domac.r
RFLAGS = -q
make: $(RFILES) $(R2) getfd.r
  linker
$(RFILES): defs.h
$(R2): defs.h
  cc $*.c -eas=../test
print.file: $(CFILES)
  pr $? $(PFLAGS) >-/p1
```

```
    touch print.file
  *end
```

This makefile looks for the `.r` files listed in `RFILES` in the directory specified by `RDIR: rels`. The only exception is `../test/domac.r`, which has a complete pathlist specified.

Even though `getfd.r` does not have any explicit dependents, its dependency on `getfd.a` is still checked. The source files are all found in the current directory.

Note: you can also use this makefile to make listings. By typing `make print.file` on the command line, `make` expands the macro `$?` to include all of the files updated since the last time `print.file` was updated. If you keep a dummy file called `print.file` in your directory, `make` only prints out the newly made files. If no `print.file` exists, `make` prints all files.

Chapter 7: Making Backups

This chapter explains the concept of incremental backups. The OS-9 utilities to create the backups are detailed here. It also offers two different strategies for making backups.

This chapter includes the following topics:

Incremental Backups

Whether it's caused by system failure or accidental erasure, loss of stored data is a programmer's nightmare. Consequently, backups of files, programs, and disks are a normal part of existence. Backing up a hard disk is usually slow and tedious because the entire system is backed-up.

You can use incremental backups instead of full system backups. Incremental backups save only the files that have changed since the last backup. You must still perform a full system backup, but by using incremental backups you can perform them less often.

OS-9 provides two utilities you can use with either tape or disk media to facilitate the use of incremental backups:

- `fsave`
- `frestore`

Certain terms must be defined to discuss incremental backups.

- A full system backup is referred to as a *level 0 backup*.
- Consequent incremental backups are referenced by different level numbers.

For example, a level 5 backup includes all files changed since the most recent backup with a level less than 5. While this sounds complex, it is actually quite easy to use and extremely helpful.

Two other terms need to be defined:

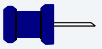
- A *source device* is the directory structure or file you are backing up.
- A *target device* is the tape or disk you are using to hold your backup information.

Making an Incremental Backup: The fsave Utility

The fsave utility performs an incremental backup of a directory structure to tape(s) or disk(s). The syntax for the fsave utility is:

```
fsave [<opts>] [<path>] [<opts>]
```

Typing fsave by itself on the command line makes a level 0 backup of the current directory onto a target device with the name /mt0.



Note

/mt0 is the default OS-9 device name for a tape device just as /h0 is the default OS-9 device name for a hard disk.

/h0/sys/backup_date is the backup log file fsave maintains. Each time you execute an fsave, the backup log is updated. The backup log keeps track of the name of the backup, the date it was created, and more importantly, the level of the backup. When you execute fsave, this backup log is examined to find the specified level of the current backup and the previous backups with the same name. Once the backup is finished, a new entry is made in the file indicating the date, name, and level of the current backup.

fsave has the following options:

Table 7-1 fsave Options

Option	Description
-?	Display the use of fsave.
-b[=]<int>	Allocate <int>k buffer size to read files from the source disk.

Table 7-1 fsave Options (continued)

Option	Description
`-d[=]<dev>`	Specify the target device to store the backup. The default is `/mt0`.
`-e`	Do not echo file pathlists as they are saved to the target device.
`-f[=]<path>`	Save to the file specified by `<path>`.
`-g[=]<int>`	Specify a backup of files owned by group number `<int>` only.
`-j[=]<num>`	Specify the minimum system memory request.
`-l[=]<int>`	Specify the level of the backup to perform.
`-m[=]<path>`	Specify the pathlist of the date backup log file to use. The default is `/h0/sys/backup_dates`. **NOTE:** The file must exist; `fsave` does not create the file.
`-p`	Turn off the `mount volume` prompt for the first volume.
`-s`	Display the pathlists of all files needing to be saved and the size of the entire backup without actually executing the backup procedure.
`-t[=]<dirpath>`	Specify the alternate location for the temporary index file.
`-u[=]<int>`	Specify a backup of files owned by user number `<int>` only.

Table 7-1 fsave Options (continued)

Option	Description
`-v`	Do not verify the disk volume when mounted.
`-x[=]<int>`	Pre-extend the temporary file. `<int>` is given in kilobytes.

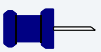
The fsave Procedure

When you start an `fsave` procedure, `fsave` prompts you to mount the first volume to use. `Volume` in this case refers to the disk or tape used to store the backup:

```
fsave: please mount volume.
(press return when mounted).
```

If you use a disk as the backup medium, `fsave` verifies the disk and displays the following information:

```
verifying disk
Bytes held on this disk: 546816
Total data bytes left:   62431
Number of Disks needed: 1
```



Note

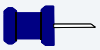
The numbers above are used only as an example.

If you use a tape as the backup medium, no preliminary information is displayed and the backup begins at this point.

As each file is saved to the backup device, its pathlist is echoed to the terminal. If this is a long backup, you may want to use the `-e` option to turn off the pathlist echoing.

If `fsave` receives an error when trying to backup a file, it displays the following message and continues the `fsave` operation:

```
error saving <file>, error - <error number>, its incomplete
```



Note

The most common error found when executing `fsave` is a record lock error. Record lock errors are caused when another user has the file in question open.

To prevent record lock errors, perform `fsave` operations only when no one else is using the system.

If the backup requires more than one volume, `fsave` prompts you to mount the next volume before continuing.

At the end of the backup, `fsave` prints the following information:

```
fsave: Saving the index structure
Logical backup name:
Date of backup:
Backup made by:
Data bytes written:
Number of files:
Number of volumes:
Index is on volume:
```

The index to the backup is saved on the last volume used.

`fsave` performs recursive backups for each pathlist if one or more directories are specified on the command line. You can specify a maximum of 32 directories on the command line.

WARNING

When using disks for backup purposes, `fsave` does not use an RBF file structure to save the files on the target disk. It creates its own file structure. This makes the backup disk unusable for any purpose other than `fsave` and `frestore` without reformatting the disk. The backup destroys any data stored on the disk before using `fsave`.

Example fsave Commands

Typing `fsave` by itself on a command line specifies a level 0 backup of the current directory. This assumes the `/mt0` device is used and `/h0/SYS/backup_dates` is used as the backup log file for this backup.

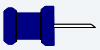
The following command specifies a level 2 backup of the current directory using the `/mt1` device. `/h0/misc/my_dates` is used as the backup log file:

```
$ fsave -l=2 -d=/mt1 -m=/h0/misc/my_dates
```

The following command specifies a level 0 backup of all files owned by user 0.0 in the `CMDS` directory, if `CMDS` is in your current directory:

```
$ fsave -pb=32 -g=0 -u=0 -d=/d2 CMDS
```

This backup uses `/d2` as the target device and `/h0/sys/backup_dates` as the backup log file. The mount volume prompt is not generated for the first volume. A 32K buffer is used to read the files from the `CMDS` directory.



Note

The `backup_dates` file must exist. `fsave` does not create the file. If the file does not exist, use `touch <filename>` to create an empty backup log file prior to starting `fsave`.

Restoring Incremental Backups: The frestore Utility

The `frestore` utility restores a directory structure from multiple volumes of tape or disk media. The syntax for `frestore` is:

```
frestore [<opts>] [<path>] [<opts>]
```

Typing `frestore` by itself on the command line attempts to restore a directory structure from the `/mt0` device to the current directory.

If you specify the pathlist of a directory on the command line, the files are restored in that directory. `fsave` creates the directory structure and an index of the directory structure.

If more than one tape/disk is involved in the `fsave` backup, each tape/disk is considered to be a different *volume*. The volume count begins at one (`1`). When you begin an `frestore` operation, you must use the last volume of the backup first because it contains the index of the entire backup.

`frestore` first attempts to locate and read the index of the directory structure of the source device. `frestore` then begins an interactive session with you to determine which files and directories in the backup should be restored to the current directory.

`frestore` has the following options:

Table 7-2 frestore Options

Option	Description
`-?`	Display the use of `frestore`.
`-a`	Force access permission for overwriting an existing file. You must be the owner of the file or a super user to use this option.
`-b[=]<int>`	Specify the buffer size to use to restore the files.

Table 7-2 frestore Options (continued)

Option	Description
`-c`	Check the validity of files without using the interactive shell.
`-d[=]<path>`	Specify the source device. The default is `/mt0`.
`-e`	Display the pathlists of all files in the index, as the index is read from the source device.
`-f[=]<path>`	Restore from a file.
`-i`	Display the backup name, creation date, group.user number of the owner of the backup, volume number of the disk or tape, and whether the index is on the volume. `-i` does not cause any files to be restored. The information is displayed, and `frestore` is terminated.
`-j[=]<int>`	Set the minimum system memory request.
`-p`	Suppress the prompt for the first volume.
`-q`	Overwrite an already existing file when used with the `-s` option.
`-s`	Force `frestore` to restore all files from the source device without an interactive shell.
`-t[=]<dirpath>`	Specify an alternate location for the temporary index file.
`-v`	Display the same information as `-i`, but do not check for the index. `-v` does not cause any files to be restored. The information is displayed and `frestore` is terminated.

Table 7-2 frestore Options (continued)

Option	Description
`-x[=]<int>`	Pre-extend the temporary file. `<int>` is given in kilobytes.
`-z[=]<path>`	Add pathlists in `<path>` to restoration list (from root).

The Interactive Restore Process

Once you call `frestore`, the following prompt is displayed:

```
frestore> mount the last volume
(press return when ready)
```

When you are ready, `frestore` attempts to read in the index and create the directory structure of the backup. It then displays the prompt:

```
frestore>
```

This prompt tells you that you are in the interactive shell. If the index is not on the mounted volume, `frestore` displays an error message and again prompts you to mount the last volume.

Once in the interactive shell, the `frestore` commands and options are displayed when you type a return at the prompt:

```
frestore>
commands:
add [<path>] [-g=<#> -u=<#> -r -a] -- marks files for restoration
del [<path>] [-g=<#> -u=<#> -r -a] -- unmarks files for restoration
dir [<dir names>] [-e] -- displays a directory or directories
chd <path> -- changes directories within the restore file structure
pwd -- gives the pathlist to current dir in the restore file structure
cht <path> -- changes directories on target system
est [<path>] [-f -q] -- restores marked files in and below the current dir
check [-f] -- checks validity if marked files in and below the current dir
dump [<file>] -- dumps the contents of a file to stdout
$   -- forks a shell
quit -- quit frestore program
options:
-g=<group#> -- only mark files with 'group#'
-u=<user#> -- only mark files with 'user#'
```

```
    -r -- mark directories recursively
    -e -- display directory with extended format
    -f -- force restoration of already restored files
    -q -- overwrite already existing files without question
    -a -- force marking or unmarking of an already restored file or dir
    * -- matches any string of characters on 'add' or 'del' only
    ? -- matches any single character on 'add' or 'del' only
```

The index from the source device sets up a restore file structure paralleling the usual OS-9 file/directory structure.

Use the `dir` and `chd` shell commands to display the restore file structure. For example:

```
frestore>dir
                  Directory of .
DIR1      file1     file2     file3
```

All files to be backed up on to the source device appear in the restore file structure regardless of what volume they appear in. Information concerning the file structure is available using the `-e` option with the `dir` command:

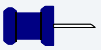
```
frestore>dir -e
Directory of .
  Owner   Last modified  Attributes Volume Block Offset   Size    Name
------  -------------- ----------- ------ ----- ------  -----   ------
       1.23    89/08/22 16/14  ----r-wr       1      0      0   CF12    file1
       1.23    89/08/25 11/00  ----r-wr       1      2      0   A356    file2
       1.23    89/08/21 11/12  ----r-wr       1      4      0   45F0    file3
       1.23    89/08/24 10/57  d-ewrewr       0      5      0    120    DIR1
```

The interactive shell allows you to mark the files you want restored with the `add` command. You can mark groups of files using the options of the `add` command:

`-g` Mark files by group number.

`-u` Mark files by user number.

`-r` Mark all directories within a specified directory.



Note

Specify the following:

- relative or complete pathlists within the restore file structure to mark files one at a time.
- a pathlist of a directory to mark an entire directory.

Marking files does not restore them. It merely marks them as *to be restored*. You can see this when you use the `dir` command. Each file added to the *to be restored* list is marked by a plus sign (`+`) by its filename.

For example, the following directory has `file1` and `file2` marked for restoration, but `file3` is not marked. The directories `DIR1` and `DIR2` also have marked files:

```
frestore>add file1 file2 dir1/file5 dir1/file6 dir2/file7
frestore>dir
             Directory of .
+DIR1    +DIR2    +file1   +file2
file3
frestore>dir dir1
             Directory of DIR1
file4    +file5   +file6
frestore>dir dir2
             Directory of DIR2
+file7   file8
```

Unmark Files

You can unmark files with the `del` command. To unmark entire directories, specify the directory's name on the command line. If you also use the `-r` option, all files and directories included in the specified directory are unmarked. For example:

```
frestore>del -r dir2
frestore>dir
```

```
             Directory of . 10:42:32
+DIR1    DIR2     +file1   +file2
file3
frestore>dir dir2
             Directory of DIR2
file7    file8
```

Restore Files

Once files are marked, use the `rest` command to restore the target device's current directory.

Overwrite Existing Files

Files existing on the target system with the same name are overwritten without prompting if you use `rest -q`. Otherwise, `frestore` displays the following prompt:

```
frestore> file1 already exists
          write over it or skip it (w/s)
```

Change Directories on the Target Device

The `cht` command allows you to change directories on the target device. This allows you to selectively restore files to specific directories.

Note

An asterisk (`*`) preceding the name of a file in a `dir` listing indicates an error occurred while backing up this file. This file is incomplete and should not be restored.

After restoring files, you may continue marking and unmarking files. Files previously restored have a hyphen (-) displayed next to their names in the restore file structure:

```
frestore>dir
             Directory of . 10:42:32
-DIR1    DIR2      -file1    -file2
file3
frestore>dir dir1
                     Directory of DIR1
file4    -file5    -file6
```

Restore Files More Than Once

There are two methods of restoring files more than once:

1. Use the `-a` option with the `add` command. This forces the file(s) previously marked as restored to be marked as *to be restored*.
2. Use the `-f` option with the `rest` command. This restores any file previously marked as restored in the current directory.

Restore Files Without Using the Interactive Shell

The `-s` option forces `frestore` to restore all files/directories of the backup from the source device without the interactive shell.

Specify a Source Device

Using the `-d` option allows you to specify a source device other than `/mt0`. For example, to restore all files/directories found on the `/mt1` source device to the directory `BACKUP` without using the interactive shell, type:

```
$ frestore -d=/mt1 -s BACKUP
```

Identify the Backup Mounted on the Source Device

The `-v` option causes `frestore` to identify the name and volume number of the backup mounted on the source device. The date the backup was made and the group.user number of the person who made the backup is also displayed. `-v` does not restore any files. For example:

```
$ frestore -v
Backup: DOCUMENTATION
Made:   1/16/91 10:10
By:     0.0
Volume: 0
```

Indicate Whether the Index Is on the Volume

The `-i` option displays the above information and also indicates whether the index is on the volume. Both `-v` and `-i` terminate `frestore` after displaying the appropriate information. These options are useful when trying to locate the last volume of the backup if any mix-up has occurred.

Echo Pathlists

The `-e` option echoes each file pathlist as the index is read off the source device.

Example Command Lines

To restore files/directories from the source device `/mt0` to the current directory by way of an interactive shell, type:

```
$ frestore
```

The following example restores files/directories from the source device `/d0` to the current directory using a 32K buffer to write the restored files. As each file is read from the index, the file's pathlist is echoed to the terminal.

```
$ frestore -eb=32 -d=/d0
```

Incremental Backup Strategies

Many different strategies are available for those concerned with regularly scheduled backups. Most strategies are well documented in computer books and magazines. The following two strategies are offered as examples of methods that can be used.

The Small Daily Backup Strategy

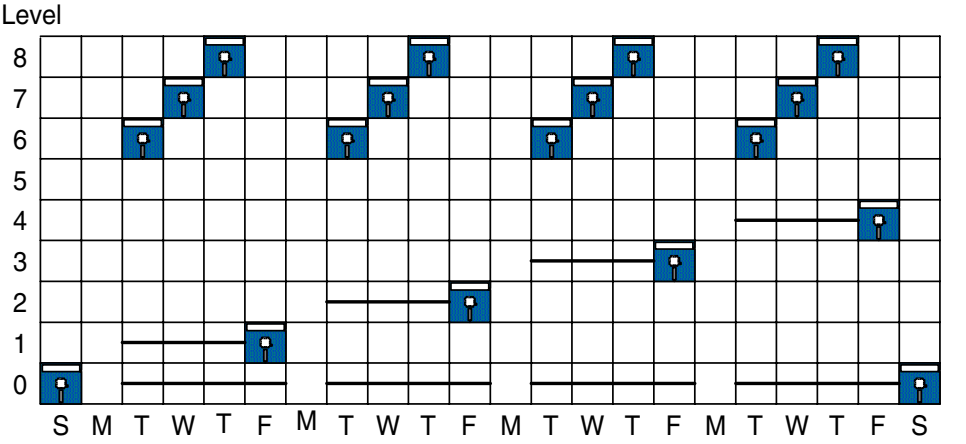
This strategy requires making a level 0 backup once every four weeks. Level 1, level 2, level 3, and level 4 backups are made on the weeks following the level 0 backup. Between each major backup, four daily backups would be made: level 5, 6, 7, and 8. A recommended daily schedule is graphically presented in **Figure 7-1**.

This strategy is ideal for small microcomputer systems backed up by floppy disks. Mounting disks is much easier and faster than tapes. You can usually keep each daily backup on one disk.

This strategy is also perfect for small timely backups with little redundancy in the backups.

One major disadvantage of this scheme is the restore time necessary in case of a major system failure such as a hard disk being formatted, erased, or corrupted. Because of the lack of redundancy, more `frestore` operations are necessary to re-create the systems file structure. On large systems with tape backups, this is a major consideration.

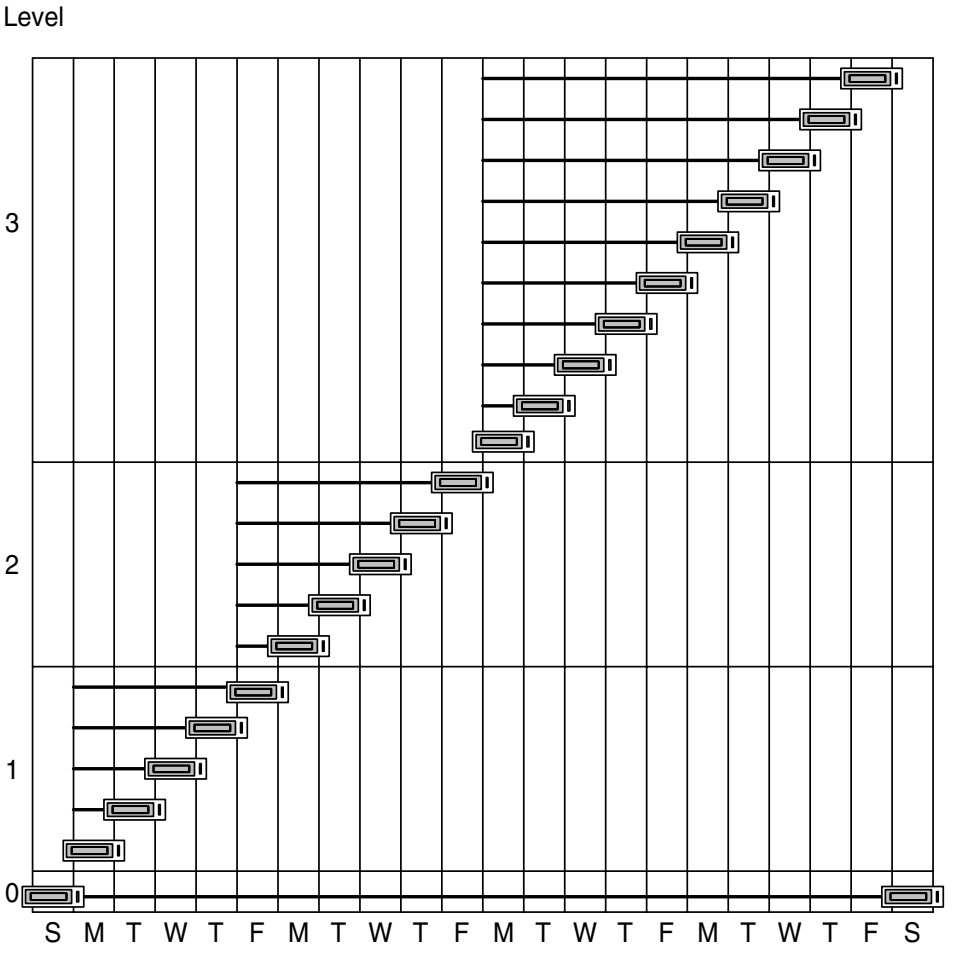
Figure 7-1 Small Daily Backup Strategy



The Single Tape Backup Strategy

While most strategies rely on scheduled backup level changes, the *single tape backup* strategy depends on the size of the backup. The idea behind this strategy is to increase the level of the backup only when the backup cannot fit on a single tape. The only scheduled level backup is the level 0 backup. The level 0 backup occurs only when a higher level backup would not fit on a single tape or once a month, whichever occurs first. An example month's schedule is graphically presented in **Figure 7-2**.

Figure 7-2 Single Tape Backup Strategy



This strategy is designed for tape backups of larger systems. Tapes are used efficiently because a question as to how many tapes are needed never arises. This strategy also cuts down on person hours, tape mounting, and storage space used for tapes. It allows for enough redundancy to make restoring a full system fairly painless.

Disadvantages, however, do exist. Each time you do a backup, you must determine the size of the backup using `fsave -s`. As you near a full tape's worth of data, this takes an increasing amount of time.

Use of Tapes/Disks

Whatever strategy you use, you must make a decision concerning the number of tapes or disks to use. This decision must weigh the emphasis placed on:

- Redundancy
- Resources
- Person-hours
- Storage

It must be offset with the possibility of tape or disk failure and system restoration.

In the first example strategy, you must make the daily backups on different volumes to overcome the lack of redundancy. You can use the four daily volumes week after week as daily backup volumes because of the lower level backups at the beginning of each week.

In the second example, theoretically, you could use the same tape for each day until a new level backup is reached. This insures no redundancy and minimal storage. It is also the most dangerous in case of tape failure. Using a number of alternating tapes for each level cuts down on storage and still allows a safety net in the case of tape failure. Using alternating level 0 tapes is another possibility.

The Tape Utility

OS-9 provides a `tape` controller utility to facilitate setting up, reading, and rewinding tapes from the terminal. When using tape media to backup or restore your system, the `tape` utility is very practical. The syntax of the `tape` utility is:

```
tape {<opts>} [<dev>] {<opts>}
```

`tape` uses the default device `/mt0` if you do not specify the tape device `<dev>` on the command line and you do not use the `-z` option.

`tape` has the following available options:

Table 7-3 tape Options

Option	Description
`-?`	Display the use of `tape`.
`-b[=<num>]`	Skip a specified number of blocks. Default is one block. If `<num>` is negative, the tape skips backward.
`-e=<num>`	Erase a specified number of blocks of tape.
`-f[=<num>]`	Skip a specified number of tapemarks. Default is one tapemark. If `<num>` is negative, the tape skips backward.
`-o`	Put tape off-line.
`-r`	Rewind the tape.
`-s`	Determine the block size of the device.
`-t`	Retension the tape.
`-w[=<num>]`	Write a specified number of tapemarks. Default is one tapemark.

Table 7-3 tape Options (continued)

Option	Description
`-z`	Read a list of device names from standard input. The default is `/mt0`.
`-z=<file>`	Read a list of device names from `<file>`.

If you specify more than one option, `tape` executes each option function in a specific order. Therefore, you can skip ahead a specified number of blocks, erase, and then rewind the tape all with the same command. The order of options executed is as follows:

Step 1. Get device name(s) from the `-z` option.

Step 2. Skip the number of tapemarks specified by the `-f` option.

Step 3. Skip the number of blocks specified by the `-b` option.

Step 4. Write a specified number of tapemarks.

Step 5. Erase a specified number of blocks of tape.

Step 6. Rewind the tape.

Step 7. Put the tape off-line.

For example, the following command skips four files on the `/mt0` device, erases the next two blocks, rewinds the tape, and takes the tape off-line:

```
tape -e=2 -f=4 -ro
```

The next example determines the block size of the device:

```
tape -s
```

The next example retensions the tape, rewinds it, and then takes it off-line:

```
tape -rot
```

Chapter 8: OS-9 for 68K System Management

System managers have a range of options to consider. OS-9 allows a system manager to tailor the system to the needs of users by changing system modules and setting up the system defaults. OS-9 also allows the system manager to maximize system performance by using RAM disks, making bootfiles, and making a startup file.

This chapter contains the following topics:

Setting Up the System Defaults: The Init Module

The `Init` module is sometimes referred to as the configuration module. It is a non-executable module located in memory in the `OS9Boot` file or in ROM. The `Init` module contains system parameters used to configure OS-9 during startup. The parameters set up the initial table sizes and system device names. For example, the amount of memory to allocate for internal tables, the name of the first program to run (usually either `SysGo` or `shell`), and an initial directory are specified. You can examine the system limits in the `Init` module at any time.

Note

The `Init` module MUST be present in the system in order for OS-9 to work.

The values in the `Init` module's table are the system defaults. You can change these defaults in two ways:

- Edit the `CONFIG` macro in the `systype.d` file. The `systype.d` file is located in the `DEFS` directory. After `systype.d` is edited, the `Init` module is remade and placed in the new bootfile.
- Modify the `Init` module with the `moded` utility.

This chapter covers both methods. Regardless of the method you use, the changes become the system defaults.

System Defaults Listed in the Init Module

The following is a list of the system defaults listed in the `Init` module. `Offset` refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and by linking the module with the relocatable library: `sys.l` or `usr.l`.

Table 8-1 System Defaults Listed in the Init Module

Offset	Name	Description
$30	Reserved	Reserved for future use.
$34	M$PollSz	**Number of Entries in the IRQ Polling Table** One entry is required for each interrupt generating device control register. The atomic kernel default is 16. The development kernel default is 32.
$36	M$DevCnt	**Device Table Size** The number of entries in the system device table. One entry is required for each device in the system. The atomic IOMan default is 8. The development IOMan default is 32.
$38	M$Procs	**Initial Process Table Size** Indicates the initial number of active processes allowed in the system. For Atomic OS-9, this table is fixed; for the development kernel, it automatically expands as needed. The atomic kernel default is 32. The development kernel default is 64.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$3A	M$Paths	**Initial Path Table Size** The initial number of open paths in the system. For Atomic OS-9, this table is fixed; for the development kernel, it automatically expands as needed. The atomic IOMan default is 32. The development IOMan default is 64.
$3C	M$SParam	**Offset to Parameter String for Startup Module** The offset to the parameter string (if any) to pass to the first executable module. An offset of zero indicates no parameter string is required. The parameter string itself is located elsewhere, usually near the end of the Init module.
$3E	M$SysGo	**First Executable Module Name Offset** The offset to the name string of the first executable module; usually SysGo or shell.
$40	M$SysDev	**Default Directory Name Offset** The offset to the initial default directory name string; usually /d0 or /h0. The kernel does a chd and chx to this device before forking the initial device. If the system does not use disks, this offset must be zero.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$42	M$Consol	**Initial I/O Pathlist Name Offset** This offset usually points to the /TERM string. This pathlist is opened as the standard I/O path for the initial process. It is generally used to set up the initial I/O paths to and from a terminal. This offset should contain 0 if no console device is in use.
$44	M$Extens	**Customization Module Name Offset** The offset to a name string of a list of customization modules (if any). A customization module is intended to complement or change OS-9's existing standard system calls. OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system state if found. Modules listed in the name string are separated by spaces. The default name string to search for is OS9P2. If there are no customization modules, set this value to 0. **NOTE:** A customization module may only alter the d0, d1, and ccr registers. **NOTE:** Refer to **Customization Modules** for more information on these modules.
$46	M$Clock	**Clock Module Name Offset** If there is no clock module name string, set this value to 0.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$48	M$Slice	**Time slice** The number of clock ticks per time slice. The number of clock ticks per time slice defaults to 2.
$4A	Reserved	Reserved for future use.
$4C	M$Site	**Installation Site Code Offset** This value is usually set to 0. OS-9 does not currently use this field.
$50	M$Instal	Offset to Installation Name
$52	M$CPUTyp	**CPU Type** CPU type: 68000, 68008, 68010, 68020, 68030, 68040, 68070, or 683XX. The default is 68000.
$56	M$OS9Lvl	**Level, Version, and Edition** This four byte field is divided into three parts: • level: 1 byte • version: 2 bytes • edition: 1 byte For example, level 1, version 3.0, edition 1 would be 1301.
$5A	M$OS9Rev	**Revision Offset** The offset to the OS-9 level/revision string.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$5C	M$SysPri	**Priority** The system priority at which the first module (usually SysGo or shell) is executed. This is generally the base priority at which all processes start. The default is 128.
$5E	M$MinPty	**Minimum Priority** The initial system minimum executable priority. The default is 0.
$60	M$MaxAge	**Maximum Age** The initial system maximum natural age. The default is 0.
$62	M$MDirSz	**Module Directory Size** The initial module count for the system. For the Atomic kernel this table is fixed; for the Development kernel, it automatically expands as needed. The default for the atomic and development kernel is 64.
$64	Reserved	Reserved for future use.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$66	M$Events	**Number of Entries in the Events Table** The initial number of entries allowed in the events table. For the Atomic kernel this table is fixed; for the Development kernel, it automatically expands as needed. The atomic kernel default is 16. The development kernel default is 32. See the ***OS-9 for 68K Technical Manual*** for a discussion of using events.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$68	M$Compat	**Revision Compatibility** This byte is used for revision compatibility. The default is 0. The following bits are currently defined: **Bit — Set Bit To:** 0 — Save all registers for IRQ routines. If you have OS-9 for 68K version 3.0 or greater, this flag is ignored. 1 — Prevent the kernel from using stop instructions. 2 — Ignore *sticky* bit in module headers. 3 — Disable cache burst operation (68030 systems). 4 — Patternize memory when allocated or de-allocated. 5 — Prevent kernel cold-start from starting system clock. 6 — Kernel ignores spurious IRQs. 7 — Only the process creating an alarm can delete it.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$69	M$Compat2	**Compatibility Bit #2** This byte is used for revision compatibility. The following bits are currently defined: **Bit Function** 0 0 External instruction cache is *not* snoopy.* 1 External instruction cache is snoopy or absent. 1 0 External data cache is *not* snoopy. 1 External data cache is snoopy or absent. 2 0 On-chip instruction cache is *not* snoopy. 1 On-chip instruction cache is snoopy or absent. 3 0 On-chip data cache is *not* snoopy. 1 On-chip data cache is snoopy or absent. 4 0 68349: cache/sram bank 0 is sram. 1 68349: cache/sram bank 0 is cache. 5 0 68349: cache/sram bank 1 is sram. 1 68349: cache/sram bank 1 is cache. 6 0 68349: cache/sram bank 2 is sram. 1 68349: cache/sram bank 2 is cache. 7 0 68349: cache/sram bank 3 is sram. 1 68349: cache/sram bank 3 is cache * snoopy = cache maintaining its integrity without software intervention.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$6A	M$MemList	**Colored Memory List Offset** The colored memory list contains an entry for each type of memory in the system. The list is terminated by a long word of zero. If this field contains a 0, colored memory is not used in this system. For a complete discussion on colored memory, see the ***OS-9 for 68K Technical Manual***.
$6C	M$IRQStk	**Size of Kernel's IRQ Stack** This field contains the size (in longwords) of the kernel's IRQ stack. The value must be 0 or between 256 and $ffff. If the value is 0, the kernel uses a small default IRQ stack. A larger IRQ stack is recommended. The default value is 256 longwords.
$6E	M$ColdTrys	**Retry Counter** The retry counter if the kernel's initial chd to the system device fails. The default value is 0.
$70	Reserved	
$72	Reserved	

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$74	M$CacheList	**Cache List Offset** The cache list entries describe alternate cache modes for user-state accesses to memory regions. The list is terminated by a long word of -1. If this field is 0, the cache lists are not used in the system. For a complete discussion on cache lists, refer to the ***OS-9 for 68K Technical Manual***.
$76	M$IOMan	**I/O Manager Module Name Offset** The offset to a name string of a list of I/O manager modules (if any). OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system-state if found. Modules listed in the name string are separated by spaces. The default name to search for is IOMan. If there are no I/O modules, set this to 0. **NOTE:** The I/O modules may only alter the d0, d1, and ccr registers. **NOTE:** Refer to **Customization Modules** for information about these modules.

Table 8-1 System Defaults Listed in the Init Module (continued)

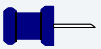
Offset	Name	Description
$78	M$PreIO	**Pre-I/O Module Name Offset** The offset to a name string of a list of Pre-I/O modules (if any). OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system-state if found. Modules listed in the name string are separated by spaces. The default name to search for is `PreIO`. If there are no Pre-I/O modules, set this to `0`. **NOTE:** The I/O modules may only alter the `d0`, `d1`, and `ccr` registers. **NOTE:** Refer to **Customization Modules** for information about these modules.

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$7a	M$SysConf	**System Configuration Flags** This word field is used for system configuration control. The following bits are currently defined. **Bit Function** 0 0 System tables are expanded as needed. 1 System table overflow results in an error. The default values in the table are set in the Init module. **NOTE:** System table expansion only applies to the Development kernel. For the Atomic kernel, table sizes are fixed from the Init module values. 1 Reserved 2 0 CRC checking performed by F$VModul. 1 CRC checking disabled for F$VModul. **NOTE:** CRC check disabling applies only to the Atomic kernel and only for checks made after cold start. 3 0 System-state time-slicing enabled. 1 System-state time-slicing disabled. 4 0 SSM builds user-state protection tables on a per-process basis 1 SSM builds one user-state page table (to allow access to all known memory) at cold-start. **NOTE:** This option only applies to the development kernel. The atomic kernel case always builds a single user-state page table. 5 -15 Reserved

Table 8-1 System Defaults Listed in the Init Module (continued)

Offset	Name	Description
$7c	Reserved	
$7e	M$PrcDescStack	**Size of Process Descriptor's Stack** This field determines the stack area size in a process descriptor. This stack is used by the process when it is performing system calls (for example, I/O operations). Systems with file managers/drivers using the stack heavily (for example, drivers written in C) may need to increase this value. The default is 1500 bytes.



Note

Throughout this chapter, the system directories referred to are the defaults found in the `Init` module, unless otherwise specified.

The following is a portion of the distributed `init.a` file:

```
_INITMOD equ 1 flag reading init module

CPUTyp  set 68000   cpu type (68008/68000/68010/etc.)
Level   set 1       OS-9 Level One
Vers    set 3       Version 3.0
Revis   set 0
Edit    set 0       Edition
IP_ID   set 0       interprocessor identification code
Site    set 0       installation site code
MDirSz  set 64      initial module directory count
PollSz  set 32      IRQ polling table size (fixed)
DevCnt  set 32      device table size (fixed)
Procs   set 64      initial process table size (divisible by 64)
Paths   set 64      initial path table size (divisible by 64)
Slice   set 2       ticks per time slice
SysPri  set 128     initial system priority
MinPty  set 0       initial system minimum executable priority
```

```
MaxAge  set 0       initial system maximum natural age limit
MaxMem  set 0       top of RAM (unused)
Events  set 32      initial event table size
Compat  set 0       version smoothing byte
Config  set 0       system configuration default
StackSz set 1024    IRQ Stack Size in bytes (must be 1k <= StackSz < 256k)
ColdRetrys set 0    number of retries for coldstart's "chd" before failing
* NOTE: for V3.0, NumSigs is unimplemented
NumSigs set 16      default queued signal maximum
PrcDescStack set 1500   default stack size in process descriptor
```

For More Information

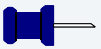
For more information on the `Init` module, see the ***OS-9 for 68K Technical Manual***.

Customization Modules

You can attach customization modules to OS-9 during the system's cold-start procedure to:

- Increase OS-9's functionality.
- Allow hardware customization such as special bus arbitration modes.

While customization modules extend its capabilities, OS-9 itself is not changed.



Note

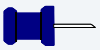
A customization module may only alter the `d0`, `d1`, and `ccr` registers.

In the `Init` module, the `M$Extens` offset points to a list of module names. By default, the module name in the list is `OS9P2`. If the modules are found during cold-start, they are called. If an error is returned, the system stops. The most commonly used modules are listed here:

`syscache`

The `syscache` module allows the system to enable and control any hardware caches present. The default `syscache` module supplied by Microware controls the on-chip cache(s) for the processor being used. You can customize this module to use any external (off-chip) cache hardware the system may have. The `syscache` module installs the `F$CCtl` system call routines. If you do not install the `syscache` module, no system caching takes place.

Standard `syscache` modules (to support the on-chip capabilities of the processor) are provided for the 68020, 68030, 68040 and 68349.



Note

External hardware caches are only supported by the development kernel. On-chip caching is supported by both the atomic and development kernels.

`FPU/FPSP`	`FPU` is used on all other processors when the system does not have a hardware floating point unit (for example, MC68EC040, MC68030 without MC68882 FPCP). The `FPU` module provides emulation support for the MC68882 floating point unit. `FPSP` is used on MC68040 systems (those with an on-chip FPU) and provides software emulation of MC68881/2 instructions not implemented on the MC68040.
`SSM`	The system security module (SSM) allows operation of the memory management unit (MMU) for the processor in use. For the development kernel, MMU operation under OS-9 provides basic user-state protection mechanisms for the system. This ensures user processes only use memory they are allowed to access. For the 68040 processors, the SSM module (in conjunction with the `CacheList` entries of the `Init` module) allows fine-tuning of the system memory's cache attributes. This allows for cache modes other than the default write-through mode (copy-back and non-cacheable regions). For the atomic kernel, user-state protection is not implemented. Thus, you should only use an SSM providing cache support (only the 68040 SSM) in an atomic kernel environment.

The standard SSM modules provided by Microware are:

`SSM451`	Supports the MC68451 MMU for 68010 systems. This SSM only provides protection functions.
`SSM851`	Supports the MC68851 PMMU and MC68030 MMU. Used on MC68020 and MC68030 systems. This SSM only provides protection functions.
`SSM040`	Supports the MC68040 MMU. Used on MC68040 systems. This SSM provides protection functions (if running in a development kernel environment) as well as cache mode support.

Changing System Modules

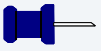
The provided system modules are configured to satisfy the needs of the majority of users. However, you may wish to alter the existing modules or create new modules. You can make new system modules and alter existing system modules by either:

- Using the `moded` utility, or
- Changing the defaults in the `systype.d` file.

The system modules most commonly altered are the device descriptors and the `Init` module.

Using the moded Utility

The `moded` utility allows you to edit individual fields of certain types of OS-9 modules. You can change the `Init` module and any OS-9 device descriptor modules with `moded`. Because `moded` is somewhat restrictive, if you are building a device descriptor or changing a field such as the file manager names, you may not want to use `moded`.



Note

The provided `moded.fields` file comes with module descriptions for standard RBF, SBF, SCF, PIPE, NETWORK, UCM, GFM, and `socket` module descriptors. It also includes a description for the `Init` module.

To use the `moded` utility, type `moded`, the name of the desired device descriptor, and any options. The `moded:` prompt shows you have entered the editor's command mode.

When `moded` is started, it attempts to read the `moded.fields` file. `moded.fields` contains module field information for each type of module to edit. Without this file, `moded` cannot function.

You can use the following commands from command mode:

Table 8-2 moded Utility Commands

Command	Description
`e(dit)`	Edit the current module.
`f(ile)`	Open a file of modules.
`l(ist)`	List the contents of the current module.
`m(odule)`	Find a module in a file.
`w(rite)`	Update the module CRC and writes to the file.
`q(uit)`	Return to the shell.
`$`	Call the OS-9 shell.
`?`	Print this help message.

Editing the Current Module

To edit the current module, use the `e` command. If there is no current module, the editor prompts for the module name to edit. The editor prints the name of a field, its current value, and prompts for a new value.

For More Information

For more information about `moded`, refer to the ***Utilities Reference*** manual.

You can enter the following edit commands:

Table 8-3 Edit Commands

Command	Description
`<expr>`	A new value for the field.
`-`	Re-display last field.
`.`	Leave edit mode.
`?`	Print edit mode commands.
`??`	Print description of the current field.
`<cr>`	Leave current value unchanged.

If the definition of any field is unfamiliar, use the `??` command. This provides a short description of the current field.

Exit Edit Mode

Once you have made all necessary changes to the module, exit the edit mode by either:

- Reaching the end of the module, or
- Typing a period.

At this point, the changes made to the module exist only in memory. To write the changes to the actual file, use the `w` command. This also updates the module header parity and CRC.

Editing the systype.d File

You can also change system modules by editing the systype.d file located in the DEFS directory. systype.d contains macros such as TERM and DiskH0 for each device descriptor and the Init module. These macros contain information such as basic memory map information, exception vector methods (for example, vectors in RAM or ROM), I/O device controller memory addresses, and initialization data for each device descriptor and the init module.

systype.d consists of five main sections used when installing OS-9:

- Init module CONFIG macro.
- SCF Device Descriptor macros and definitions.
- RBF Device Descriptor macros and definitions.
- ROM configuration values.
- Target system specific definitions.

The CONFIG macro is used when creating the Init module to determine six or more system dependent variables:

Table 8-4 System Dependent Variables

Name	Description
MainFram	A character string program, such as login, used to print a banner identifying the system. You can modify this string.
SysStart	A character string the OS-9 kernel uses to locate the initial process for the system. This process is usually stored in a module called SysGo. Two general versions of SysGo have been provided in the files: SysGo.a for disk-based OS-9 and SysGo_nodisk.a for ROM-based OS-9.
SysParam	A character string passed to the initial process. This usually consists of a single carriage return.

Table 8-4 System Dependent Variables (continued)

Name	Description
SysDev	A character string containing the name of the path to the initial system disk. The kernel coldstart routine sets the initial execution and data directories to this device before forking the SysStart process. Set this label to zero for a ROM-based system. For example, SysDev set 0.
ConsolNm	A character string containing the name of the path to the console terminal port. Messages to be printed during startup appear here.
ClockNm	A character string containing the name of the clock module.

You can set other system parameters in this macro to override the default values created by the init.a source file. This allows you to perform *system tuning* without modifying the generic init.a file.

The following is a portion of an example systype.d file:

```
CONFIG macro
   endm
   Slice set 10
   ifdef _INITMOD
Compat set ZapMem patternize memory
endc
```

When editing the Init module, constants may use either values or labels. CPUTyp set 68020 is an example of a constant using a value. These constants may appear anywhere in the systype.d file. Compat set ZapMem is an example of a constant using a label. These constants must appear outside the CONFIG macro and must be conditionalized to be invoked only when init.a is being assembled. If these values are placed inside the CONFIG macro, the old defaults are still used. If a constant requiring a label is placed outside the macro and

not conditionalized, `illegal external reference` errors result when making other files. You can use the `_INITMOD` label to avoid these errors.

The SCF and RBF device descriptor macro definitions are used when creating device descriptor modules. Five elements are common to SCF and RBF:

Table 8-5 SCF/RBF Device Descriptor Macro Definitions

Name	Description
`Port`	The address of the device on the bus. Generally, this is the lowest address the device has mapped. `Port` is hardware dependent.
`Vector`	The vector given to the processor at interrupt time. `Vector` is hardware/software dependent. You can program some devices to produce different vectors.
`IRQLevel`	The interrupt level (1 - 7) for the device. When a device interrupts the processor, the level of the interrupt is used to mask out lower priority devices.
`Priority`	The software dependent interrupt polling table priority . A non-zero `priority` determines the position of the device within the vector. Lower values are polled first. A `priority` of `0` indicates the device desires exclusive use of the vector. OS-9 does not allow a device to claim exclusive use of a vector if another device has already been installed on the vector, nor does it allow another device to use the vector once the vector has been claimed for exclusive use.
`DriverName`	The module name of the device driver. You determine the name used by the I/O system to attach the device descriptor to the driver.

RBF macros may also contain an optional sixth element to describe various standard floppy disk formats. These values are defined in the file `rbfdesc.a` in the `IO` directory.

SCF macros contain two additional elements:

- `Parity`
- `Baud Rate`

For More Information

The ***OS-9 for 68K Technical Manual*** defines the standard codes used by SCF.

The driver uses these values to determine the parity, word length, and baud rate of the device. These values are usually standard codes used by device drivers to access device specific index tables.

You should place definitions such as system specific control register definitions in `systype.d`. This allows you to maintain all system specific definitions in a single, system specific file.

Examine `systype.d`. If it does not accurately describe your system, use any text editor to edit the appropriate macro(s) in `systype.d`.

After editing the macro, change your data directory to the `IO` directory. Use the `make` utility to generate the required descriptors. For example, the `make d0` would generate the descriptors `d0` and `dd.d0`. The output files are placed in the `CMDS/BOOTOBJS` directory. Include these new descriptors in the bootfile.

For More Information

For more information about make, refer to **Chapter 6: The make Utility** and the ***Utilities Reference*** manual.

Making Bootfiles

A *bootfile* contains a group of modules to be loaded into memory during the system's bootstrap sequence. The provided bootfiles have been configured to satisfy the majority of users, but you may want to add or remove modules from an existing bootfile.

bootlist Files

Bootfiles are usually created using a `bootlist` file and the `-z` option of the `OS9Gen` or `TapeGen` utilities. The `bootlist` files contain a list of files, one file per line, to use in creating the bootfile. Using a `bootlist` file is a convenient way to maintain bootfile contents, as you can easily edit the `bootlist` file.

The `bootlist` files are usually located in the `CMDS/BOOTOBJS` directory, along with the individual files used for constructing the bootfile.

Bootfile Requirements

The contents and module order of a bootfile are usually determined by the end-user's system configuration and requirements. However, note the following points when you construct a bootfile:

- The kernel *MUST* be present in the system, either in ROM or in the bootfile. If the kernel is in the bootfile, *IT MUST BE THE FIRST MODULE*.
- The `Init` module must be present in the system, either in ROM or in the bootfile.

All other modules depend on the system configuration.

Making RBF Bootfiles

To make a bootfile for an RBF device (hard disk or floppy disk), you need to edit the `bootlist` file to match your requirements and then run the `OS9Gen` utility:

```
chd /h0/cmds/bootobjs
<edit bootlist file>
OS9Gen <device> -z=<bootlist>
```

Some systems may not support boot files greater than 64K in length and/or non-contiguous.

The `<device>` you specify is the disk on which you wish to install the bootfile. If this device is a hard disk, specify the *format-enabled* device name.

For example, to make a floppy-disk bootfile, type:

```
OS9Gen /d0 -z=bootlist.d0
```

To make a hard disk bootfile, type:

```
OS9Gen /h0fmt -z=bootlist.h0
```

Making Tape Bootfiles

To make a bootfile for an SBF device (tape), edit the bootlist file to match your requirements and then run the `TapeGen` utility:

```
chd /h0/cmds/bootobjs
<edit bootlist file>
TapeGen /mt0 -bz=bootlist.tape
```

Using the RAM Disk

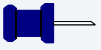
OS-9 provides support for RAM disks. These disks reside solely in Random Access Memory (RAM). You can access the information stored on a RAM disk significantly faster than the same information stored on a hard or floppy disk. You may store and access any files on a RAM disk.

To use a RAM disk, you must have:

- A device descriptor
- The RAM disk driver
- The RBF file manager

You may use multiple RAM disks as long as each RAM disk has a different port address. The only real limitation to the number of RAM disks is the size of the memory. However, some practical considerations exist. For example, using one large RAM disk is more efficient than using many small RAM disks.

In many system configurations, a RAM disk is used as the default system device. When the RAM disk is used as the default system device, it is known as device `dd`, instead of `r0`. The name of the device descriptor is `dd_r0`. Using this descriptor allows compilers to use the RAM disk as a *fast access* device for temporary files. The RAM disk is usually initialized at startup with definition and library files, if it is to be used as the default system device. The `init.ramdisk` procedure file provided in the `SYS` directory accomplishes this.



Note

RAM disks may be *volatile* or *non-volatile*.

- A volatile RAM disk disappears when the system is reset or the power is shut off.

- A non-volatile RAM disk resides in a place such as battery backed up RAM and does not disappear when the system is reset or powered down.

Volatile RAM Disks

Volatile RAM disks may be allocated memory either from free system memory or from outside free system memory. The port address controls the number of volatile RAM disks allocated from free system memory. There can be up to 1024 different disks, with each disk having a unique address from 0 to 1023.

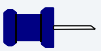
Volatile RAM disks not allocated from the free system memory must not be part of the system memory list, and they must have a port address greater than or equal to 1024. This port address indicates the actual start address of the RAM disk.

Non-Volatile RAM Disks

A non-volatile RAM disk may not be located in any memory search list known to the system's general memory lists (the RAM disk must be *outside* the system's knowledge). If it is located in a memory search list known to the system's general memory lists, the RAM disk may be wiped out because the memory is assumed to be unallocated and may later be given to another module. In addition, the format protect bit must be set for non-volatile RAM disks and the port address must be greater than or equal to 1024.

Making a Startup File

Using bootfiles is not the only way of loading modules and devices into memory at the time of startup. A startup procedure is executed each time OS-9 is booted and the standard `SysGo` is used. On disk-based systems, the startup procedure executes a `startup` file. The `startup` file is located in the root directory of the system disk.



Note

The `startup` file is an OS-9 procedure file. It contains OS-9 commands to be executed immediately after booting the system.

While you should load some modules and devices, such as the kernel, from the `bootlist` file, loading most modules and devices from the `startup` file can be advantageous. For example, it is easier to upgrade a system by adding modules to the `startup` file, or the files contained in the `startup` file. To change these files, you simply use a text editor and make the changes. To change the `bootlist` file, you must also use the `os9gen` utility.

Remember: A procedure file is made up of executable commands. Each command is executed exactly as if it were entered from the shell command line. Each line beginning with an asterisk (`*`) is a comment and is not executed.

From the root directory, you can examine the `startup` file by entering:

```
$ list startup
```

A listing similar to the following is displayed:

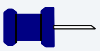
```
-t -np
*
* OS-9
* Copyright 1984 by Microware Systems Corporation
* Copyright 2001 by RadiSys Corporation
*
* The commands in this file are highly system dependent and should
* be modified by the user.
*
* setime; * start system clock
link shell cio csl; * make "shell", "cio", and "csl" stay in memory
```

```
load math; * load math module
* iniz r0 h0 d0 t1 p1 ; * initialize devices
* load -z=sys/loadfile ; * make some utilities stay in memory
* load bootobjs/dd.r0 ; * get default device descriptor
* init.ramdisk>/nil >>/nil & ; * initialize it if it's the ramdisk
* tsmon /t1 & ; * start other terminals
list sys/motd
```

The first executable line, `-t -np`, turns on the *talk mode* option of the shell and turns off the OS-9 prompt option for the duration of this procedure. The talk mode option echoes each executed command to the terminal display. This allows you to see what commands are being executed.

The other executable lines in the distributed `startup` file are followed by a comment explaining the command's purpose. Some standard commands are provided as comments. If you want to execute the command during the startup procedure, use a text editor to remove the asterisk preceding the command.

For example, to execute the `setime` command when the startup file is executed, remove the asterisk preceding the command.



Note

For systems with battery backed clocks, run `setime` to start timeslicing, but use the `-s` option. The date and time are read from the clock.

Initializing Devices

The `iniz r0 h0 d0 t1 p1` commented command initializes the following specific devices:

`r0`	RAM Disk
`h0`	Hard Disk
`d0`	Floppy Disk
`t1`	Terminal

`p1` Serial Printer

For More Information

For more information about `iniz`, refer to the ***Utilities Reference*** manual.

Whenever OS-9 opens a path to a device, it first checks to see if the device is *known* to OS-9. To be known, a device must be initialized and memory must be allocated for its device driver. If the device is unknown at the time of the request, OS-9:

- Initializes the device
- Allocates memory
- Opens the path

For example, a simple `dir /d0` command initiates this sequence of events if `d0` has not been previously initialized.

The `iniz` utility initializes devices. `iniz` performs an `I$Attach` system call on each device name passed to it. This initializes the device and links it to the system.

To initialize a device after the system has been started, type `iniz` and the name(s) of the device(s) to attach to the system. `iniz` goes through the procedure of initializing the device(s) and allocating the memory needed for the device. If the device is already attached, it is not re-initialized, but the link count is incremented.

For example, to increment the link count of modules, `t2` and `t3`, type:

```
$ iniz t2 t3
```

You can read the device names from standard input with the `-z` option or from a file with the `-z=<file>` option. To increment the link counts of devices listed in a file called `/h0/add.files`, type:

```
iniz -z=/h0/add.files
```

Closing a Path to a Device

You can use the `deiniz` utility to close a path to a device. `deiniz` checks the link count before removing the device from storage.

- If the link count is greater than one, `deiniz` lowers the link count.
- If the link count is one, `deiniz` lowers the link count, making it zero, and removes the device from the system device table. The device then becomes *unknown* to OS-9.

To use the `deiniz` utility, type `deiniz` followed by the name(s) of the devices(s) to remove from the system.

For example, to decrement the link count of module `p2`, type:

```
$ deiniz p2
```

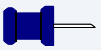
`deiniz` can read the device names from standard input with the `-z` option or from a file with the `-z=<file>` option. To remove the files listed in a file called `/h0/not.needed`, type:

```
$ deiniz -z=/h0/not.needed
```

For More Information

For more information about `iniz` and `deiniz`, refer to the ***Utilities Reference*** manual.

This initialize/de-initialize sequence can slow program execution and could cause memory fragmentation problems. To avoid these symptoms, Microware recommends all devices connected to the system at startup be `iniz`-ed in the `startup` file.



Note

You must place non-sharable devices in a bootfile to become known to the system. If a non-sharable device is `iniz`-ed, it is unusable because the link count has been incremented, causing it to appear to be in use.

`iniz`-ing the connected device at startup:

- Initializes the device.
- Allocates memory for its driver for the duration of the time the system is running, unless specifically `deiniz`-ed.

For example, a system with two floppy drives and one hard disk would `iniz` these devices in the startup file:

```
iniz h0 d0 d1 t1 p1 p
```

Loading Utilities into Memory

The next line of the `startup` file loads a number of utilities into memory. If a utility is not already in memory, you must load it into memory before using it. Pre-loading basic utilities at startup time avoids the necessity of loading the utility each time it is executed.

To load utilities into memory at startup, you must create a file containing the names of each utility to load, one utility per line. While the file may have any desired name, Microware recommends `loadfile` for obvious reasons. You can place this file in any directory as long as you specify its location on the command line. If `loadfile` were located in the `SYS` directory, the `startup` file command line is:

```
load -z=sys/loadfile
```

Previous versions of the OS-9 package had the following commented line in the startup file:

```
load utils
```

This method involved creating a `utils` file by merging the desired utilities into a single file in the commands directory. While you can still use this method, using `loadfile` is preferable because it uses less disk space and is easier to edit.

Loading the Default Device Descriptor

Many OS-9 compilers and application programs look for definition files and libraries in directories located on the default system device. The default system device is known as `dd`. `dd` may be defined as any disk device, but it is usually synonymous for one of the following devices:

`r0`	RAM Disk
`h0`	Hard Disk
`d0`	Floppy Disk

If you use a default device (`dd`) and the device descriptor is not in the bootfile, you must load the device descriptor. The next line in the `startup` file loads the device descriptor. The default device is the RAM disk named `r0`.

For More Information

RAM disks are covered elsewhere in this chapter.

If you want another device to be the default device descriptor, change the `.r0` extension to reflect the appropriate device. If you have a `dd` device in your bootfile or if no default device is to be used, leave this line as a comment.

Initializing the RAM Disk

If you are going to use the RAM disk, a library and definition file structure may be built on the RAM disk. The next line in the `startup` file executes the `init.ramdisk` procedure file. `init.ramdisk` is located in the root directory. It sets up `LIB` and `DEFS` directories on `/dd`. To name the RAM disk `/r0`, you must change a single line in `init.ramdisk`; change `chd /dd` to `chd /r0`.

Multi-User Systems

The `tsmon` utility is used to make your system a multi-user system. `tsmon` supervises idle terminals and initiates the login procedure for multi-user systems. The `startup` file command line: `tsmon /t1&` initiates the time-sharing monitor on the serial port `/t1`.

For More Information

Refer to the ***Utilities Reference*** manual for more information about:

- `tsmon`
- Each field in a password file entry (see the `login` utility).

`tsmon` can monitor up to 28 device name pathlists. Therefore, if you have multiple devices for `tsmon` to monitor, you can specify up to 28 devices on each `tsmon` command line. You can use the `ex` built-in shell command to execute `tsmon` without creating another shell. This conserves system memory. For example:

```
ex tsmon term t1 t2 t3 t4 t5&
```

When a carriage return is typed on any of the specified paths, `tsmon` automatically forks `login` and standard I/O paths are opened to the device.

The `login` procedure uses the `password` file located in the `SYS` directory for individual `login` validation. The provided `password` file has two example login entries. If `login` fails because you could not supply a valid user name or password, control returns to `tsmon`.

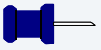
System Shutdown Procedure

There are times when, for one reason or another, you want to bring your system down. When you reset or power down your system, you may need to do more than just press the reset button. You need to gracefully shut down certain programs. For example, most network communications, print spoolers, and inter-system processes need special attention. These processes may have options or other arrangements you need to consider before shutting down your system.

In addition to taking care of processes requiring special attention, you should prepare the system's users for the shutdown. If at all possible, allow users enough time to save their files and get off the system. One way of alerting users the system is going down is by echoing a message using the `echo` and `tee` utilities. However, you should realize messages sent over the system in this manner are not seen by users who do not press a carriage return after the message has been sent. For example, if a programmer is sitting at a shell prompt, the message does not appear on the terminal screen until a carriage return is entered.

For More Information

For more information about `echo` and `tee`, refer to the ***Utilities Reference*** manual.



Note

Verbal warnings are important. This means in addition to sending a warning message out over the system, you may want to use either an intercom system or the telephone to talk to each person connected to the system.

You can simplify the process of actually shutting down your system by creating a procedure file. Once created, you can run the procedure either from the shell command line prompt or you can create a separate password entry for the sole purpose of shutting down the system.

For example, if you have a procedure file called `shutdown.sys`, you could create the following password file entry:

```
sys,shutdown,0.0,128,.,sys,shell shutdown.sys
```

Once you login as user `sys` with password `shutdown`, the shut down procedure begins because the system immediately has the shell execute the `shutdown.sys` file.

The following is an example of a procedure file you could use to shut down the system:

```
-t -nx -np
*
* System Shutdown Procedure
*
echo WARNING The system will shut down in 3 minutes ! tee /t1 /t2 /t3 /t4 /t5
sleep -s 60
echo WARNING The system will shut down in 2 minutes ! tee /t1 /t2 /t3 /t4 /t5
sleep -s 115
echo WARNING  5 seconds to system shut down ! tee /t1 /t2 /t3 /t4 /t5
sleep -s 5
spl -$; * terminate spooler
nmon /n0 -d         * shutdown network
sleep -s 3          * wait 3 seconds
break;              * call ROM debugger
```

The first six commands after the comment identifying the function of the procedure broadcast three warnings to the terminals on the system. The first warning tells the users the system is going down. The other two warnings serve as reminders. Remember you should also give verbal warnings.

The remaining command lines shut down the system:

Table 8-6 Shut Down Commands

Command	Description
`spl -$`	Terminate the spooler. All unfinished jobs are lost when the spooler is terminated.
`nmon /n0 -d`	Bring the network down. Users from other networks are no longer able to login to the system being shut down.
`sleep -s 3`	Cause the system to wait three seconds before executing the next command line. This allows the previous commands time to complete execution.
`break`	Send a `break` call to the ROM debugger. When the ROM debugger receives this call, the system shuts down.

Installing OS-9 on a Hard Disk

Once you have brought up the system and tested its basic operations, install OS-9 on the hard disk and use it as the system boot device. Installing the distribution software on the hard disk involves five steps:

Step 1. Checking the hard disk device descriptor.

Step 2. Formatting the hard disk.

Step 3. Copying the distribution software on to the hard disk.

Step 4. Making the hard disk the system boot disk.

Step 5. Test-booting from the hard disk.

Check the Hard Disk Device Descriptor

The installed hard disk may not necessarily match the parameters in the provided `/h0` and `/h0fmt` device descriptors. For example, the number of cylinders and heads for your hard disk may be different than the default parameters specified in the device descriptors. Before attempting to use the hard disk, carefully examine the disk macros in `systype.d`. If:

- the parameters match the drive in use, the supplied descriptors work.
- the parameters do not match the drive in use, edit `systype.d` and remake the descriptors or use the `moded` utility to remake the descriptors. `moded` makes/changes any device descriptor module and updates its CRC.

Once the descriptors are made, make a new bootfile with the new descriptors replacing the old ones.

Format the Hard Disk

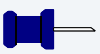
Once the descriptors match the type of drive in use, format the hard disk. Formatting the hard disk builds an OS-9 file structure on the media and tests the media for defective areas. Any new descriptors are also checked.

WARNING

If you have any vital information such as data or programs on this disk, you should perform backups to floppy or tape of this information. The format process completely erases any data on the disk.

To turn off page pause and format the hard disk, enter:

```
$ tmode nopause
$ format /h0fmt -c=<cluster size>
```



Note

`/h0fmt` *must* be the device name, as `/h0` is format protected. Use the `-c` option for large drives only.

For More Information

For more information about `format` and `tmode`, refer to the ***Utilities Reference*** manual.

The `format` utility asks whether you want to perform a physical format and a physical verify. Answer `y` to both questions. The physical format operation is a lengthy process. The larger your hard disk is, the longer you can expect to wait. The logical verify reads each cluster from the disk.

Copy the Distribution Software on to the Hard Disk

For More Information

For more information about `dsave`, refer to the ***Utilities Reference*** manual.

Once the hard disk has been formatted correctly, use the `dsave` utility to copy the distribution software on to the hard disk.

To copy the distribution files:

Step 1. Insert the first system disk in drive `/d0`. The first system disk contains the `CMDS` directory.

Step 2. Change your current data directory to `/d0`:

```
$ chd /d0
```

Step 3. Copy all files from `/d0` to `/h0`:

```
$ dsave -eb50 /h0
```

If you have more than one floppy disk to copy:

Step 1. Remove the disk in `/d0` and replace it with the new disk to copy.

Step 2. Change your execution directory to `/h0/CMDS`:

```
$ chx /h0/cmds
```

The hard disk is now your current execution directory.

Step 3. Copy all files from `/d0` to `/h0`:

```
$ dsave -eb50 /h0
```

Repeat this step until all floppy disks have been copied to the hard disk.

Making the Hard Disk the System Boot Disk

Copying files on to the hard disk installs the software on the hard disk. It *does not* make the hard disk a bootable disk. To make the hard disk the system boot disk, use the `os9gen` utility.

For More Information

For more information about `os9gen`, refer to the ***Utilities Reference*** manual.

The `OS9Boot` file is distributed with your system software. An `OS9Boot.h0` bootfile may also be included. The only difference between these files is the default system device name string in the `Init` module. `OS9Boot` refers to `/d0`, while `OS9Boot.h0` refers to `/h0`.

Assuming you have copied these files on to the hard disk, do the following to make the hard disk bootable:

Step 1. Change your current data directory to `/h0`:

```
$ chd /h0
```

Step 2. Rename `OS9Boot` to retain a copy to use with a floppy system:

```
$ rename OS9Boot OS9Boot.d0
```

Step 3. Make the hard disk bootable with the correct bootfile. You must specify `/h0fmt` as the device.

```
$ os9gen /h0fmt OS9Boot.h0
```

Test Boot from the Hard Disk

Once you have completed these steps, make sure the system actually boots from the hard disk.

If the system fails to boot correctly, reboot the system. Carefully examine the results of the actions previously described.

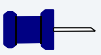
Managing Processes in a Real-Time Environment

The ability to manage processes in a real-time environment is one of OS-9's advantages. OS-9 has three main methods by which system managers can manage processes in a real-time environment:

1. Manipulating process' priority.
2. Using `D_MinPty` and `D_MaxAge` to alter the system's process scheduling.
3. Having system state processes and user state processes.

Manipulating Process' Priority

When you execute processes on the command line, you can change their initial priorities using process priority modifiers. This allows you to set the priority on crucial tasks higher so they run sooner and more often than less crucial processes.



Note

The initial priority is also a parameter for the fork and chain system calls.

For More Information

Process priority modifiers are covered in **Chapter 5: The Shell**.

Using D_MinPty and D_MaxAge to Alter the System's Process Scheduling

The D_MinPty and D_MaxAge system global variables can affect the way OS-9 schedules processes. D_MinPty and D_MaxAge are available to super users through the F$SetSys system call. These system variables can be used to affect the aging of processes.

Defining the Minimum Priority

D_MinPty defines a minimum priority below which processes are neither aged nor considered candidates for execution. Processes with priorities less than D_MinPty remain in the waiting queue and continue to hold any system resources they held before D_MinPty was set.

Note

D_MinPty is usually set to 0. All processes are eligible for aging and execution when this value is set to 0 because all processes have an initial priority greater than 0.

Remember: A process' initial priority is aged each time it is passed by for execution while it is waiting for CPU time.

If you have a critical process needing to be run and several other users have processes they want to run:

- Use the process priority modifier to increase the priority of the critical process.
- Set D_MinPty to a value less than the priority you assigned to the critical process but greater than the priority of the other processes.

The critical process now continues using the CPU until another process with a priority greater than D_MinPty is entered into the waiting queue or the critical process is finished.

For example, if D_MinPty is set to 500 and you set the priority of your process at 600, your process continues to use the CPU while processes with priorities less than 500 cannot run until D_MinPty is reset.

WARNING

D_MinPty is potentially dangerous. If the minimum system priority is set above the priority of all running tasks, the system completely shuts down and can only be recovered by a reset. It is crucial to restore D_MinPty to 0 when the critical task finishes or to reset D_MinPty or a process' priority in an interrupt service routine.

Defining the Maximum Age

D_MaxAge defines a maximum age over which processes are not allowed to mature. By default, this value is set to 0. When D_MaxAge is set to 0, it has no effect on the processes waiting to use the CPU.

When set, D_MaxAge essentially divides tasks into two classes:

Low priority	A low priority task is any task with a priority below D_MaxAge. Low priority tasks continue aging until they reach the D_MaxAge cutoff, but they are not executed unless there are no high priority tasks waiting to use the CPU.
High priority	A high priority task is any task with a priority above D_MaxAge. A high priority task receives the entire available CPU time, but it is not aged. When the high priority task(s) are inactive, the low priority tasks are run.

For example, if D_MaxAge is set to 2000 and three processes with initial priorities of 128 are in the active queue, the processes run just as if D_MaxAge had not been set. Then, if a process with an initial priority of 2500 is entered into the active queue, it receives CPU time when the

process currently in the CPU has finished. Once using the CPU, the high priority process runs uninterrupted until a process with a higher priority enters the active queue or the process finishes. When the process finishes executing, the low priority processes is again able to use the CPU.

Note

Any process performing a system call is not pre-empted until the call is finished, unless the process voluntarily gives up its time slice. This exception is made because these processes may be executing critical routines affecting shared system resources and could be blocking other unrelated processes.

Using System-State and User-State Processes

You can also use system-state processes to help manage real-time priority processing. System-state processes are processes running in a supervisor or protected mode. System-state processes basically have unlimited access to system memory and other resources. When a process in system state wants to use the CPU, it waits until it has the highest age.

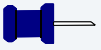
Processes in user state do not have access to all points in memory and do not have access to all of the commands.

When a process gains time in the CPU, it runs only for the time specified by the time slice. When it finishes using its time slice, it is entered back in the waiting queue according to its initial priority. To force system-state processes not to time-slice, set the appropriate bit in the `M$SysConf` flags of the `Init` module.

Using the tmode and xmode Utilities

The `tmode` and `xmode` utilities are also available to help you customize OS-9.

`tmode`	displays or changes the operating parameters of your terminal. `tmode` affects open paths, not the device descriptor itself, so the changes are temporary. The changes made by `tmode` are inherited if the paths are duplicated, but not if the paths are opened explicitly.
`xmode`	displays or changes the initialization parameters of any SCF-type device such as a video display, printer, or RS-232 port. `xmode` actually updates the device descriptor. The change persists as long as the computer is running, even if paths to the device are repetitively opened and closed. Some common uses of `xmode` are to change the baud rates and control definitions.



Note

`tmode` and `xmode` work only on SCF devices.

In SSM systems, you must have write permission for the descriptor module in order for `xmode` to work. You can use the `fixmod` utility to change the permissions.

Using the tmode Utility

To use the `tmode` utility, type `tmode` and any parameter(s) to change. If you give no parameters, the present values for each parameter are displayed. Otherwise, the parameter(s) given on the command line are processed. You can give any number of parameters on a command line. Use spaces or commas to separate each parameter.

If a parameter is set to zero, OS-9 no longer uses the parameter until it is reset to a code OS-9 recognizes. For example, the following command sets `xon` and `xoff` to zero:

```
tmode xon=0 xoff=0
```

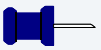
Consequently, OS-9 does not recognize `xon` and `xoff` until the values are reset.

To reset the values of a parameter to their default as given in this manual, specify the parameter with no value.

For More Information

The `tmode` parameters are documented in the ***Utilities Reference*** manual.

Use the `-w=<path#>` option to specify the path number affected. If a path number is not provided, standard input is affected.



Note

If you use `tmode` in a shell procedure file, you must use the `-w=<path#>` option to specify one of the standard paths (0, 1, or 2) to change the terminal's operating characteristics. The change remains in effect until the path is closed. To permanently change a device characteristic, you must change the device descriptor. You may alter the device descriptor to set a device's initial operating parameters using `xmode`.

Five parameters need driver support in order to be changed by `tmode`: `type`, `par`, `cs`, `stop`, and `baud`. If you try to change these parameters without driver support, `tmode` has no effect.

Using the xmode Utility

To use the `xmode` utility, type `xmode` and any parameter(s) to change. If you give no parameters, the present values for each parameter are displayed. Otherwise, the parameter(s) given on the command line are processed. You can give any number of parameters on a command line. Use spaces or commas to separate each parameter. You must specify a device name if the given parameter(s) are to be processed.

For More Information

The `xmode` parameters are documented in the ***Utilities Reference*** manual.

Like `tmode`, if a parameter is set to zero, the device no longer uses the parameter until it is reset to a recognizable code. To reset the values of parameters to their default, specify the parameter with no value. This resets the parameter to the default value as given in this manual.

Five parameters require further explanation: `type`, `par`, `cs`, `stop`, and `baud`. `xmode` changes these parameters only if the device is `iniz`-ed directly after the `xmode` changes and the driver supports these changes. Changing these parameters is usually done in the `startup` file or by first `deiniz`-ing a file. For example, the following command sequence changes the baud rate of `/t1` to 9600:

```
$ deiniz t1
$ xmode /t1 baud=9600
$ iniz t1
```

This type of command sequence changes the device descriptor and initializes it on the system. Only the five parameters mentioned above need this special sequence to be changed. All other `xmode` parameters are changed immediately.

The Termcap File Format

The `termcap` file is a text file containing control code definitions for one or more types of terminals. Each entry is a complete description list for a particular kind of terminal.

The first section of a termcap entry is divided into three parts. Each part is a different way of naming the terminal. A vertical bar `(|)` character separates the parts of a termcap entry. The three parts are:

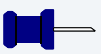
- A two character entry. This is a holdover from early UNIX editions.
- The most common name for the terminal. This name must contain no blanks.
- Long name fully describing the terminal. This name may contain blanks for readability.

For example:

```
kh|abm85h|kimtron abm85h:
```

You must set the `TERM` environment variable to the name used in the second part of the name section. In the following example, `TERM` is set to `abm85h`:

```
$ setenv TERM abm85h
```



Note

You can check the values stored in `TERM` by using the `printenv` command:

```
$ printenv
TERM=abm85h
```

The rest of the entry consists of a sequence of control code specifications for each control function. Use a colon `(:)` character to separate each item in the list. You can continue an entry on to the next

line by using a backslash (\) character as the last character of the line. It must appear after the last colon of the previous item. The next line must begin with a colon. For example:

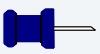
```
ka|amb85|kimtron abm85:\
:ct=\E3:  ...
```

Each item begins with a terminal *capability*. Each capability is a two character abbreviation. Each capability is either a boolean itself or it is followed by a string or a number. If a boolean capability is present in the termcap entry, then the capability exists on that terminal.

All numeric capabilities are followed by a pound sign (#) and a number. For example, the number of columns capability for an 80 column terminal could be described as follows:

```
co#80:
```

All string capabilities are followed by an equal sign (=) and a character string. You can enter a time delay in milliseconds directly after the equal sign (=) if padding is allowed in that capability. The padding characters are supplied by `tputs()` after the remainder of the string is sent to provide the time delay. The time delay may be either an integer or a real. The time delay may be followed by an asterisk (*) to specify the padding is proportional to the number of lines affected.



Note

It is often useful to specify the time delay using the real format. For example, the clear screen capability is specified as ^z with a time delay of 3.5 milliseconds by the following entry:

```
cl=3.5*^z:
```

You may indicate escape sequences by an \E to indicate the escape character. A control character is indicated by a circumflex (^) preceding the character. The following special character constants are supported:

```
\b   Backspace    ($08)
\f   Formfeed     ($0C)
\n   Newline      ($0A)
```

```
\r    Return        ($0D)
\t    Tab           ($09)
\\    Backslash     ($5C)
\^    Circumflex    ($5E)
```

You may specify characters as three Octal digits after a backslash (\). For example, if you must use a colon in a capability definition, it must be specified by \072. If you must place a null character in a capability definition use \200. C routines using termcap strip the high bits of the output, therefore \200 is interpreted as \000.

Termcap Capabilities

`termcap` recognizes the following termcap capabilities. Not all of these capabilities need to be present for most programs to use `termcap`. They are provided for completeness.

Note

(P) indicates you may optionally specify padding.

(P*) indicates the optional padding may be based on the number of lines affected.

Table 8-7 Termcap Capabilities

Name	Type	Padding	Description
`ae`	string	(P)	End alternate character set
`al`	string	(P*)	Add new blank line
`am`	boolean	(P)	End alternate character set
`as`	string	(P)	Start alternate character set
`bc`	string		Backspace if not `^H`
`bs`	boolean		Terminal can backspace with `^H`
`bt`	string	(P)	Back tab
`bw`	boolean		Backspace wraps from column 0 to last column

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
CC	string		Command character in prototype if terminal can be set
cd	string	(P*)	Clear to end of display
ce	string	(P)	Clear to end of line
ch	string	(P)	Horizontal cursor motion only, line stays same
cl	string	(P*)	Clear screen
cm	string	(P)	Cursor motion
co	numeric		Number of columns in line
cr	string	(P*)	Carriage return (default ^M)
cs	string	(P)	Change scrolling region (VT100), like cm
cv	string	(P)	Vertical cursor motion only
da	boolean		Display may be retained above
dB	numeric		Number of milliseconds of backspace delay needed
db	boolean		Display may be retained below
dC	numeric		Number of milliseconds of carriage return delay needed
dc	string	(P*)	Delete character

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
dF	numeric		Number of milliseconds of form feed delay needed
dl	string	(P*)	Delete line
dm	string		Delete mode (enter)
dN	numeric		Number of milliseconds of newline delay needed
do	string		Down one line
dT	numeric		Number of milliseconds of tab delay needed
ed	string		End of delete mode
ei	string		End of insert mode **NOTE:** if ic is used, enter :ec=:
eo	string		Can erase overstrikes with a blank
ff	string	(P*)	Hard copy terminal page eject (default ^L)
hc	boolean		Hard copy terminal
hd	string		Half-line down (1/2 linefeed)
ho	string		Home cursor (if no cm)
hu	string		Half-line up
hz	string		Hazeltime: cannot print tildas (~)

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
ic	string	(P)	Insert character
if	string		Name of file containing initialization string
im	boolean		Insert mode (enter). **NOTE:** If ic is specified use :im=:
in	boolean		Insert mode distinguishes nulls on display
ip	string	(P*)	Insert pad after character inserted
is	string		Terminal initialization string
k0-k9	string		Sent by other function keys 0-9
kb	string		Sent by backspace key
kd	string		Sent by down arrow key
ke	string		Take terminal out of *keypad transmit* mode
kh	string		Sent by home key
kl	string		Sent by left arrow key
kn	numeric		Number of other keys
ko	string		Termcap entries for other non-function keys

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
kr	string		Sent by right arrow key
ks	string		Put terminal in keypad transmit mode
ku	string		Sent by up arrow key
l0-l9	string		Labels on other function keys
li	numeric		Number of lines on screen or page
ll	string		Last line, first column (if no cm entry)
ma	string		Arrow key map
mi	boolean		OK to move while in insert mode
ml	string		Memory lock on above cursor
ms	boolean		OK to move while in standout or underline mode
mu	string		Turn off memory lock
nc	boolean		Carriage return down not work
nd	string		Non-destructive space
nl	string	(P*)	Newline character
ns	boolean		Terminal is a non-scrolling CRT
os	boolean		Terminal overstrikes

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
pc	string		Pad character (rather than null)
pt	boolean		Has hardware tabs
se	string		End stand out mode
sf	string	(P)	Scroll forwards
sg	numeric		Number of blank characters left by se or so
so	string	(P)	Begin stand out mode
sr	string	(P)	Scroll reverse
ta	string		Tab (other than ^I or without padding)
tc	string		Entry of terminal similar to last termcap entry
te	string		String to end programs using cm
ti	string		String to begin programs using cm
uc	string		Underscore one character and move past it
ue	string		End underscore mode
ug	numeric		Number of blank characters left by us or ue

Table 8-7 Termcap Capabilities (continued)

Name	Type	Padding	Description
ul	boolean		Terminal underlines but does not overstrike
up	string		Upline (cursor up)
us	string		Start underscore mode
vb	string		Visible bell
ve	string		Sequence to end open/visual mode
vs	string		Sequence to start open/visual mode
xb	boolean		Beehive terminal (f1=<esc>, f2=^C)
xn	boolean		Newline is ignored after wrap
xr	boolean		Return acts like ce \r\n
xs	boolean		Standout not erased by writing over it
xt	boolean		Tabs are destructive

Of the capabilities, the most complex and important capability is cursor addressing, cm. The string specifying the cursor addressing is formatted similar to the C function: printf(). It uses % notation to identify addressing encodings of the current line or column position. The line and the column being addressed could be considered the arguments to the cm string. All other characters are passed through unchanged. The following is the notation used for cm strings:

%d	A decimal number (origin 0)
%2	Same as %2d
%3	Same as %3d
%.	ASCII equivalent of value
%+x	Adds x to value, then %
%>xy	If value > x adds y, no output
%r	Reverses the order of row and column, no output
%i	Increments line/column (for 1 origin)
%%	Gives a single %
%n	Exclusive or row and column with 0140
%B	BCD (16*(x/10) + (x%10), no output
%D	Reverse coding (x-2*(x%16)), no output

The following examples illustrate the use of the preceding notations:

cm=6\E&%r%2c%2Y:	This terminal needs a 6 millisecond delay, rows and columns reversed, and rows and columns to be printed as two digits. The <esc>& and Y are sent unchanged. (HP2645)
cm=5\E[%i%d;%dH:	This terminal needs a 5 millisecond delay, rows and columns separated by a semicolon (;), and because of its origin of 1, rows and columns are incremented. The <esc>[, ; and H are transmitted unchanged. (VT100)
cm=\E=%+ %+ :	This terminal uses rows and columns offset by a blank character. (ABM85H)

Example Termcap Entries

```
ka|abm85|kimtron abm85:\
:ce=\ET:cm=\E=%+ %+ :cl=^Z:\
:se=\Ek:so\Ej:up=^K:sg#1
```

If two entries in the same termcap file are very similar, you can define one as identical to the other with certain exceptions. To do this, `tc` is used with the name of the similar terminal. This capability must be the last in the entry. All exceptions to the other terminal must appear before the `tc` listing.

If a capability must be cancelled, use `<cap>@`. For example, this might be a complete entry:

```
kh|abm85h|kimtron abm85h:\
:se=\EG0:so\EG4:tc=abm85:
```

Appendix A: ASCII Conversion Chart

ASCII Symbol Definitions

ASCII is an acronym for American Standard Code for Information Interchange. It consists of 96 printable and 32 unprintable characters. The following conversion table includes Binary, Decimal, Octal, Hexadecimal, and ASCII. The unprintable characters are defined below:

Table A-1 ASCII Symbol Definitions

Symbol	Definition
ACK	acknowledge
BEL	bell
BS	backspace
CAN	cancel
CR	carriage return
DC	device control
DEL	delete
DLE	data link escape
EM	end of medium
ENQ	enquiry
EOT	end of transmission
ESC	escape

Table A-1 ASCII Symbol Definitions (continued)

Symbol	Definition
ETB	end of transmission
ETX	end of text
FF	form feed
FS	file separator
GS	group separator
HT	horizontal tabulation
LF	line feed
NAK	negative acknowledgment
NUL	null
RS	record shipment
SI	shift in
SO	shift out
SOH	start of heading
SP	space
STX	start of text
SUB	substitute
SYN	synchronous idle

Table A-1 ASCII Symbol Definitions (continued)

Symbol	Definition
US	unit separator
VT	vertical tabulation

Table A-2 ASCII Characters

Binary	Decimal	Octal	Hex	ASCII
0000000	0	0	0	NUL
0000001	1	1	1	SOH
0000010	2	2	2	STX
0000011	3	3	3	ETX
0000100	4	4	4	EOT
0000101	5	5	5	ENQ
0000110	6	6	6	ACK
0000111	7	7	7	BEL
0001000	8	10	8	BS
0001001	9	11	9	HT
0001010	10	12	A	LF
0001011	11	13	B	VT

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
0001100	12	14	C	FF
0001101	13	15	D	CR
0001110	14	16	E	SO
0001111	15	17	F	SI
0010000	16	20	10	DLE
0010001	17	21	11	DC1
0010010	18	22	12	DC2
0010011	19	23	13	DC3
0010100	20	24	14	DC4
0010101	21	25	15	NAK
0010110	22	26	16	SYN
0010111	23	27	17	ETB
0011000	24	30	18	CAN
0011001	25	31	19	EM
0011010	26	32	1A	SUB
0011011	27	33	1B	ESC
0011100	28	34	1C	FS

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
0011101	29	35	1D	GS
0011110	30	36	1E	RS
0011111	31	37	1F	US
0100000	32	40	20	SP
0100001	33	41	21	!
0100010	34	42	22	"
0100011	35	43	23	#
0100100	36	44	24	$
0100101	37	45	25	%
0100110	38	46	26	&
0100111	39	47	27	’
0101000	40	50	28	(
0101001	41	51	29	)
0101010	42	52	2A	*
0101011	43	53	2B	+
0101100	44	54	2C	,
0101101	45	55	2D	-

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
0101110	46	56	2E	.
0101111	47	57	2F	/
0110000	48	60	30	0
0110001	49	61	31	1
0110010	50	62	32	2
0110011	51	63	33	3
0110100	52	64	34	4
0110101	53	65	35	5
0110110	54	66	36	6
0110111	55	67	37	7
0111000	56	70	38	8
0111001	57	71	39	9
0111010	58	72	3A	:
0111011	59	73	3B	;
0111100	60	74	3C	<
0111101	61	75	3D	=
0111110	62	76	3E	>

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
0111111	63	77	3F	?
1000000	64	100	40	@
1000001	65	101	41	A
1000010	66	102	42	B
1000011	67	103	43	C
1000100	68	104	44	D
1000101	69	105	45	E
1000110	70	106	46	F
1000111	71	107	47	G
1001000	72	110	48	H
1001001	73	111	49	I
1001010	74	112	4A	J
1001011	75	113	4B	K
1001100	76	114	4C	L
1001101	77	115	4D	M
1001110	78	116	4E	N
1001111	79	117	4F	O

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
1010000	80	120	50	P
1010001	81	121	51	Q
1010010	82	122	52	R
1010011	83	123	53	S
1010100	84	124	54	T
1010101	85	125	55	U
1010110	86	126	56	V
1010111	87	127	57	W
1011000	88	130	58	X
1011001	89	131	59	Y
1011010	90	132	5A	Z
1011011	91	133	5B	[
1011100	92	134	5C	\
1011101	93	135	5D	]
1011110	94	136	5E	^
1011111	95	137	5F	_
1100000	96	140	60	`

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
1100001	97	141	61	a
1100010	98	142	62	b
1100011	99	143	63	c
1100100	100	144	64	d
1100101	101	145	65	e
1100110	102	146	66	f
1100111	103	147	67	g
1101000	104	150	68	h
1101001	105	151	69	i
1101010	106	152	6A	j
1101011	107	153	6B	k
1101100	108	154	6C	l
1101101	109	155	6D	m
1101110	110	156	6E	n
1101111	111	157	6F	o
1110000	112	160	70	p
1110001	113	161	71	q

Table A-2 ASCII Characters (continued)

Binary	Decimal	Octal	Hex	ASCII
1110010	114	162	72	r
1110011	115	163	73	s
1110100	116	164	74	t
1110101	117	165	75	u
1110110	118	166	76	v
1110111	119	167	77	w
1111000	120	170	78	x
1111001	121	171	79	y
1111010	122	172	7A	z
1111011	123	173	7B	{
1111100	124	174	7C	|
1111101	125	175	7D	}
1111110	126	176	7E	~
1111111	127	177	7F	DEL

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X