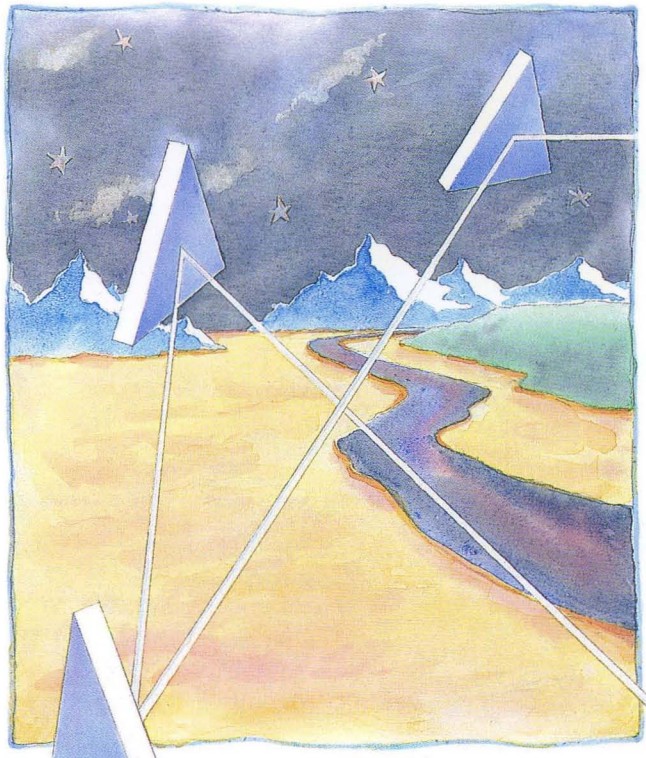


Using Your BSD Environment

011020-A00

apollo

Using Your BSD Environment

Order No. 011020-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July, 1988

This document was produced using the Interleaf Technical Publishing Software (TPS) and the InterCAP Illustrator I Technical Illustrating System, a product of InterCAP Graphics Systems Corporation. Interleaf and TPS are trademarks of Interleaf, Inc.

Preface

Using Your BSD Environment details the BSD environment, one of the operating environments supported by the Domain®/OS operating system. This manual is for users who are acquainted with both UNIX* software and Apollo® networks. If you're not familiar with the UNIX system, these tutorial references may be helpful:

- Bourne, Stephen W. *The UNIX System*. Reading: Addison–Wesley, 1982.
- Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*, Englewood Cliffs, N.J.: Prentice–Hall, 1984.
- Thomas, Rebecca and Jean Yates. *A User's Guide to the UNIX System*. Berkeley: Osborne/McGraw–Hill, 1982.

By now, you also should have read *Getting Started with Domain/OS* (002348), the beginner's guide to using BSD software on an Apollo node. Thus, you know how to use the keyboard and display, read and edit text, create and execute programs, and request system services using interactive commands. You'll need a working knowledge of these tasks to fully understand the concepts presented in this more advanced user's guide.

---

* UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

The manual is organized as follows:

Chapter 1	Introduces the Domain/OS operating system, describing how objects are organized in the system naming tree, and how to identify these objects.
Chapter 2	Describes the features of the BSD environment under Domain/OS.
Chapter 3	Discusses how the system functions at startup and login. Describes how to create, modify, and organize the various scripts that set up your node's particular operating environment. Also tells how to change your password, log-in shell, user information, and home directory.
Chapter 4	Explains the function of the Display Manager (DM), the default window management tool. Also describes how to use DM commands, and how to define keys to perform DM functions.
Chapter 5	Describes how to use the DM, the default window manager, to control your node's display.
Chapter 6	Describes how to use the DM to control the characteristics of edit pads and to edit text.
Chapter 7	Briefly introduces the shells available in the BSD environment.
Chapter 8	Explains how to use the C shell.
Chapter 9	Describes how to use the Bourne shell.
Chapter 10	Explains how to use the Korn shell.
Chapter 11	Describes file management, including procedures for creating, renaming, copying, comparing, removing, displaying, and printing files.

Chapter 12	Describes directory management, including procedures for creating, renaming, copying, comparing, removing, and displaying directories.
Chapter 13	Describes link management, including procedures for creating, displaying, redefining, renaming, copying, and removing links.
Chapter 14	Explains how to control access to files and directories on the system by using both standard UNIX file protection mechanisms and Domain/OS Access Control Lists (ACLs).

Related Manuals

The following file lists current titles and revisions for all available manuals:

/install/doc/apollo/os.v.*latest software release number***_manuals**

At SR10.0, e.g., refer to **/install/doc/apollo/os.v.10.0_manuals** to ensure that you are using the correct version of manuals. You may also want to use this file to ensure that you have ordered all of the manuals that you need. The *Domain Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) provide a complete list of related documents. For more information on using the BSD environment, refer to the following documents:

If you are a new user, read *Getting Started With Domain/OS* (002348). This tutorial manual explains how to log in and out, manage windows and pads, and execute simple commands. It gives user-oriented examples and includes a glossary of important terms.

The *Domain Display Manager Command Reference* (011418) contains information about the use of the default Display Management software. This manual is arranged for quick and easy access, and provides examples where necessary.

The *BSD Command Reference* (005800) describes all the UNIX shell commands supported in the BSD environment. This manual documents various general purpose, communications, and graphics commands and application programs. It also describes games available to the BSD user.

The *BSD Programmer's Reference* (005801) describes all UNIX system calls; C, standard I/O, mathematical, internet network, and compatibility library subroutines; special files; file formats and conventions; and language conventions supported in the BSD environment.

Domain/OS Programming Environment Reference (011010) describes the support tools and utilities available to BSD users. You may also need to consult the *Domain Distributed Debugging Environment Reference* (011024) if you plan to use Domain/OS debugging tools for your programming tasks.

UNIX Text Processing (011018) contains material on the text editors supported by the BSD environment. It also describes the available BSD text formatters, standard macro packages, and supported preprocessors.

Managing BSD System Software (010853) describes the tasks necessary to configure and maintain BSD system software services such as TCP/IP, line printer spoolers, and UNIX communications processing. Also explains how to maintain file system security, create user accounts, and manage servers and daemons. You may also wish to consult *Planning Domain Networks and Internets* (009916) and *Managing Domain/OS and Domain Routing* (005694) to learn more about creating and managing networks.

The *DOMAIN C Language Reference* (002093) describes C program development on Domain/OS. It lists the features of C, describes the C library, and gives information about compiling, binding, and executing C programs.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to

hardware, software, and documentation. By using this formal channel, you make it easy for us to respond to your comments.

See the **mkapr** (make apollo product report) command description in the *BSD Command Reference* for information about how to submit an APR. (You may also view the description online by following the procedure described in the next section of this preface.) Alternatively, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Getting Help

For information about available UNIX commands, system calls, and functions, press <HELP>. Then, at the prompt, type the *name* of the relevant command, system call, or function as follows:

```
Help on: name
```

This invokes the **man** (manual information) command, which lets you select and display on-line versions of reference material from the *BSD Command Reference*, the *BSD Programmer's Reference*, and *Managing BSD System Software*. A read window containing a formatted version of the manual page(s) on the specified *name* is opened and remains open until you close it by pressing <EXIT>. While the manual page is displayed, you may continue to execute shell commands (including other **man** commands).

> **NOTE:** The **man** command uses the symbolic links in effect for the SYSTYPE of the shell in which it is executed. See Chapter 2 for more on the SYSTYPE environment variable.

Documentation Conventions

Unless otherwise noted, this manual uses these symbolic conventions:

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user–supplied values	*Italic words or characters in formats and command descriptions represent values that you must supply.*
example user input	In examples, information that the user enters appears in color.
`output`	Information that the system displays appears in `this typeface`.
[ ]	Square brackets enclose optional items in formats and command descriptions.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the indicated key.
. . .	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or examples have been omitted.
——— ⊞ ———	This symbol indicates the end of a chapter.

Contents

Chapter 3 Understanding Startup and Login

Chapter 4 Using The Display Manager

Chapter 5 Controlling the Display

Chapter 6 Editing a Pad

Chapter 7 Introduction to Shell Usage

Chapter 8 Using the C Shell

Chapter 9 Using the Bourne Shell

Chapter 10 Using the Korn Shell

Chapter 11 Managing Files

Chapter 12 Managing Directories

Chapter 13 Managing Links

Chapter 14 Controlling Access to Files and Directories

Appendix A Initial Directory and File Structure

Appendix B Summary of Predefined Standard and UNIX Key Definitions

Figures

Tables

Chapter 1

Introducing Domain/OS

Domain/OS is an operating system which supports a high-speed communications network connecting two or more of our computers, called **nodes**. Each node loads programs into its own memory, and uses the computing functions of its own central processing unit (CPU). Because Domain/OS enables nodes to share information, you can log into any node and access information stored anywhere in the network.

Many of the operations you'll perform on the system involve the use of **objects** (files, directories, devices, and links) that store information such as programs, data, or text. Before you can work with these objects, you must understand how the system organizes and identifies them.

This chapter describes Domain/OS, how it organizes objects in the system naming tree, and how to use pathnames to identify these objects.

Overview

Domain/OS uses a physical network, in which member nodes can load data from the network into memory just as they would load data from their own disk. Let's take a look at how nodes use the system to share information. Figure 1–1 shows a simple network composed of three nodes and two disks.

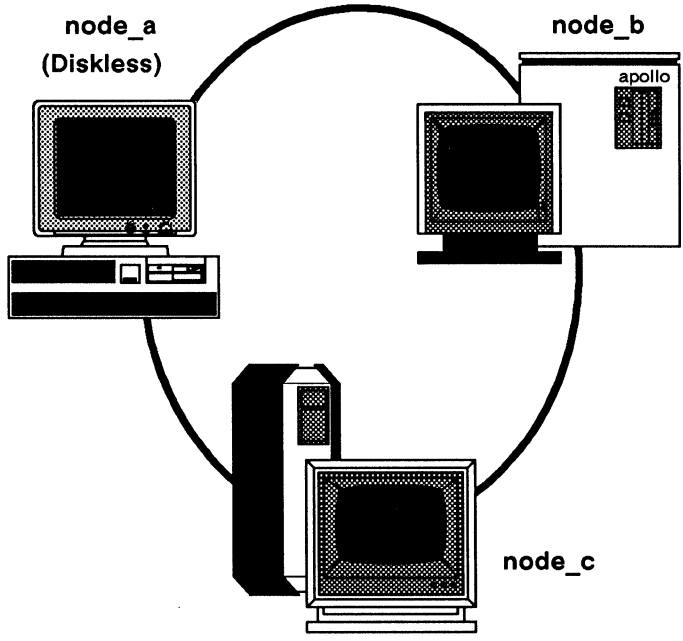


Figure 1–1. A Simple Domain/OS Network

Domain/OS makes the information on all disks available to any node in the network. For example, in Figure 1–1, **node_c** can access information stored on its own disk, as well as information stored on the disk connected to **node_b**. Although **node_a** doesn't have its own disk, it can, via the network, access information stored on the disks connected to **node_b** or **node_c**.

Each node in the network requires the use of at least one disk, called a **boot volume**, that contains the operating system and other system software it needs to run. Some nodes, called **disked nodes**, are physically connected to the disk that they use as the boot volume. Other nodes, called **diskless nodes**, share the boot volume of some other disked node in the network, called a **network partner**. In Figure 1–1, **node_b** and **node_c** are disked nodes. Because **node_a** is a diskless node, it must use either **node_b** or **node_c** as its partner.

To run in the network, a diskless node must have a network partner. The network partner's disk provides all of the necessary operating system and support software for the diskless node. Because a diskless node relies on its partner for system software, it can operate only when the partner node is operating. If the partner node is removed from the network while the diskless node is running, the diskless node will crash.

The user interface on each node, whether disked or diskless, is made up of two main programs: the **Display Manager (DM)** and the **shell**.

The DM is the system program that controls your node's display and enables you to create processes. The DM responds to DM commands that you type in the DM command input pad of your display. Later in this manual, we'll describe your node's display environment and explain how to use the DM to control this environment.

The shell is the program that you use to perform more traditional computing operations such as managing files and compiling programs. Three shells are available to the BSD user: the Bourne shell, the C shell, and the Korn shell. Each shell responds to commands that you type in the shell process's command input pad. Each command invokes a different utility program that performs a specific computing operation. This manual describes these shell programs and the shell commands you use to perform standard computing operations.

The Naming Tree

To make information available to all the nodes in the network, Domain/OS organizes objects in a hierarchical structure called a **naming tree**. The naming tree serves as a type of map that the system uses to keep track of where objects reside in the network. To access an object, you refer to its location in the naming tree. Figure 1–2 shows a sample naming tree.

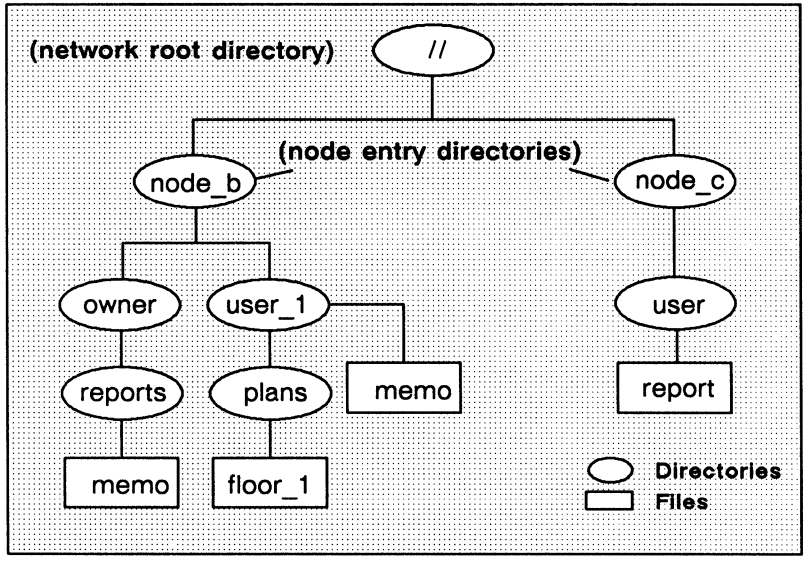


Figure 1–2. A Sample Naming Tree

The double slashes (//) in Figure 1–2 represent the top level of the naming tree, the **network root directory**. Each node maintains its own copy of the network root directory, which contains the name of each **node entry directory** the node can access. Figure 1–2 shows a network root directory containing the names of two node entry directories: **node_b** and **node_c**.

Each disked node in the network has a node entry directory name associated with it. This name refers to the branch of the naming tree that resides on its disk. (Since diskless nodes don't have disks, they use the node entry directory of their partner.) In Figure 1–2,

all of the objects under the node entry directory, **node_b**, reside on the disk **node_b**, while all of the objects under the node entry directory **node_c** reside on the disk **node_c**.

Entry directories contain one or more upper-level, or root-level, directories. A **root-level directory** is one level below the entry directory and normally serves as the main directory for a branch of logically related objects. For example, the /sys directory that we supply is a root-level directory that contains many of the system objects that make up the operating system. (Appendix A contains a set of figures that illustrate how the system organizes the software we supply with your node.) A root-level directory can also serve as a user's main directory for storing files.

In Figure 1-2, the directories **owner** and **user_1** are root-level directories, one level below the entry directory **node_b**. The directory **owner** serves as the main directory for all objects that belong to the owner of the node. The root-level directory **user_1** is the main directory for the user of a diskless node (**node_a**) that uses **node_b** as its entry directory. The directory **user** serves as the main directory for the user on **node_c**. (This is a custom only.)

In summary, the network root directory contains the names of node entry directories in the network. The system uses your node's network root directory to determine which node entry directories in the network it can access. Each node entry directory contains one or more root-level directories. A root-level directory serves as the main directory for a group of objects.

Your node can access only the node entry directories whose names appear in the local copy of the network root directory. To keep your local copy of the network root directory up to date, you should **catalog** new disked nodes as they are added to the network. To catalog new nodes, use the shell command **ctnode** (catalog node) described in the *BSD Command Reference*.

Some network sites use the **ns_helper** (naming server helper) to maintain a current network root directory. If this applies to your site, you needn't use **ctnode** to catalog nodes; **ns_helper** does it for you. Ask your system administrator for more information. *Managing BSD System Software* describes **ns_helper** and explains how to catalog nodes to update the network root directory.

Using Pathnames

The system identifies each object in the naming tree by its unique location. Whenever you specify a command to create or access an object, you also specify a **pathname** that points to the object's location in the naming tree. The pathname tells the system what path to follow when searching for an object.

The commands you use to create and manage objects require you to specify a pathname as a command argument. When you invoke a command, the command specifies the operation, and the pathname tells the system where in the naming tree to perform it.

For example, the following shell command removes the file **memo** in the naming tree shown in Figure 1–3:

```
% rm    //node_b/user_1/memo
  └┘   └──────────────────┘
  command       pathname
```

The shell command **rm** (remove file) tells the system to remove the file at the location specified by the pathname. Figure 1–3 shows the path the system follows to the file.

The pathname directs the system to:

1. Start at the network root directory (//).
2. Follow the path through the entry directory, **node_b**, and the subdirectory, **user_1**.
3. Stop at the file, **memo**.

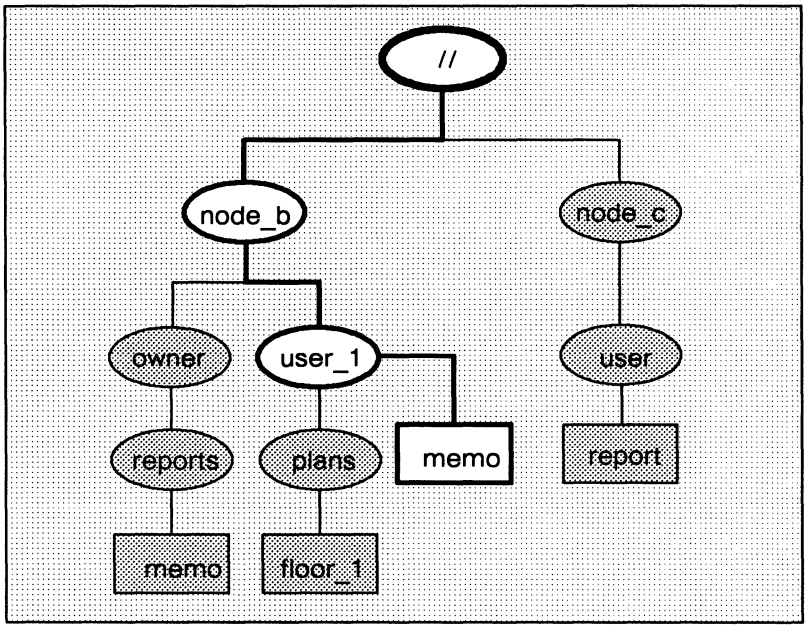


Figure 1–3. A Sample Path Through the Naming Tree

When the system searches for a location in the naming tree, it begins its search at some point in the tree and follows a path to the location. The pathname in the previous examples explicitly specified the network root directory as the starting point for the system's search through the naming tree. The double slashes (//) at the beginning of the pathname specify the network root directory. This type of pathname, called an **absolute pathname**, tells the system the full path, from the network root directory to the final location.

You don't have to begin pathnames with the network root directory specification. For example, the single slash (/) symbol directs the system to begin its search at your node's entry directory. Here is an example using the single slash to start a search at your node's entry directory:

```
% rm /user_1/memo
```

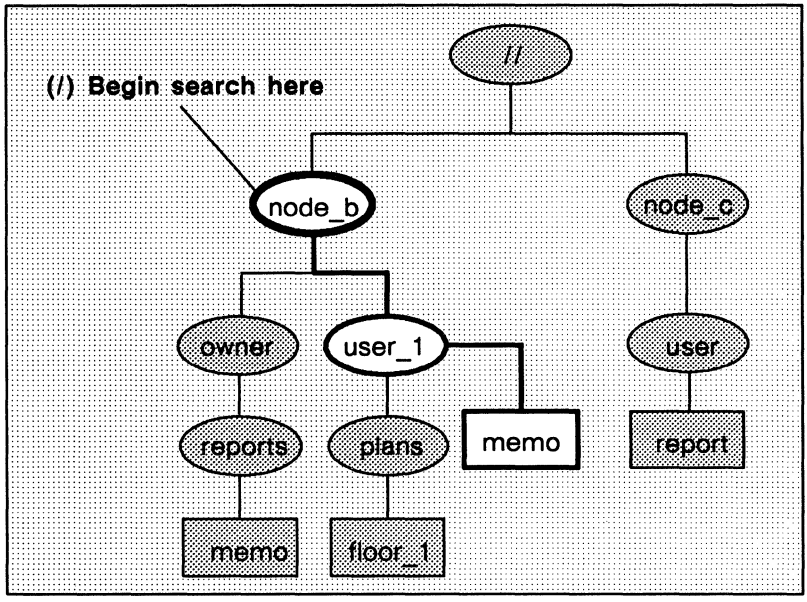


Figure 1–4. A Sample Path Beginning at the Node Entry Directory

For this example, let's assume that your node's entry directory is **node_b**. As shown in Figure 1–4, the pathname directs the system to:

1. Start at your node's entry directory, **node_b**.
2. Follow the path through the root–level directory, **user_1**.
3. Stop at the file, **memo**.

You can specify other starting points in the naming tree by beginning a pathname with any of the symbols in Table 1–1.

Table 1-1. Pathname Symbols

Symbol	System starts search at:
//	Network root directory
/	Node entry directory
No symbol or .	Working directory
~	Home directory
..	Parent directory

The Working Directory

If you specify a pathname without a symbol preceding it, or precede it with a dot, the system starts its search at a default location in the naming tree called the working directory. Think of the **working directory** as the directory location in which you are currently working (thus, it may also be known as your **current directory**). Each process that you create uses one of the directories in the naming tree as its working directory.

When you log into a node, the system creates a process running the shell program and sets that process's working directory to the **home directory** name designated in your user account. The system uses this directory as your working directory unless you change it to another directory. (Chapter 12 describes how to change your working directory.)

The following command removes the file **memo** in the current working directory:

```
% rm memo
```

In this example, let's assume that the current working directory is the directory **reports**. As shown in Figure 1-5, the system begins its search at **reports** and removes the file **memo**.

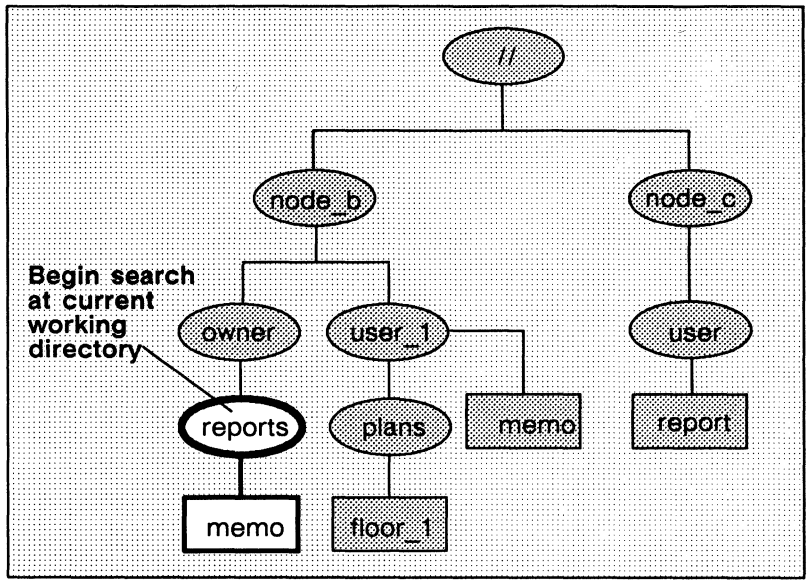


Figure 1–5. A Sample Path Beginning at the Current Working Directory

In Figure 1–5, another file named **memo** exists at another location in the naming tree (in the directory **user_1**). If the current working directory was **user_1** instead of **reports**, the command in our example would remove this file instead. So you see, a pathname that starts at the working directory functions differently depending on the directory currently being used as the working directory.

The Home Directory

If you begin a pathname with the **tilde** symbol (~), the system starts its search at a location in the naming tree called the **home directory**. Like the working directory, each process has a home directory that points to some directory in the naming tree.

When you log into a node, the system creates a process running the shell program and sets it to the home directory name designated in your user account. The system uses this directory as your home directory unless you change it to another directory. (Chapter 3 describes how to change your home directory.)

The following command removes the file **memo** in the directory **reports** found in the home directory:

```
% rm ~/reports/memo
```

In this example, let's assume that the home directory is the root-level directory **owner**. As shown in Figure 1–6, the pathname directs the system to:

1. Start at your home directory, **owner**.
2. Follow the path through the directory, **reports**.
3. Stop at the file, **memo**.

Like pathnames that use the current working directory, pathnames starting at the home directory work differently depending on the directory currently being used as your home directory.

Note that a tilde with no pathname given as an argument defaults to the current user's home directory.

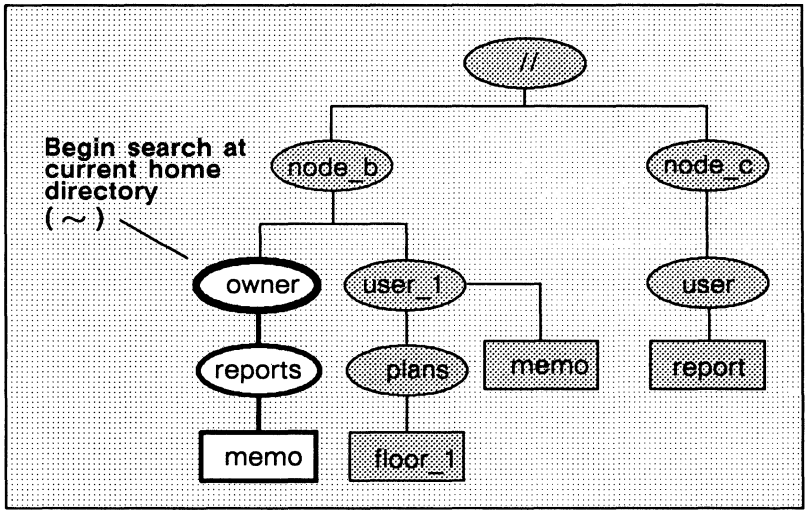


Figure 1–6. A Sample Path Beginning at the User's Home Directory

The Parent Directory

If you precede the pathname with two dots (..), the system starts its search at a location called the parent directory. A **parent directory** is the directory one level above the current working directory. For example, the following command uses the double dot symbol to remove the file **memo** in the directory **user_1**:

```
% rm ../memo
```

In this example, let's assume that the current working directory is the directory **plans**. As shown in Figure 1–7, the system begins its search at the directory **user_1** (the parent directory of the current working directory **plans**) and removes the file **memo**. It is important to note that these double dots can be strung together with slashes (e.g., *../../filename*) to search the parent's parent directory and so on.

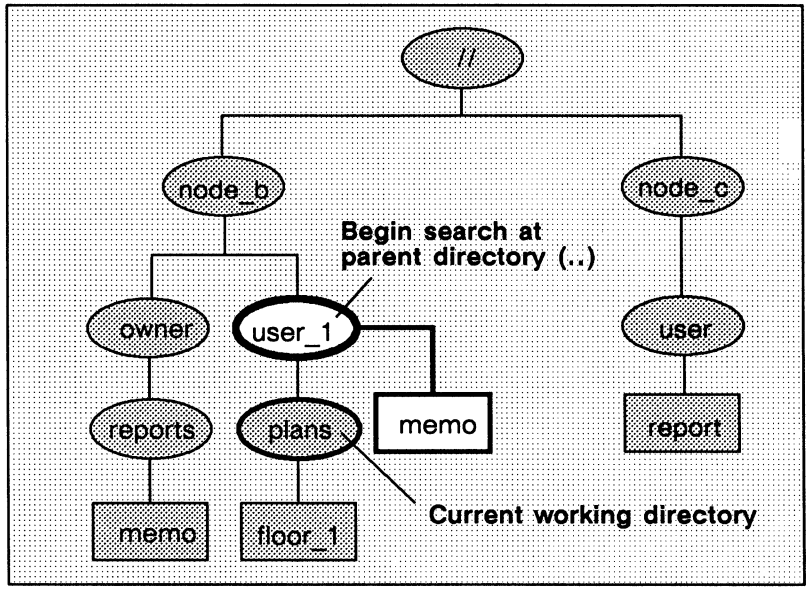
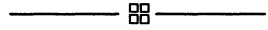


Figure 1–7. A Sample Path Beginning at the Parent Directory

Pathname Summary

In this section, you learned how to use pathnames to point to objects in the system naming tree. The examples showed you how to use pathnames with commands to tell the system the naming tree location where you want a particular operation performed.

Pathnames also serve to identify objects. As you read through this manual, you will find that many of the objects that make up the operating system are referred to by their pathnames. For example, Chapter 3 describes many of the objects the system uses at startup and login. Appendix A illustrates how the system organizes the system software that we supply with your node; system objects are referred to by their pathnames. By understanding which objects the system uses and where they are located, you'll better understand how these objects work together to make up a functioning system.



Chapter 2

Using Domain/OS Features in the BSD Environment

The BSD environment under Domain/OS supports a distributed file system, and multiple networks using bit-mapped, high-resolution displays. Besides bringing the benefits of a networked architecture and a true single-level store to a UNIX system, Domain/OS offers many features seldom found on either time-sharing or workstation implementations of software. This chapter highlights those features.

Domain/OS Architecture

Domain/OS architecture comprises two or more nodes connected by a high-speed local area network. Each node is a functional workstation, with its own central processor, memory, and memory management hardware. Programs and data required by processes running on a node may be demand-paged across the network.

This remote paging ability means, for example, that a process running on one node can invoke a program that resides on the disk of another node to manipulate data that reside on a third node. You may even create remote processes (processes that run on other nodes in the network) that you can manipulate through a window on your node, thus distributing the computational workload over multiple processors.

Those nodes that have their own mass storage devices may be operated as standalone computers, and can support additional users (including those connected via serial communications ports). To take advantage of this networked architecture, all Domain/OS software supports a distributed file system. Data and programs on all mounted volumes in the network are accessible (given the necessary permissions) to any node in the network. The resultant system is one in which an arbitrary number of users can be serviced without adversely affecting performance. Users have the power of a dedicated processor, memory–management hardware, and a high–resolution bitmapped display at their disposal.

The User Interface

We provide for a more varied user interface by supplying features that significantly differ from those provided in other UNIX implementations. The most important difference, from the user's point of view, is the ability of an Apollo node to display "windows" into many processes (shells, programs, etc.). These windows have some unique features not found on the CRT terminals largely used in the development of UNIX software.

Software Extensions in /usr/apollo

BSD provides a directory called **/usr/apollo**. The **/usr/apollo** directory contains the subdirectories **bin**, **lib**, and **include**, which supply software extensions beyond the standard set normally found on the Berkeley 4.3 distribution tape. The directory **/usr/apollo/bin** contains commands, **/usr/apollo/lib** contains object libraries and needed files, and **/usr/apollo/include** contains **.h** files.

NOTE: Normally, users don't refer to the pathname **/usr/apollo/include** directly, but rather use **/usr/include/apollo**, which is a soft link to **/usr/apollo/include**. This specific feature allows use of the notation **# include <apollo/ev.n>**.

The commands, libraries, and include files in these subdirectories handle functions that specifically apply to Domain/OS. (Note, however, that some additional related files can be found in the **/etc** directory.) For example, there are special network commands, commands for manipulating windows and displays, and commands for doing disk volume maintenance.

In most cases, the commands found in **/usr/apollo/bin** follow the conventions of other standard UNIX commands. However, there may be some exceptions when it comes to command line options or arguments. Be sure to check the appropriate manual page in the *BSD Command Reference* for complete information before using these software extensions.

The Display and the Display Manager

Your node's display is your "window" into Domain/OS. Unlike most terminals that dedicate their entire display to a single program or process, Apollo nodes let you divide the display screen into multiple environments for running programs, and reading or editing files. With each new environment you create, a set of display components through which you can enter input and view output are also generated.

What you see through a window is either a "frame" containing graphics or a "pad" containing text. Refer to the *Domain Display Manager Command Reference* for more information about frame mode and graphics.

Keyboard Mapping

On Apollo nodes, nearly all key binding is programmable. The DM normally binds the keys to a default function map when you log in. Although you can change these key bindings any time, it is usually best to begin with the default bindings, and then customize your key definitions as needed. For more information on the DM and keyboard mapping, see the *Domain Display Manager Command Reference*.

Domain/OS supports three types of keyboards: the Low-Profile Model I keyboard, the Low-Profile Model II keyboard, and the Multinational keyboard.

The directory **/sys/dm** contains the command files that define each type of keyboard;

- **std_keys3** keyboard definitions for the Low-Profile Model II keyboard
- **std_keys2** keyboard definitions for the Low-Profile Model I keyboard
- **std_keys3[a-g]** keyboard definitions for the Multinational keyboard

All of these files contain a line invoking the command file **std_keys.basic**. The Multinational keyboard command files also invoke the **std_keys.mn** file.

UNIX Key Definitions

An alternate version of the standard key definitions, modified to provide necessary UNIX functions, resides in the **/sys/dm** directory. This alternate version is named **std_keys.unix**.

The **std_keys.unix** definition file includes commands that bind various keys to certain version-specific (or shell-specific) features. They are described in detail later in this manual, in the Bourne shell and the C shell chapters. If your environment is set to one of the UNIX environments, these key definitions files are automatically invoked.

To put any key definition file into effect, execute the **cmdf** (command file) command at the Display Manager prompt, where the filename argument is one of the key definitions files mentioned earlier. For example, to invoke UNIX key definitions on a keyboard:

```
Command: cmdf /sys/dm/std_keys.unix
```

Table 2-1 shows which keys are redefined when the keyboard is remapped to **std_keys.unix** in this manner.

Table 2-1. Keys Remapped in std_keys.unix

Key	Definition
<DELETE>	Deletes a character.
<HELP>	Gets a specified UNIX manual page.
<SHELL>	Executes the DM command **cp $(shell)** which creates a BSD shell as specified by the **$SHELL** environment variable.
<TAB>	Inserts a literal ASCII tab character.
CTRL/C	Generates an interrupt signal.
CTRL/D	Produces an end-of-file (EOF) condition in the input pad.
CTRL/H	Deletes a character.
CTRL/I	Generates a literal ASCII tab character.
CTRL/J	Performs a carriage return.
CTRL/L	Redraws the screen.
CTRL/M	Performs a carriage return.
CTRL/N	Searches for next occurrence of pattern.
CTRL/O	Flushes output (not implemented).
CTRL/P	Searches for previous occurrence of pattern.
CTRL/Q	Turns off hold mode in the window.
CTRL/R	Does nothing. (The standard UNIX functionality is irrelevant in a pad.)

(Continued)

Table 2-1. Keys Remapped in std_keys.unix (Cont.)

Key	Definition
CTRL/S	Turns on hold signal.
CTRL/U	Deletes a line of input text from the cursor to the start of the line.
CTRL/Y	Suspends when read (not implemented).
CTRL/Z	Produces a suspend process signal normally used by shells that support job control (i.e., **/bin/ksh**, **/bin/csh**.
CTRL/	Generates a quit signal.
CTRL/~	Moves to previous window.

Environment Variables

UNIX users should be familiar with the concept of environment variables, process-wide ASCII strings that assume the general form

name = value

Environment variables are maintained by the kernel's process manager and are made available to UNIX programs. Typically, you initialize these variables in one of the command files that the window manager reads when the node is booted, and later when you log in.

When a new process is created, all environment variables of the creating process are inherited by the new process. All process creation mechanisms (e.g., **pgm_$invoke**, **fork**, **vfork**) provide for this inheritance.

When a new process is created by the Display Manager, that process inherits all environment variables from the current context process. The DM also inherits environment variables when **cv** (read file) and **ce** (edit file) are used.

Environment variables defined in the DM startup file are inherited by all server processes created during DM startup, and by the first process you create at login.

> **NOTE:** After the first user process is created, the DM inherits environment variables from the current context process (and passes them to new processes) as described above.

A program interface for environment variable usage is defined in the **/usr/include/apollo/ev.h** files. C language programs may manipulate environment variables through these interfaces. Alternatively, C programs may use the UNIX calls **getenv** and **putenv** or access the external environment variable.

Certain environment variables are well-known. Some variables are predefined by the system at login; others have special significance to system software or other special attributes.

One such environment variable determined at login time is the SYSTYPE environment variable, which specifies the default UNIX version running on a node. The UNIX version acts as a modifier of the environment in which programs execute on the node. Valid SYSTYPES are **bsd4.3** (4.3 Berkeley Software Distribution) and **sys5.3** (System V, Release 3).

The **/etc/environ** file contains a line that specifies the SYSTYPE for a node; this, in turn, helps determine the default log-in shell for the user. (See Chapter 3 for further information on this.) To display or change the SYSTYPE used to execute programs from a UNIX shell, use the **ver** (version) command as shown in the "Environment Switching" section later in this chapter.

Table 2-2 shows the entire set of environment variables used by the BSD Bourne shell.

Table 2–2. Environment Variables Used by the BSD Bourne Shell

Variable Name	Description
USER	User's login name.
LOGNAME	Synonomous with USER. The synonyms are provided to support both the SysV and BSD environments.
PROJECT	Project (group) ID under which the user logged in.
ORGANIZATION	The current Apollo organization ID for the user.
NODEID	The unique node identifier for the node on which the process is running; expressed in hexadecimal.
NODETYPE	The type of node on which the process is running.
HOME	The user's home directory pathname, established at login.
TERM	The device name of the "terminal" in use; predefined for the sake of C or UNIX programs. Values for Apollo displays are of the form **apollo_***xxx* where *xxx* is three or more characters. The file **/etc/termcap** lists valid terminal types.
SHELL	The pathname of the shell in which the process is running (in this case, **/bin/sh**).
TZ	The timezone string. Like TERM, this variable is predefined for the sake of C or UNIX programs. The valid format is *SSSnDDD*, where *SSS* is the standard timezone name (e.g., EST), *n* is the difference between the standard timezone and UTC, and *DDD* is the daylight timezone name.
SYSTYPE	UNIX system version in use (i.e., bsd4.3).
ENV	If ENV is part of the environment at shell startup or is set on the **sh** command line (**-DENV=~/.shrc**), the value is used as a pathname to a shell startup script. This is the same as the Korn shell ENV variable.

Name Space Support

The UNIX file system has traditionally contained a small number of system directories with well-known names (**/usr**, **/bin**, **/etc**, **/dev**, and **/tmp**). The structure and content of these directories differ between versions of UNIX software. To support identically-named AT&T and Berkeley versions of these directories on the same Domain/OS file system, we use "variant" links. These links allow a portion of the link text to be replaced by an environment variable.

Symbolic links placed in your node's root directory during the installation procedure let programs use either the **sys5.3** or **bsd4.3** versions of the **/bin**, and **/usr** directories (**/tmp**, **/etc**, and **/dev** are common to both). Normally, the links are created by the installation script; if, at some time, you need to re-create them, use the **ln** (make links) command. For example, to create a SYSTYPE-dependent link for **/bin**, type the following command line:

```
% ln -s '/$(systype)/bin' /bin
```

> **NOTE:** Single quotes around link text are required, to keep the dollar sign from being interpreted as a shell metacharacter.

The SYSTYPE environment variable is used to select the UNIX file system variant, and therefore, commands, libraries, spool directories, etc. Table 2-3 shows top-level BSD directory organization.

Table 2-3. Top-Level BSD Directory Organization

Name	Object Type	Contents
/usr	directory	**bin, lib, include, apollo, apollo/bin, ucb, man**
/bin	variant link	-
/etc	directory	SYSTYPE-specific links and files
/dev	ordinary link	-
/bsd4.3	directory	**usr, bin, etc**
/tmp	ordinary link	-

NOTE: In Table 2–3, ordinary links are those that don't contain the name of an environment variable. In the case of **/dev** and **/tmp**, these should be links to your node's **'node_data/dev** and **'node_data/tmp** files respectively.

Each node's **/tmp** directory is usually a link to **'node_data/tmp**. One of the less obvious side effects of this can be easily illustrated. For example, the following two command lines executed on node **//foo** both list the contents of **//foo**'s **'node_data/tmp** directory:

```
% ls /tmp
cattoc                  ipc.out       toc143
% ls //foo/tmp
cattoc                  ipc.out       toc143
%
```

To list the contents of **//bar/tmp**, you need to be more explicit:

```
% ls //bar/sys/node_data/tmp
dirs     ln
%
```

Environment Switching

The object–module stamping scheme, described earlier, lets you execute SysV programs from any BSD shell and vice versa, without any knowledge of the UNIX version for which the program was targeted.

When you invoke a program stamped with a **systype** other than **any**, the SYSTYPE environment variable for the process in which the program is running is set to the value found in the object module. This ensures that programs of one UNIX version that depend on certain system files continue to work when executed from a process running in another version. The **/usr/apollo/bin/systype** program displays the version stamp of the specified object files.

A shell's SYSTYPE value defines the version (**bsd4.3, sys5.3**) of system directories searched when a command name is given; hence, it defines the version of the command that is executed.

To simplify the execution of a version *x* command from a version *y* shell, we provide a "set-version" command. See the **ver** (change shell command version) command in the *BSD Command Reference*. You can use **ver** in these three ways:

- To display the current value of SYSTYPE, execute **ver** with no arguments, as shown in the following example:

  ```
  % ver
  bsd4.3
  ```

  This is equivalent to typing the following:

  ```
  % echo $SYSTYPE
  ```

- To change the value of SYSTYPE, and the version of subsequently executed commands, use the form **ver** *value*. For example, you may do the following:

  ```
  % ver sys5.3
  % ls
  prog.c
  prog.o
  testfile
  %
  ```

  Here, the first command line sets the SYSTYPE to **sys5.3**. This is equivalent to typing

  ```
  % setenv SYSTYPE=sys5.3
  % rehash
  ```

  The second command line executes a **sys5.3** version of **ls** (SYSTYPE remains the same until it is reset).

- To execute the *value* version of *command* without changing SYSTYPE, use the form **ver** *value command*. For example, the first command executes the **sys5.3** version of the **man** (print manual page), searching for the manual page on the **ls** (list directory) command:

  ```
  % ver sys5.3 man ls
  ```

This is equivalent to typing the following two consecutive command lines:

```
% SYSTYPE=sys5.3
% man ls
```

Remember that the value of SYSTYPE remains the same as it was before the **man** command was executed.

Password and User Identification

The process of login verification and home-directory setting are always handled by the Domain/OS login mechanism, but we provide a way to generate an **/etc/passwd** file so that UNIX programs that need to access it can do so.

Users cannot edit the **/etc/passwd** file directly, although they can read it. The registry server program **rgyd** generates the **/etc/passwd** file from the registry database, and updates **/etc/passwd** when the registry is updated. We provide the **edrgy** (edit registry) command so you can edit your registry information. Chapter 3 describes how to use **edrgy** to change your home directory.

All Domain/OS network registry information must be case correct. Otherwise, case sensitive programs will report that your home directory cannot be found.

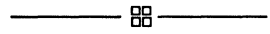
File Protection, Permissions, and Ownership

Domain/OS supports the standard UNIX protection mechanisms. We also provide an additional protection mechanism, the access control list (ACL). Every object (file, directory, etc.) has an ACL associated with it, although this is not noticeable if you only use standard UNIX permissions.

In addition to its own ACL, each directory contains two ACLs called **initial ACLs**. You can use the initial ACLs to control the way files and directories created in a directory inherit their protections. When you create a new file or directory, or copy one to a new location in the file hierarchy, the system assigns an ACL to it

by copying the appropriate initial ACL stored in the parent directory.

The **/usr/apollo/bin** commands **chacl** (change ACL), **cpacl** (copy ACL), **lsacl** (list ACL), and **dbacl** (Domain/Dialogue-based ACL editor) let view and control ACL values. For information about these commands, see the *BSD Command Reference*. A general description of ACLs is also located on the manual page for the **acl** (access control list) command.



Chapter 3

Understanding Startup and Login

Each time you start up a node and log in to it, the system executes various programs that set up the node's operating environment. You can tailor the operating environment on your node by modifying the scripts the system uses at startup and login. For example, you may want to start specific **daemons** (server processes) when you start up your node. Or, you may want your own specific key definitions, default window positions, and tabs defined each time you log in.

This chapter describes how the system functions at startup and login, and describes the steps you can take to tailor your operating environment. It also describes procedures for changing your password, log–in shell, user information, and home directory after you log in.

Understanding the System at Startup

The operating guide for your node describes the proper procedure for starting it up. When you initiate the node's startup by turning on the power, the node performs a series of operations to **boot** the operating system (load the operating system from disk into memory) and begin executing it. The operating system then executes a series of start-up files to set up the operating environment on your node.

This section explains the sequence of events occurring at startup for both disked and diskless nodes.

Disked Node Startup

If your node is a disked node, it reads the programs it needs for startup from its own disk. The flowchart in Figure 3-1 shows the start-up sequence on a disked node.

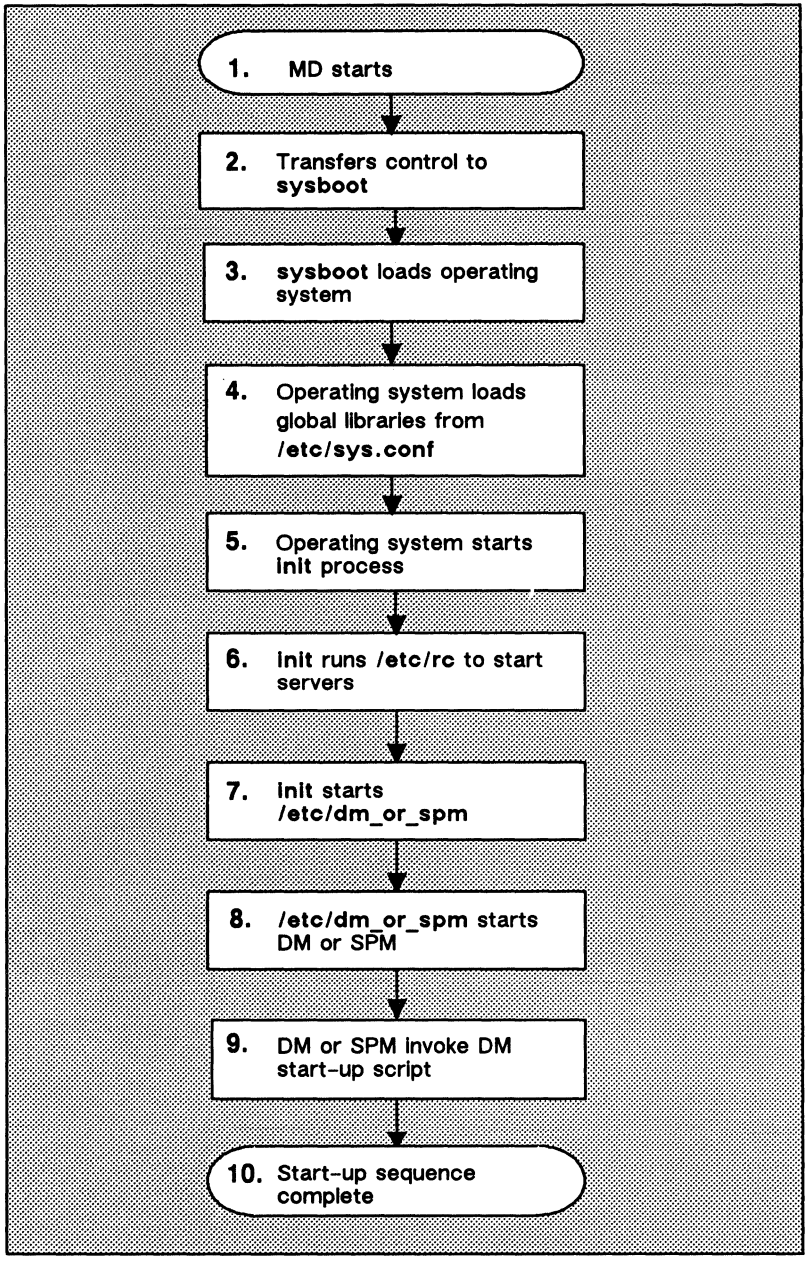


Figure 3–1. The Start–Up Sequence for Disked Nodes

The descriptions that follow explain each step in the start-up sequence shown in Figure 3-1.

1. When you power on your node in normal mode (follow the instructions in your operating guide), a program called the **Mnemonic Debugger (MD)** begins executing. The MD resides in the node's boot **PROM (Programmable Read-Only Memory)**.

2. The MD reads a program called **sysboot** from your node's disk and loads it into the CPU's memory. The MD then transfers control to **sysboot**. The **sysboot** program is responsible for booting the operating system.

3. The **sysboot** program loads the operating system into the CPU's memory. Once loaded, the operating system begins executing and takes control.

4. The operating system reads the file **/etc/sys.conf** to load global libraries.

5. The operating system starts the **init process** by running the program **/etc/init.**

   The **/etc/init** program reads the file **/etc/environ**. The **/etc/environ** file contains two lines, one for specifying the environment (BSD, SysV, Aegis™), and one for specifying the SYSTYPE variable (bsd4.3, sys5.3). If the environment is BSD or SysV, the default log-in shell for the node is **/bin/sh** (Bourne shell). If the environment is Aegis, the default log-in shell is **/com/sh** (Aegis shell).

6. The init process runs the **/etc/rc** script to start the necessary daemons. The **/etc/rc** file, which is normally a link to **'node_data/etc/rc**, is a file of commands to be executed at boot time. Many of these commands invoke daemons that must be invoked by the super-user ("root"). Any programs started by **/etc/rc** inherit the SYSTYPE value specified in the **/etc/environ** file.

   The **/etc/rc** program executes two additional **rc** scripts named **/etc/rc.user** (not run as "root", but as "user") and **/etc/rc.local**. The **rc** scripts contain commands that start various daemons. These server programs run regardless of log-in and log-out activity and provide various system services to the node.

For example, the **netman** program makes the node available as a host for diskless partners. For a description of these and all of the Domain server programs, see *Managing BSD System Software.*

If you want your node to automatically start any daemons, there are two methods you can use. The method you use depends on the types of servers you wish to run.

- To start servers such as **netman** or **mbx_helper**, that do not have to run (and will not be run) with a user ID of "root", edit the **/etc/rc.user** file and remove the pound sign (#) from the command line that invokes the server.

- To start up UNIX daemons such as **cron**, **inetd**, and **lpd**, or the Network Computing System (NCS) servers **llbd** and **glbd** (the location brokers), create a file in the directory **/etc/daemons** that has the same name as the server you wish to start. That is, if you wish to run the **llbd** server, create the file **/etc/daemons/llbd** (it doesn't matter what's in the file, **/etc/rc** only looks at the file name). See *Managing the NCS Location Broker* for more information about NCS servers.

Note, however, that the system will not start any of these servers until the next time the **rc** script is run. To do this, you should shut down and restart your node. (See your node's operating guide for node startup and shutdown procedures.)

7. The **/etc/init** program reads the file **/etc/ttys** (normally, this file is a link to the file **'node_data/etc/ttys**) and starts the **/etc/dm_or_spm** program associated with the display and listed in the file. Any programs started by **/etc/ttys** inherit the SYSTYPE value specified in the **/etc/environ** file. Other lines in the **etc/ttys** file contain directives that start **getty** on the tty lines for the node; see the **/etc/ttys** file for further information.

8. The **/etc/dm_or_spm** program starts either:

   - The **Display Manager (DM)** on nodes with displays.

   - The **Server Process Manager (SPM)** on Domain Server Processors (DSPs). The SPM allows you to create a process on a DSP from a remote node in the network. (For more information about the SPM, see *Managing BSD System Software.*)

9. The DM or the SPM executes a start-up file that sets up the initial operating environment on your node. Table 3-1 lists the different files used at startup. As shown in Table 3-1, the system chooses which file to execute according to the type of node.

   All of the DM start-up script files listed in Table 3-1 reside in the directory **'node_data**. The tick character (') that precedes the directory name is a special symbol that returns a value for **node_data**.

   NOTE: On Apollo nodes, the tick character is located on the same key as the tilde (~) character. It is not to be confused with the quote character ('), which is on the same key as the double quotes (").

   For example, on disked nodes, **'node_data** points to the **/sys/node_data** directory on the node's disk. On diskless nodes, the directory **'node_data** points to the directory **/sys/node_data.***node_id* on the partner node's disk. The *node_id* suffix refers to the diskless node's hexadecimal node ID. (Refer to the "Diskless Node Startup" section for more information on diskless node startup.)

Table 3-1. Node DM Start-Up Script Files

Node Type	Start-Up Scripts
1024x800 (Landscape) DN3xx, DN460, DN550, DN560, DN570, DN3000 (Color), DN3000 (15-inch Black & White) DN4000 (Color)	**startup.19l**
1280x1024 (Color Landscape) DN580	**startup.1280color**
1280x1024 (Black & White Landscape) DN3000 (19-inch Black & White), DN4000 (19-inch Black & White)	**startup.1280bw**
Displayless Domain Server Processors (DSPs)	**startup.spm**

Figure 3-2 shows a sample DM start-up script similar to the one we provide with DN3000 nodes. The DM start-up scripts for other nodes are similar.

```
# startup, /sys/dm, default system startup command file for 1280x1024

# Window positions for the DM's input and output windows.
# Do not comment these out.
(608,744)dr; (1023,799)cv /sys/dm/output
(556,744)dr; (608,799)cv /sys/dm/output;pb
(0,744)dr; (556,799)cv /sys/dm/input

# The default Apollo compose key is F5. It is normally NOT enabled.
# To enable it, uncomment the following line.
#
# cps /usr/apollo/bin/kbm -c f5
#
# To change it to a different key, edit the previous line as appropriate.
```

Figure 3-2. A Sample DM Start-Up Script

The DM start-up scripts that run on nodes that have displays contain a set of commands that instruct the Display Manager to draw the initial display windows on the screen. One of the windows contains the "login:" prompt.

These DM start-up scripts also let you enable a default Apollo compose key, or to change it to another key. For more information about this function, see Appendix F.

The **startup.spm** script used by DSPs is similar to the other start-up scripts. However, since DSPs don't have displays, **startup.spm** does not contain commands for creating windows.

10. Once the DM start-up script finishes executing, the node startup completes, and the system prompts you to log in.

Diskless Node Startup

The start-up sequence for diskless nodes is somewhat different than the start-up sequence for disked nodes. A diskless node does not have its own disk to store the operating system and other software files it needs to run. Therefore, each time it starts up, the diskless node must load parts of the operating system across the network from its partner node. The diskless node also relies on its partner for any utility programs and libraries it needs. Figure 3-3 presents a flowchart showing the start-up sequence for a diskless node.

From your perspective as a user, starting up a diskless node is the same as starting up a disked node; you turn the power on in normal mode and wait for the log-in prompt to appear. However, the start-up sequence that goes on internally is somewhat different. The descriptions that follow explain each step in the diskless node start-up sequence shown in Figure 3-3. Once you've read the descriptions, go back and compare each step with the disked node start-up sequence described in the "Disked Node Startup" section.

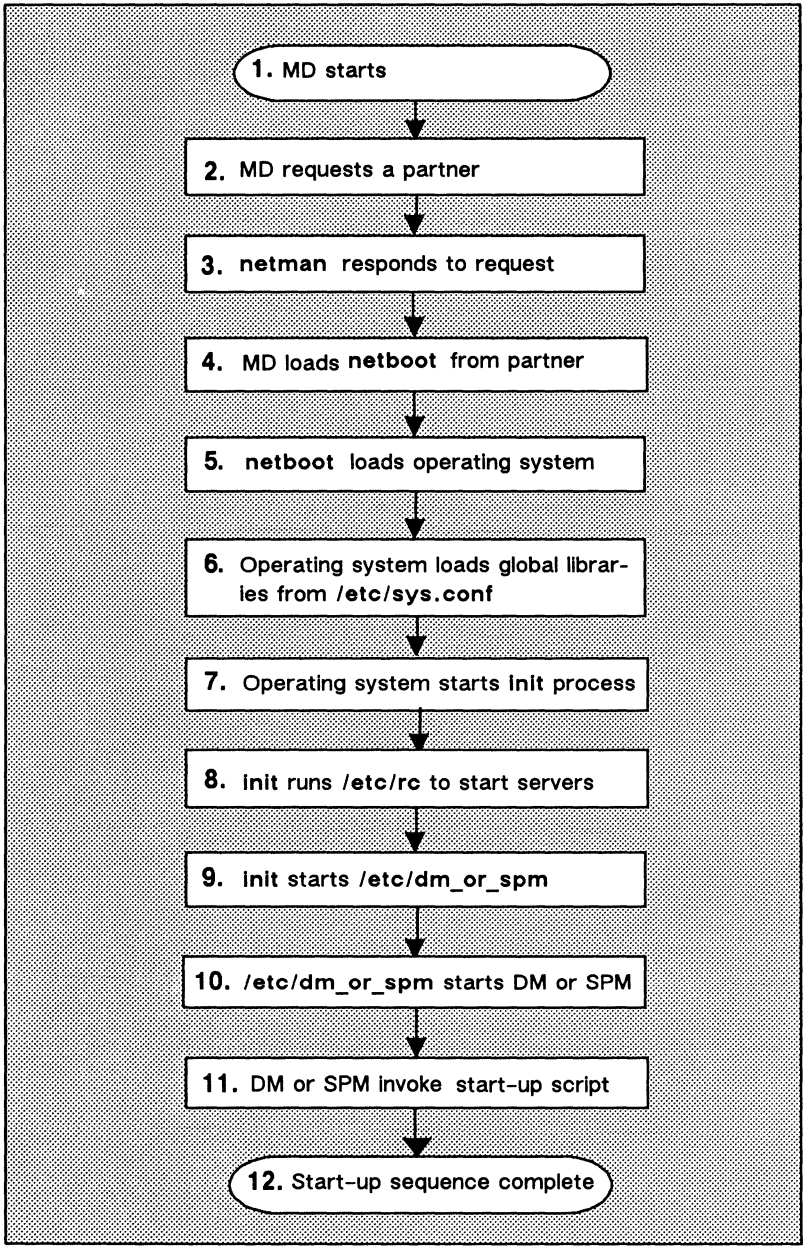


Figure 3–3. The Start–Up Sequence for a Diskless Node

1. When you power on your node in normal mode (by following the instructions in your node's operating guide), a program called the Mnemonic Debugger (MD) begins executing. The MD resides in the node's boot PROM (Programmable Read-Only Memory).

2. Because a diskless node does not have a disk, the MD cannot load **sysboot** and transfer control to it. Instead, the MD must boot the system from another disked node in the network. The MD then broadcasts a message across the network asking for a partner node to volunteer the use of its boot volume.

3. All nodes running the **netman** program receive these request messages (**netman**'s purpose is to respond to them). In response to the diskless node's request, **netman** on a disked node checks the file **/sys/net/diskless_list**. This file on the disked node contains a list of hexadecimal node IDs for all nodes the disked node may offer partnership.

   If the diskless list contains the ID of the diskless node requesting partnership, **netman** volunteers the node as a partner. The first disked node to volunteer becomes the partner of the diskless node. (It remains the diskless node's partner until the next time the diskless node boots.) At this point, the diskless node displays the partner node's node ID for your information.

   You can take a look at a sample diskless list by reading the file **/sys/net/sample_diskless_list**. For a complete description of how to create a diskless list and set up partners for diskless nodes, see *Managing BSD System Software*.

4. Once the diskless node finds a partner, the MD copies the **netboot** program from the file **/sys/net/netboot** on the partner node into the diskless node's memory. The **netboot** program is a special version of **sysboot** that diskless nodes use to boot the operating system across the network. The MD, when finished loading **netboot**, transfers control to it.

5. The **netboot** program, running on the diskless node, loads the operating system from the partner node's boot volume into memory.

6. The operating system reads the file **/etc/sys.conf** to load global libraries.

7. The operating system runs **/etc/init** to start the init process; **/etc/init** reads the file **/etc/environ**. The **/etc/environ** file establishes the default log-in shell and default SYSTYPE for the node.

   The **/etc/environ** file contains two lines, one for specifying the environment (BSD, SysV, Aegis), and one for specifying the SYSTYPE variable (bsd4.3, sys5.3). If the environment is BSD or SysV, the default log-in shell for the node is **/bin/sh** (Bourne shell). If the environment is Aegis, the default log-in shell is **/com/sh** (Aegis shell).

8. The init process runs the **/etc/rc** script to start the necessary daemons. The **/etc/rc** file, which is normally a link to **'node_data/etc/rc**, is a file of commands to be executed at boot time. Many of these commands invoke daemons that must be invoked by the super-user ("root"). Any programs started by **/etc/rc** inherit the SYSTYPE value specified in the **/etc/environ** file.

   The **/etc/rc** program executes two additional **rc** scripts named **/etc/rc.user** (not run as "root", but as "user") and **/etc/rc.local**. The **rc** scripts contain commands that start various daemons. These server programs run regardless of log-in and log-out activity and provide various system services to the node. For example, the **netman** program makes the node available as a host for diskless partners. For a description of these and all of the Domain server programs, see *Managing BSD System Software*.

   If you want your node to automatically start any daemons, there are two methods you can use. The method you use depends on the types of servers you wish to run.

   - To start servers such as **netman** or **mbx_helper**, that do not have to run (and will not be run) with a user ID of "root", edit the **/etc/rc.user** file and remove the pound sign (#) from the command line that invokes the server.

- To start up UNIX daemons such as **cron, inetd,** and **lpd,** or the Network Computing System (NCS) servers **llbd** and **glbd** (the location brokers), create a file in the directory **/etc/daemons** that has the same name as the server you wish to start. That is, if you wish to run the **llbd** server, create the file **/etc/daemons/llbd** (it doesn't matter what's in the file, **/etc/rc** only looks at the file name). See *Managing the NCS Location Broker* for more information about NCS servers.

Note, however, that the system will not start any of these servers until the next time the **rc** script is run. To do this, you should shut down and restart your node. (See your node's operating guide for node startup and shutdown procedures.)

9. The **/etc/init** program reads the file **/etc/ttys** (this is normally a link to the file **'node_data/etc/ttys**) and starts the **/etc/dm_or_spm** program associated with the display and listed in the file. Any programs started by **/etc/ttys** inherit the SYSTYPE value specified in the **/etc/environ** file. Other lines in the **etc/ttys** file contain directives that start **getty** on the tty lines for the node; see the **/etc/ttys** file for further information.

10. The **/etc/dm_or_spm** program starts either:

    - The **Display Manager (DM)** on nodes with displays.

    - The **Server Process Manager (SPM)** on Domain Server Processors (DSPs). The SPM allows you to create a process on a DSP from a remote node in the network. (For more information about the SPM, see *Managing BSD System Software.*)

11. The DM or the SPM executes a start-up file that sets up the initial operating environment on your node. Table 3-1 lists the different files used at startup. As shown in Table 3-1, the system chooses which file to execute according to the type of node.

    Since diskless nodes don't have files of their own, the DM or SPM must look to the partner node to find its start-up script file. Just as on a disked node, the DM or SPM on a

diskless node searches for the script file in the directory **'node_data**. Unlike a disked node, however, **'node_data** for a diskless node points to the **/sys/node_data.***node_id* directory on the partner's disk. (The *node_id* suffix is the hexadecimal node ID of your diskless node.)

NOTE: The tick character (') that precedes the directory name is a special symbol that returns a value for **node_data**. On Apollo nodes, the tick character is located on the same key as the tilde (~) character. It is not to be confused with the quote character ('), which is on the same key as the double quotes (").

12. Once the DM or SPM finds the diskless node's DM start-up script, the script executes, the node startup completes, and the system prompts you to log in.

Figure 3-2 shows a sample DM start-up script similar to the one we provide with DN3000 nodes. For information about this script refer to the "Understanding the System at Login" section.

A single disked node can serve as the partner for several diskless nodes. Each diskless node may need to use a "node-specific" boot script to set up its own unique operating environment. Therefore, the system uses the *node_id* suffix to denote a unique DM start-up script location for each diskless node assigned to the partner.

At startup, if the partner does not have a **'node_data** directory set up for the diskless node, **netman** creates one, copying it from a template stored in the partner's **'node_data** directory. The **netman** program then copies the partner node's DM start-up script file into the diskless node's **'node_data** directory. If you want the newly created script to perform different operations at startup than its partner, edit the script.

A major difference between the disked node and diskless node start-up sequence is the step where the DM or SPM searches for the node's DM start-up script. Figure 3-4 summarizes this search.

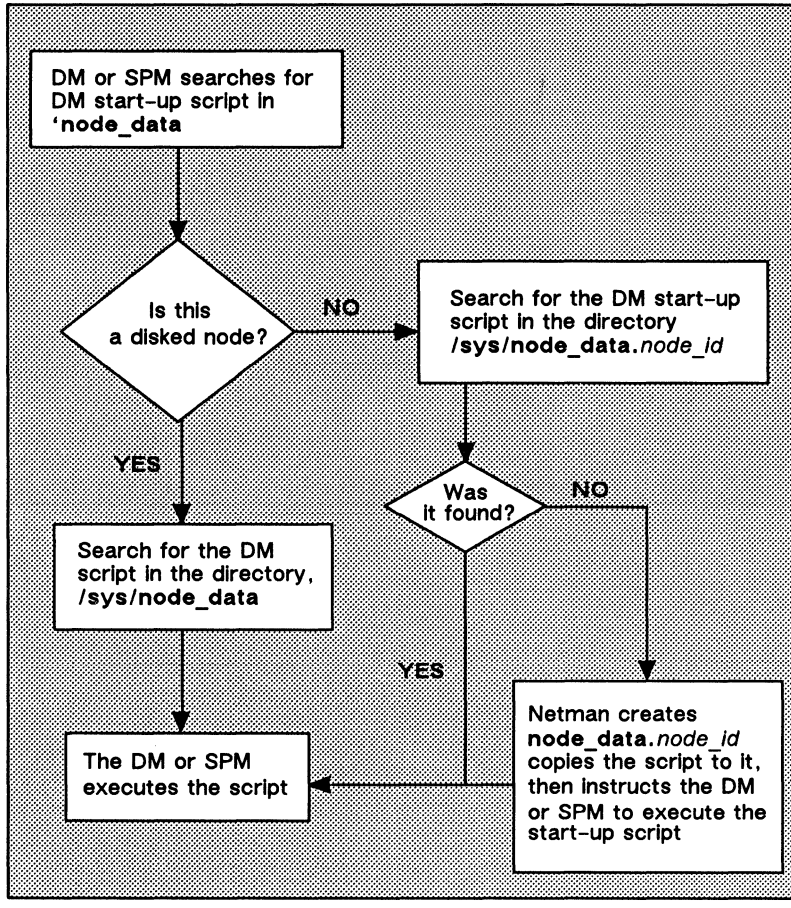


Figure 3–4. The Start–up Script Search Sequence

Understanding the System at Login

Once a node is up and running, you are ready to log in. At login, the system executes a series of scripts that set up the working environment for your log–in session. This section describes the sequence of steps the system performs at login. This section also explains how to create and modify scripts to tailor your log–in environment. Figure 3–5 shows the log–in sequence for a node.

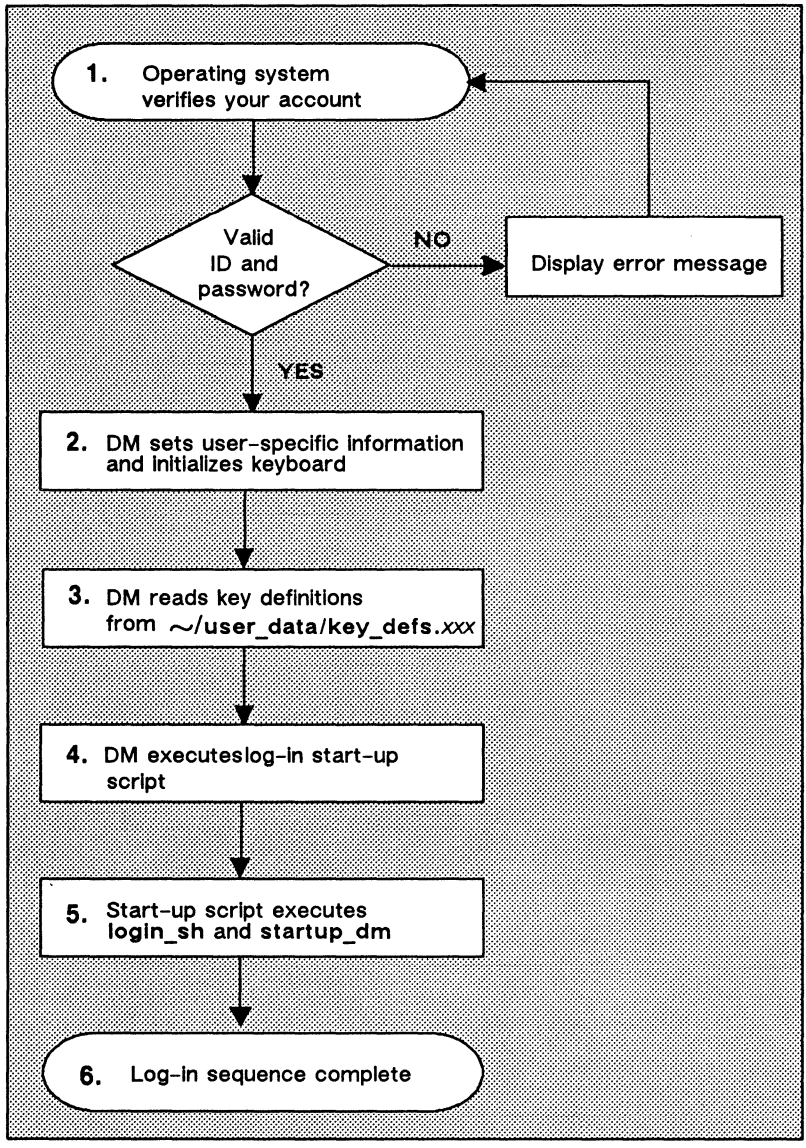


Figure 3–5. The Log–In Sequence

The descriptions that follow explain each step in the log–in sequence shown in Figure 3–5.

1. After you enter your username and password, the operating system verifies your account.

   The system verifies your account by checking the site registry. If the username and password match a valid account in the registry, the system executes the next step. If the system cannot verify the account, the log-in attempt fails, and the system displays a log-in error message in the DM output window. For more information about user accounts and registries, see *Managing BSD System Software*.

2. The DM sets your home directory from your account entry in the registry and looks there for a **.environ** file. If found, the DM sets the environment and then the SYSTYPE variable; otherwise, the node defaults are used. The DM then sets the variables SHELL, HOME, USER, LOGNAME, PROJECT, ORGANIZATION, and TERM. If no SHELL variable is specified in the registry entry, the node default is used. Based on the environment, the DM loads base key definitions, both **std_keys.basic** and either **std_keys** or **std_keys.unix**.

3. The DM reads the file **key_defs_8bit3** (for nodes with Low-Profile Model II keyboards), and **key_defs_8bit2** (for Low-Profile Model I keyboards). These files, located in the **user_data** directory of your log-in home directory, contain a record of any key definitions that you made the last time you were logged in. By reading these files, the DM carries over key definitions to the new log-in session. These files are non-ASCII files; therefore, you cannot edit them. The "Defining Keys" section in Chapter 4 describes the key definition files in more detail.

4. The DM (on nodes with displays) executes the node's **log-in start-up script**, which resides in one of the files listed in Table 3-2. As shown in Table 3-2, the system chooses which log-in start-up file to execute according to the type of node you are using. Note that on DSPs, the SPM does not execute a log-in start-up script.

   The DM looks for log-in start-up scripts in two different locations. First, it looks in **'node_data**, which refers to the node's specific **/sys/node_data** directory. (By default, no log-in start-up script exists in **'node_data**; you must put one there.) If the DM doesn't find the log-in start-up

script in **'node_data**, it executes one of the default log-in start-up scripts that we supply in the directory **/sys/dm**.

Table 3-2. Node Log-In Start-Up Script Files

Node Type	Log-In Start-Up Scripts
1024x800 (Landscape) D3xx, DN460, DN550, DN560, DN570, DN3000 (Color), DN3000 (15-inch Black & White), DN4000 (Color)	**startup_login.19l**
1280x1024 (Color Landscape) DN580	**startup_login.1280color**
1280x1024 (B & W Landscape) DN3000 (19-inch Black & White), DN4000 (19-inch Black & White)	**startup_login.1280bw**

5. As shown in Figure 3-6, the command that creates the log-in shell process is not commented out in the script. You may leave it in, comment it out by preceding it with a pound sign (#), or change it to draw the process's windows in a different location. The DM executes the **login_sh** command. The **login_sh** command executes your log-in shell based on the current value of SHELL, as set by the DM.

```
# startup_login (the per_login startup file in 'node_data or /sys/dm
# main shell whose shape is generally agreeable to users of this node
(0,300)dr: (700,700)cp /sys/dm/login_sh
# and the user's private dm command file from his home
# directory's user_data sub-directory. Personal key_defs file is also
# kept in user_data by DM.
cmdf user_data/startup_dm.1280bw
```

Figure 3-6. A Sample DM Log-In Start-Up Script

This log-in shell looks for a **shell log-in script** in your home directory. If this script exists, the shell executes it to set up your initial shell environment. The C shell looks for a script named **~/.login** (the C shell also executes a script named **~/.cshrc**; see Chapter 8 for more information about this script). The Bourne shell and the Korn shell look for a script named **~/.profile** (see Chapters 9 and 10).

6. At this point, the log-in sequence is complete.

You may want to create a DM log-in start-up script in **'node_data** in cases where you don't want the DM to execute the default version. For example, a diskless node, by default, uses one of the log-in start-up scripts located in its partner's **/sys/dm** directory. If you want the diskless node to execute its own unique DM log-in start-up script, you can create a copy in the diskless node's **'node_data** directory. For more information about **'node_data** for diskless nodes, refer back to the "Diskless Node Startup" section.

The system uses log-in start-up scripts to start processes that you'll need while you are logged in to your node. The log-in start-up scripts contain commands to execute a **log-in shell**, and to run your personal DM start-up script. For example, the log-in start-up scripts that we supply for nodes with displays create a process running the shell program. When you log out, the DM stops the shell process and deletes its pads and windows from the display.

If you wish to execute certain commands or processes once, when you log in, then you should create a **~/.profile** or **~/.login** file containing the commands. This file is only executed by a log-in shell. If you have commands that you wish to execute every time you start a new shell, you should create an additional shell start-up file. For more information about shell start-up files, see Chapters 7, 8, 9, and 10.

The last line in the sample script shown in Figure 3-6 contains the DM command **cmdf** (command file). This command invokes another script, **startup_dm.1280bw**. The DM attempts to execute this additional script as part of the log-in sequence.

If no pound sign precedes the **cmdf** command line, the DM looks in the **user_data** subdirectory of your log-in home directory for the specified file. If the DM finds the file, it executes the script; otherwise, it displays an error message in the DM output window when the log-in sequence completes.

This script, called the **DM start-up script**, is an optional script that you create to execute additional DM commands during login. For example, you may want to include commands that make specific key definitions or run specific programs. Figure 3-7 shows a sample DM start-up script.

```
# user_data/startup_dm (in login home directory)
# Some personal preference keys:
#
# Define < F4 > and < F5 > for easy Pascal indenting and undenting:
#
kd F4 t1;s/%        // ke
kd F5 t1;s/%/        / ke

# Set tab every 8 spaces:
#
ts 8 -r

# Set window default location
(0,770)dr;(600,110)  wdf1

# Build a Korn shell window
#
(0,500)dr;(799,955) cp /bin/ksh -DENV=~/.kshrc
```

Figure 3-7. A Sample DM Start-Up Script

Remember, we don't supply a DM start-up script or a shell log-in script as part of the system; if you want to use a DM start-up script or a shell log-in script, you must create one. If you do create a DM start-up script, remember to create a file that has the same filename as the file specified with the **cmdf** command. For example, the **cmdf** command in Figure 3-6, specifies the filename **startup_dm.1280bw**. The suffix **1280bw** is the suffix for files used by nodes with 19-inch monochromatic landscape displays, like the DN3000.

Logging In

This section describes the various log-in procedures you can use to log in as user, change your password, home directory, default shell, and user information. It also explains how to log in to a **Domain Server Processor (DSP)** and how to log in over a dialup line.

Logging In to a Default Account

The registry file **account**, described earlier in the "Understanding the System at Login" section, contains a default account named **user.none.none**, or simply **user**. This default account allows any user anywhere in the network to log in to an Apollo node.

To use the default account, log in with the username **user**:

```
login: user
```

NOTE: Your system administrator may have added a password to this account. In this case, ask him or her about it.

Changing Your Password

You can change your password anytime after you log in by using the **passwd** (change log-in password) command as follows:

```
% passwd username
```

The **passwd** command prompts for the old password and then for the new one. For verification purposes, you are asked to type the new password twice.

Use the new password the next time you log in. If you want to maintain a secure account, avoid using obvious passwords such as anyone's name or your initials. In addition, it is best to select a password that is at least six characters long. If security is not a high priority, you can use a blank password. (Note, however, that blank passwords destroy system security.)

The **passwd** command first writes to the registry; then, the system builds the **/etc/passwd** file from the information supplied to the registry. Only the owner of the account or the super-user may change a password. The super-user does not need to know the password in order to change it; anyone else does.

Changing Your Home Directory

Each system account has a directory associated with it, called the home directory. Anytime you log in, the system sets your initial working directory to your home directory. You can change your home directory by using **edrgy** (edit registry) command as follows:

```
% /etc/edrgy
edrgy => change username -h new_pathname
edrgy => quit
```

When you enter the pathname of your new home directory, the system attempts to update the account in your site registry directory. The registry database contains information about your account, such as your username, password, and home directory. By updating the registry, the system stores your new home directory for logging in later. See *Managing BSD System Software* for more information about system registries; see *Managing BSD System Software* for more information about the **edrgy** command.

Changing Your Default Log-In Shell

You can change your default shell after logging in by specifying the **chsh** (change shell) command using the following format:

```
% chsh username
```

Unless you are the super-user, the new log-in shell must be one of the approved shells listed in the **/etc/shells** file. If **/etc/shells** doesn't exist on your node, the only shells that may be specified are **/bin/sh**, **/bin/csh**, **/bin/ksh**, and **/com/sh** (assuming that you have a **/com** directory on your node). The super-user may change anyone's log-in shell; other users may only change their own log-in shell.

Changing Your User Information

Information concerning your username is kept in the **/etc/passwd** file. On Domain/OS systems, the **/etc/passwd** file is a typed file that is generated by the registry daemon automatically. If user information in your registry has not been made read-only, it is possible to change your full name. To do this, you must execute the **chfn** (change password file information) command as follows:

% **chfn** *username*

Unless you are the super-user, you may only change your own username.

Logging In to a Domain Server Processor (DSP)

Unlike user nodes, a Domain Server Processor (DSP) doesn't have a keyboard or display. Therefore, you must log in to it from a user node in the network.

As described earlier in the "Disked Node Startup" section, when you start up a DSP, the system starts a program called the Server Process Manager (SPM). The SPM makes it possible for you to create a process on the DSP, log into the process, and execute programs and commands, all while you sit at a user node in the network. For a complete description of the procedure for logging into a DSP, see the *Owner's Guide* for your particular processor.

Logging In Over a Dialup Line

When logging in over a dialup line, the process is somewhat different from that mentioned earlier in this chapter. Figure 3-8 illustrates this process.

> **NOTE:** If, after you get carrier, the login prompt appears scrambled, it is likely that the baud rate on your terminal is incorrect. To correct this, try pressing <RETURN> once or twice. If this fails, send a "break" from your terminal; this will force a baud rate change.

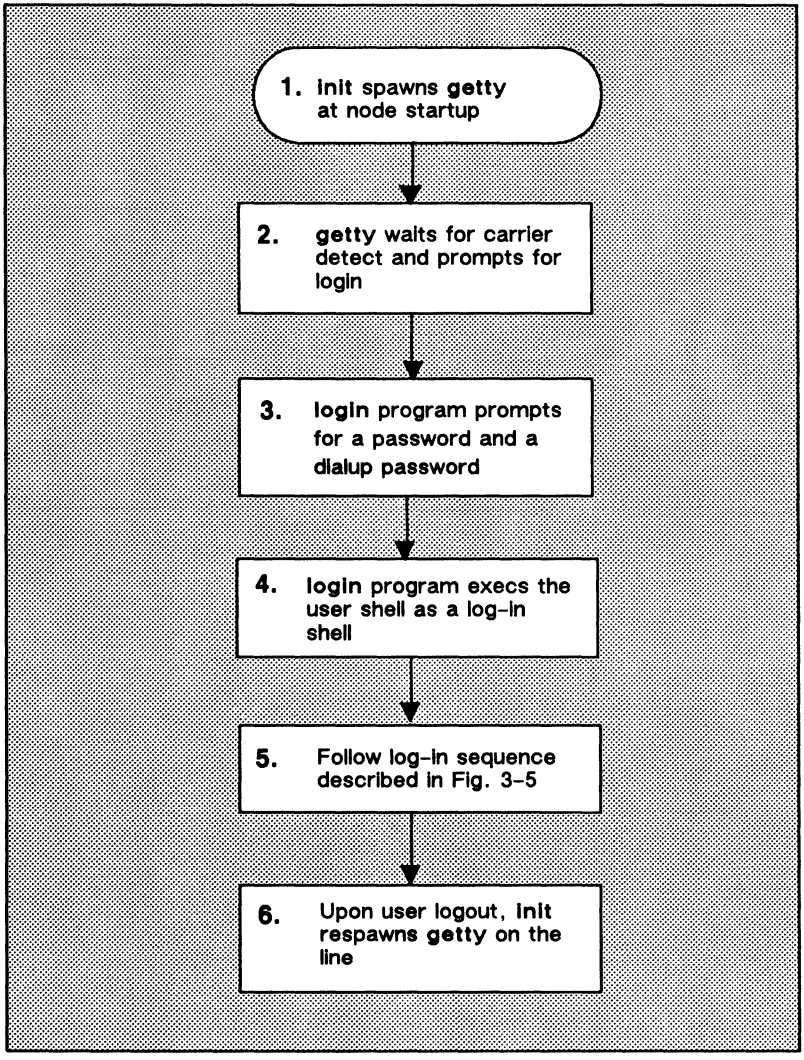


Figure 3-8. Login Over a Dialup Line

Chapter 4

Using the Display Manager

By default, the Display Manager (DM) is the window manager program that controls your node's display. Using DM commands, you can instruct the DM to perform specific display management operations, such as: moving the cursor around the display, creating and controlling processes, creating and manipulating pads and windows, and modifying display characteristics.

This chapter explains the functions of the DM and describes how to specify DM commands. It also describes how to define keys to perform DM operations. Chapter 5 describes how to use the DM to perform specific display-management tasks.

Using DM Commands

DM commands enable you to control your node's display by instructing the DM to perform specific display management operations. To use a DM command, you normally perform two basic steps:

1. Move the cursor to the spot on the display where you want the DM operation performed.

2. Specify a DM command to execute the operation.

You indicate a spot on the display either by moving the cursor to the desired spot, or by explicitly defining a point on the screen as a command argument. If you don't perform a pointing operation using either method, the DM executes the command at the current cursor position.

Some DM commands require you to define an area, or **region**, on the screen instead of a single point. You define the size of a region by defining two points on the screen; one point specifies the upper left corner, and the other specifies the lower right corner. The region is simply the area between the two points. The "Defining Points and Regions" section describes how to define points and regions.

To specify a DM command interactively:

1. Press <CMD> to move the cursor next to the "Command:" prompt in the DM input pad. (The DM remembers where the cursor came from so it can apply the next command to that point.)

2. Type the command along with any arguments or options.

3. Press <RETURN> to invoke the command.

Use this procedure to specify commands interactively from your keyboard. You can also specify commands in special DM programs, called scripts. When you invoke a DM script, the DM reads and executes DM commands in the order you specify them. The "Using DM Command Scripts" section describes how to use DM scripts.

The method you use to define a point depends on the DM command you use, and how you use it. When you specify a command interactively, you usually move the cursor to the desired point; in scripts, you specify a point explicitly as a command argument. Figure 4–1 illustrates the interactive procedure for invoking the **wc** (window close) command to delete a window.

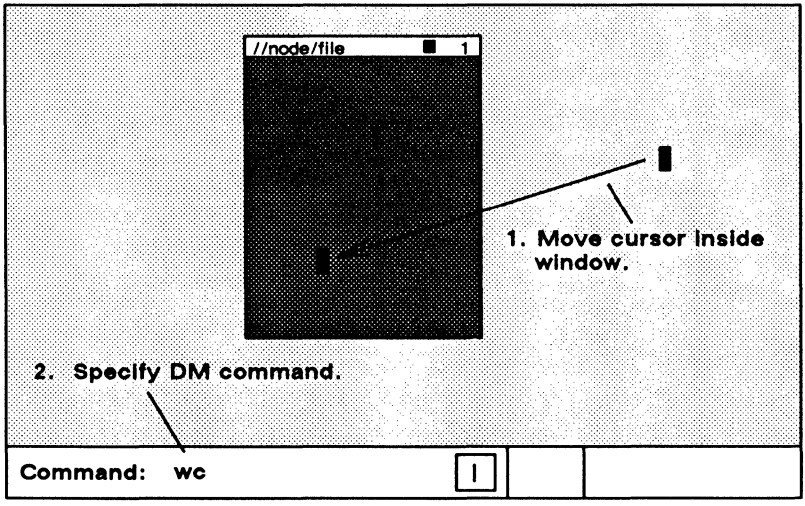


Figure 4–1. Invoking a DM Command Interactively

You can also invoke DM commands interactively using DM function keys and control key sequences. The "Using Keys to Perform DM Functions" section describes how to use these keys to perform DM functions.

DM Command Conventions

DM commands have the following general format:

[*region*] *command* [*arguments ...*] [*options ...*]

Separate the components of a command with the proper command line delimiters, as follows:

- Separate an argument from a command and any additional arguments or options with at least one blank space.
- Precede each option with a hyphen (–). Separate each option from commands, arguments, or any additional options with at least one blank space.

- If you precede the command with a region, make sure you use the correct syntax to define each point (see the section "Specifying Points on the Display"). You can place multiple blanks before and after the region, although they are not required.

- You can string multiple commands together on the same line by separating each command with a semicolon (;) as shown below:

  pt;tt;tl

  This command sequence executes three separate commands to move the cursor to the first character in a pad.

Using DM Special Characters

When you use commands in scripts and key definitions, you can use several special characters that control how the DM interprets commands. The following describes the rules for using these special characters:

@ The escape character (@) always nullifies the meaning of any special character (e.g., the input request character) it precedes. When the DM reads a command line containing the escape character, it strips off the @ character, and any special meaning of the character following it.

If you can't remember whether a character has some special meaning, it is safe to escape the character. If the character is not special, the DM still removes the @, so the character appears as it should. Character escaping is generally confined to search and substitute operations (see Chapter 5), commands requiring quoted strings, and key definitions.

\# When the DM reads the pound sign (#) in a DM script, it ignores the information on the remainder of the line. Use this character to add comments to your DM script or to prevent the execution of a line in the script.

; Use the semicolon (;) to separate commands that you specify on the same line.

& The input request character (&) enables you to supply keyboard input from the DM input pad to a command in a key definition or script. When the DM reads the &, it stops reading commands and moves the cursor to the DM input pad. When you enter input (usually a command argument), the DM replaces the & character with the specified input and continues reading commands. You can also specify a prompt in the form

& *'prompt'*

to display a prompt in the DM input pad that requests the proper input.

Like the & character, the **kd**, **es**, **cp**, **cpo**, and **cps** commands accept strings surrounded by single quotes. When you use single quotes, the only characters in the quoted string that retain their special meaning are @ and &; all other characters revert to their literal values. Note, however, that the **kd** (key definition) command does not recognize single quotes within the definition string.

Defining Points and Regions

As noted earlier, you may specify the location for a DM operation by using either the cursor or an explicit coordinate list.

If you use the cursor, remember that it actually occupies many individual screen points. When you use the cursor to point to a spot on the screen, the lower left-hand corner of the block cursor designates the exact point. (When you point to the upper edge or right edge of a window, the DM adjusts the point position to account for the size of the cursor. See the "Creating Pads and Windows" section in Chapter 5 for more information on how the DM defines window boundaries.)

Specifying Points on the Display

If you choose not to point with the cursor, you can explicitly define a point or pair of points (a region) using any of the point formats described below. Note that some formats define points in *pads*, and others define points on the *display* as a whole. You normally define points in pads when performing the pad editing operations described in Chapter 6.

line–number

Specifies a line location in a pad. Line numbers begin at 1 and increase moving toward the last line in the pad. To refer to the last line in a pad, you may specify a dollar sign ($). The edit pad window legend displays the line number of the top line in a window. You can also display the line number (plus the column number, and x– and y–coordinates) of the current cursor position by using the DM command =.

+/– *n*

Specifies a line location in a pad that is *n* lines before (–) or after (+) the current cursor position.

[[*line–number*][*,column–number*]]

Specifies a point in a pad by line and column number. The DM assumes the current line if you omit *line–number*; it assumes column 1 if you omit *column–number*. Line numbers range from 1 to the last line in the pad. Column numbers range from 1 to 256. Some examples are:

[127,14]	Line 127, column 14.
[53]	Line 53, column 1.
[,12]	Column 12 of the current line.

Note that you must use the outer set of square brackets; however, when you specify *line–number* only, the brackets are optional. When using this format, you cannot use the dollar sign ($) to specify the last line in a pad; you must specify the number of the last line.

/regular–expression/ or *\regular–expression*

Specifies a string in a pad that begins or ends a specific region. Chapter 6 describes regular expressions.

([*x-coordinate*] [,*y-coordinate*])

Specifies a point on the display by screen coordinates. Screen coordinates indicate bit positions on the display. The origin 0,0 is at the extreme upper-left corner of the screen. Table 4-1 shows the ranges for coordinate values.

Table 4-1. Ranges for Coordinate Values

Display Type	x-coordinate	y-coordinate
1024x800	0 to 1023	0 to 799
1280x1024 (landscape)	0 to 1279	0 to 1023
800x1024 (portrait)	0 to 799	0 to 1023
1024x1024 (square)	0 to 1023	0 to 1023

If you omit either coordinate from the specification, the DM uses the coordinates of the cursor. You *must* enclose the coordinates in parentheses. Some examples are:

(200,450) Bit position with an x-coordinate of 200 and a y-coordinate of 450.

(135) Bit position with an x-coordinate of 135 and the same y-coordinate as the current cursor position.

(,730) Bit position with the same x-coordinate as the current cursor position, and a y-coordinate of 730.

When you specify any of the formats described above in the DM input pad, the DM moves the cursor to the specified position. For example, to move the cursor to line 75, column 5 in an edit pad, specify the following in the DM input pad:

```
Command: [ 75,5 ]
```

You can also use any of the formats for defining points to define a region on the display. To define a region, you must define two points as follows:

[*point*] **dr**; [*point*]

The first point defines the beginning of the region and the **dr** command marks it. The second command defines the end of the region. When defining a two-dimensional region, the first point defines one corner, and the second point defines the opposite corner as shown in Figure 4-2.

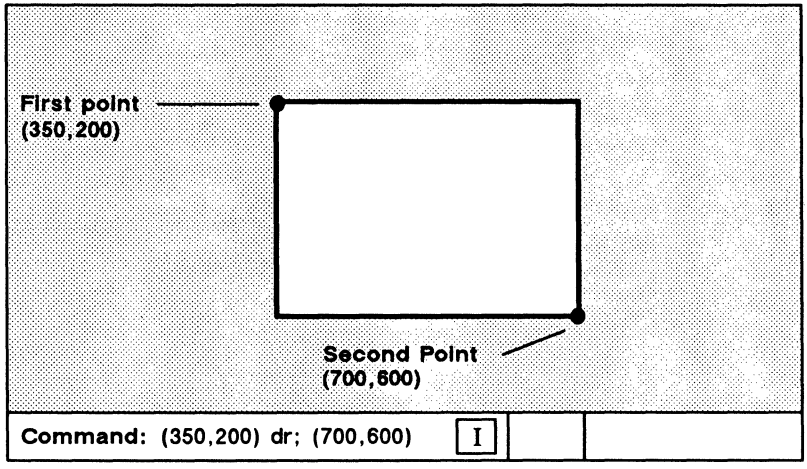


Figure 4-2. Defining a Display Region

When you define a region, if you don't specify a second position, the DM uses the current cursor position.

Like defining a single point, an easy way to define a region is to point with the cursor. Press <MARK> to invoke the **dr** command, which marks the first point. To define a region using the cursor:

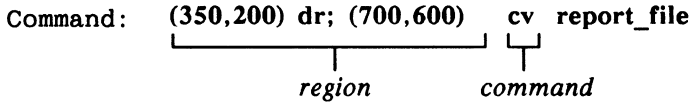
1. Move the cursor to the first point.
2. Press <MARK>.

3. Move the cursor to the second point.

4. Specify the DM command.

For a complete description of the DM commands used to control marks, see the *Domain Display Manager Command Reference*.

For commands that require a region in which to operate, you have the option of specifying the region as part of the command. The **cv** (create view) command, shown below, creates a read-only pad and window. It uses a region to define the size and location of the window it creates.



Using Keys to Perform DM Functions

You can also perform display management operations using keys, called function keys, that we've defined as specific DM commands. When you press a function key, it invokes its assigned DM command or command sequence.

By default, many keys perform DM operations when pressed simultaneously with <CTRL>. Like function keys, these key combinations, called **control key sequences**, provide you with a "shorthand" method of specifying commands.

Domain/OS predefined function keys and control key sequences enable you to execute commonly performed operations. For example, the directional keys described in Chapter 5 are predefined keys that you'll use routinely to move the cursor.

We've also defined the mouse's function keys to perform three useful DM operations. Table 4-2 describes the default mouse key functions.

Table 4–2. Default Mouse Key Functions

Mouse Key	Function
Left Key (M1)	Performs a GROW/MARK operation to change the size of windows. See Chapter 5 for details on using the left mouse key to change the size of a window.
Center Key (M2)	Works just like <POP>. To use it, move the cursor inside the window you want to pop, then press the key. See Chapter 5 for more information.
Right Key (M3)	Lets you read files whose names appear in the pad (any full or relative pathname also works). This key executes the **cv** command with the name of the file you indicate with the cursor. To use this key, position the cursor over any part of the name of the file you want to read, and then press the key.

Keyboard Types and Key Definitions

Domain/OS supports two basic types of keyboards:

- Low–Profile keyboards
- Multinational keyboard

Low–Profile keyboards (shown in Figure 4–3) include the Low–Profile Model I keyboard and the Low–Profile Model II keyboard. Notice that the key layout for both of these keyboards is the same except that the Model II keyboard has a numeric keypad and two additional function keys, F0 and F9.

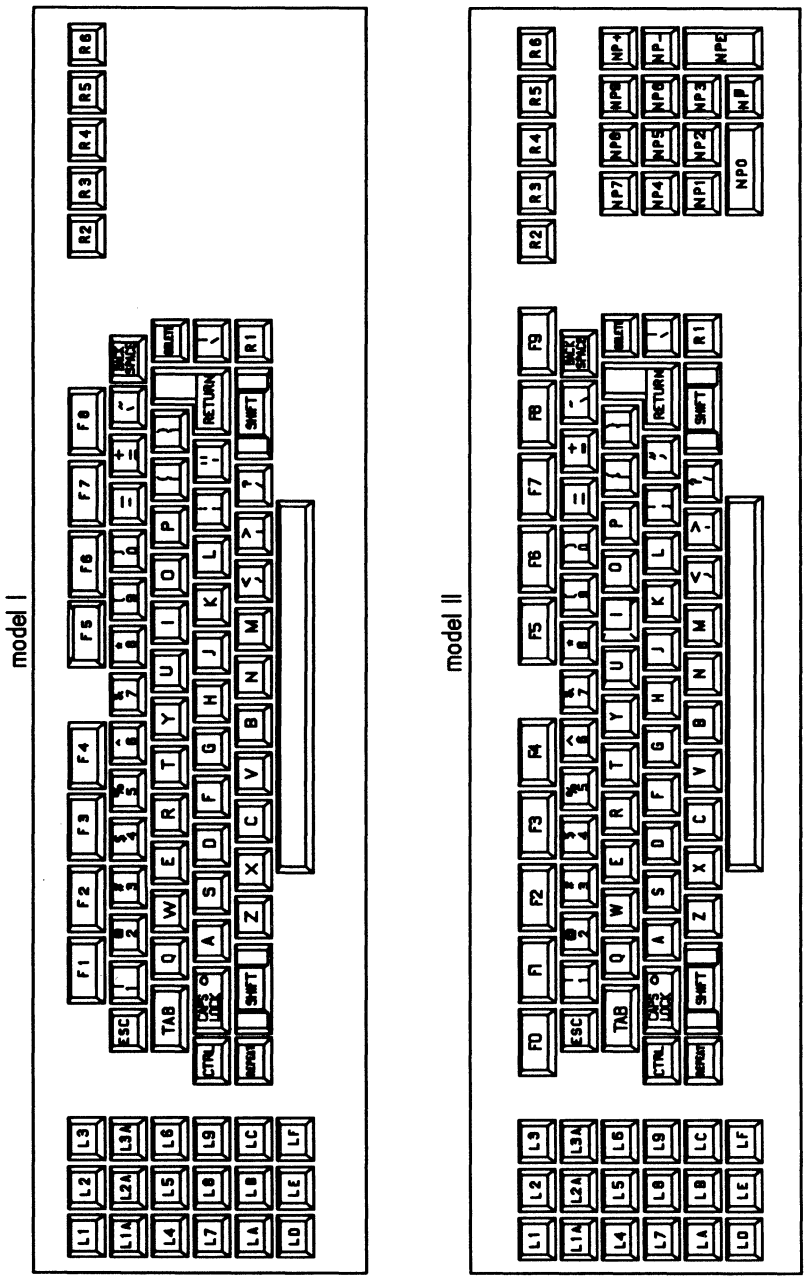


Figure 4–3. Key Names for the Low–Profile Keyboards

The Multinational keyboard is a Low-Profile Model II keyboard adapted to international standards. The Multinational keyboard has seven additional keys that impose a slightly different overall arrangement, as well as some different key labels. Each national version of the Multinational keyboard has the same physical layout. See Appendix B for information on the predefined keys of the Multinational keyboard.

European characters do not appear on the standard North American keyboards, and only a subset appears on the various models of the Multinational keyboards. You can create and display European characters in the Latin-1 character set that do not appear on your keyboard, by using the Domain/OS compose function. See Appendix F for information about the compose function.

The system stores the definitions for its predefined keys in a keyboard-specific definition file. Table 4-3 lists the names of the definition file for each keyboard.

Table 4-3. Key Definition Filenames

Keyboard	Key Definition File
Low-Profile Model I	**/sys/dm/std_keys2**
Low-Profile Model II	**/sys/dm/std_keys3**
Multinational Keyboard	**/sys/dm/std_keys3***x* (*x* is a letter from a-g)

The assigned key definitions for the Multinational keyboard are stored in a keyboard-specific definition file, **/sys/dm/std_keys3***x*, where *x* is a letter from **a** to **g** representing the following:

a	Germany
b	France
c	Norway/Denmark
d	Sweden/Finland
e	United Kingdom
f	(Reserved for future use)
g	Switzerland

All command files listed in Table 4–3 contain a line invoking the standard Domain/OS key definition file, **/sys/dm/std_keys.basic**. (In addition, the Multinational keyboard key definitions invoke the file **/sys/dm/std_keys.mn**). If your environment is set to either BSD or SysV, the **/sys/dm/std_keys.unix** file is automatically invoked when you log in. This file overwrites some of the standard key definitions to provide necessary UNIX functions.

The UNIX key definitions file includes commands that bind various keys to certain version–specific (or shell–specific) features. Chapter 2 describes which keys are redefined when the keyboard is remapped to **std_keys.unix**. You can also define your own function keys and control key sequences by assigning commands to specific key names. But, before you can define keys, you must understand how they are named. The following sections describe key naming conventions and describe how to define keys.

Key Naming Conventions

The DM identifies each key on your keyboard (and mouse) by a unique name. The names of the ordinary character keys (letters and numbers) have the same name as the characters they represent. For example, the A key has the name "A". Other keys, like the DM function keys, have special names that are different than the names written on them. <READ>, for example, has the name "R2". Figure 4–3 shows the names and locations of the keys on both the low–profile type keyboards.

For instance, the <CUT>/<COPY> function key (L1A) performs a different function when you use it with <SHIFT>. The name L1A identifies the key's normal function (when you press the key down). The name L1AS, the key's shifted name, identifies the key's function when pressed along with <SHIFT>. The key's up–transition name, L1AU, identifies the function that the key performs when it goes up. The name L1AC, the key's control key sequence name, identifies the function when pressed along with <CTRL>.

NOTE: The names of the Multinational numeric keypad keys differ from those found on the standard Low–Profile Model II keypad keys. Appendix B provides further detail on the Multinational keyboard.

When defining a key as a command or sequence of commands, you use the same name that the DM uses to identify the key. Some keys, like the DM and program function keys, function differently depending on how you use them. Therefore, each of these keys has a set of additional names that identify the manner in which the key is used.

Table 4–4 describes the key naming conventions you should use when defining keys.

Table 4–4. Key Naming Conventions

Key Type	Description
Ordinary Characters	Have the same name as the numbers and letters they represent. You can assign functions to lowercase letters and numbers, capital letters, and special characters. When specifying ordinary characters, enclose in single quotes (' ').
ASCII Control	Standard line control keys named: CR Carriage Return BS Back Space TAB Tab TABS Shifted Tab ^TAB Control Tab ESC Escape (Low–Profile) DEL Delete (Low–Profile)
Control Key	Ordinary character or program function keys used with <CTRL>. Specify a control key name as ^*x* (where *x* is an ordinary character or program function key name). For example, use ^Y for CTRL/Y or ^F4 for CTRL/F4 or F4C.

(Continued)

Table 4-4. Key Naming Conventions (Cont.)

Key Type	Description
Program Function	Reserved for user program control. They appear at the top of the keyboard and are named F1–F8 as labeled. (For Low-Profile Model II keyboards, these keys are named F0–F9.) Their up-transition names are F0U–F9U; their shifted names are F0S–F9S; and their control key names are ^F0–^F9 (or F0C–F9C.
Numeric Keypad	Only available on Low-Profile Model II keyboard and the Multinational keyboard. The keypad's numeric keys are named NP0–NP9. The ENTER key is named NPE. Low-Profile Model II keypad symbols are named NP+, NP–, and NP respectively. Keys 0–9, plus (+), and minus (–) can have shifted names (e.g., NP+S), up-transition names, and control key names.
Mouse	Located on the optional mouse. Named M1, M2, and M3; up-transition names are M1U, M2U, M3U. Can have shifted and control key names (e.g., M1S, M1C).

Defining Keys

As we described earlier, Domain/OS provides a set of default function keys and control key sequences defined as DM commands. You can override these definitions or create new ones in either of the following ways:

- Specify the **kd** (key definition) command from the keyboard or in a script.
- Call the system routine **pad_$def_pfk** from a program.

If you wish to redefine your keys, we suggest you look in the directory **/domain_examples/keydefs**. This directory contains some sample key definitions which you may find useful.

When you define keys with the **kd** command during a session on your node, the DM writes the new definitions to one of the following files:

- **key_defs_8bit2** for the Low-Profile Model I keyboard
- **key_defs_8bit3** for the Low-Profile Model II keyboard
- **key_defs_8bit3** for the Multinational keyboard

These files reside in the **user_data** subdirectory of your log-in home directory (see Chapter 3); they apply only to you, not to other node users. The DM checks these files whenever you log in, and sets your personal definitions to reset any of the standard key definitions set up by **/sys/dm/std_keys***(n)* (see Table 4-3).

Definitions made from within a program override those made by **kd** commands; however, they work only within the program's process window. Therefore, keys defined from a program may function differently in different windows. "Controlling Keys from Within a Program" describes how programs control key functions.

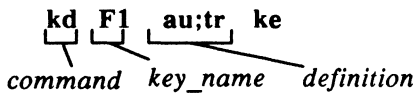
To define a key from the keyboard or from a script, specify the **kd** command in the following format:

kd *key_name definition* **ke**

In the **kd** command format, *key_name* specifies the unique name of the key you want to define. The previous section describes key naming conventions, and Figure 4-3 shows the location and names of keys. Remember, always enclose ordinary character and special character names in single quotes. For example, to define the **Z** key, specify **'Z'**.

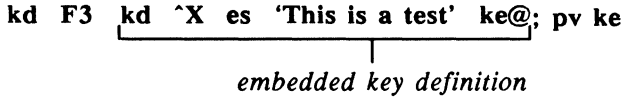
The *definition* argument specifies either a single DM command or a sequence of DM commands that the desired key will perform. (The *Domain Display Manager Command Reference* describes all of the DM commands you can use in key definitions.) When you specify a sequence of commands, either specify each command on a new line (in scripts) or separate each command with a semicolon (;). Always follow the definition argument with the *ke* argument, which signals the end of the **kd** command.

The command in this example defines the program function key, F1, to move the cursor to the end of the previous line in a window:



The definition argument in the example above specifies a command sequence composed of two commands: **au**, which moves the cursor up to the previous line, and **tr**, which moves the cursor to the end of the line. You can specify any number of commands, but you cannot exceed 256 characters in the entire **kd** command.

You can embed key definitions inside other key definitions, and thereby define keys that define other keys. The embedded key definition follows the same rules as any other key definition; however, you must precede the semicolon (;) with an escape character (@) to separate the embedded **kd** command from the next command. The following example shows an embedded key definition:



This command defines the F3 key to perform the following operations when pressed:

- Define CTRL/X to print out the string, "This is a test." (The embedded key definition specifies this function.)
- Invoke the **pv** (pad vertical) command to scroll the current pad one line. (Chapter 5 describes the **pv** command.)

The DM scans embedded key definitions three times when:

1. It makes the outer key definition.
2. It executes the outer key definition and makes the inner key definition.
3. It executes the inner key definition.

To define a key that prompts you for input, specify as part of the definition argument, the input request character (&) as follows:

&'prompt'

where *prompt* specifies the prompt string. The input request character and prompt cause the DM to prompt for part of the definition argument you specified in the key definition. For example, <READ> (R3) has the following default key definition:

kd R3 cv @&'read file: ' ke

When you press <READ>, the DM displays a prompt, "read file: " in the DM input pad and moves the cursor next to it. When you respond by typing the name of a file and pressing <RETURN>, the DM replaces **&read file:** from the key definition with your response. Thus, the **cv** (create view) command opens the file you specify.

> **NOTE:** When you define keys in scripts, you must precede the input request character (&) with the escape character (@). Make sure that you do not include tab characters in DM commands.

When you enter a response to a prompt, the DM remembers the response you typed. The next time you press the key, the DM automatically displays the previous response next to the prompt. (This is why the <READ> and <EDIT> keys remember the last files used.) You can either move the cursor to the right of the previous response and press <RETURN> to enter the response, or delete the previous response and enter a new one.

Deleting Key Definitions

To delete a key definition, specify the **kd** command without a definition argument. For example:

kd F1 ke

deletes the current definition for the key named **F1**. For keys with ordinary character names, the key reverts to its normal graphic value.

Displaying Key Definitions

To display a key's current definition, specify the **kd** command without the definition or **ke** arguments. The current key definition is displayed in the DM output window. The command in the following example displays the definition for the READ key (R3):

```
kd R3
```

Controlling Keys from Within a Program

Domain/OS enables application programs to assume control of various display and keyboard functions. For example, the character font editor, **edfont** (edit font), displays several different menus on your screen that you control with your mouse keys (M1 through M3). When used, the **edfont** program defines how these keys function; the keys do not maintain their normal DM definitions. The DM restores the mouse keys to their normal DM definitions when you end your **edfont** session. The *Domain Display Manager Command Reference* describes the **edfont** character font editor.

For your own applications, you can control key definitions through program calls to the **pad_$def_pfk** and **pad_$dm_cmd** routines. For more information on these system routines, refer to the pad routines section of the *Domain/OS Calls Reference*.

You may find the normal functions of the DM keys useful even when using an application program that has redefined them. With <HOLD>, you can temporarily override the application program's key definitions and use the normal DM definitions.

To override an application program's key definitions, press <HOLD>. By pressing <HOLD> again, you restore the application program's key definitions. Note that this function of <HOLD> is different from the normal DM function of switching a window in and out of hold mode (see Chapter 5).

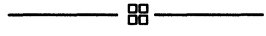
Using DM Command Scripts

A DM script is a file that contains one or several DM commands. You can use DM scripts to perform any of the DM operations described in this manual, such as creating and controlling processes, manipulating pads and windows, editing files, and defining keys.

You execute scripts by specifying the name of the script file with the DM command **cmdf** (command file) as follows:

cmdf *file*

Make sure that you do not include tab characters in DM commands. The start-up scripts discussed in Chapter 3 are examples of DM command scripts that the system uses to set up your node's operating environment. In fact, your node's log-in start-up script uses the **cmdf** command to invoke the DM start-up script that you create.



Chapter 5

Controlling the Display

This chapter describes how to use the DM to control your node's display. Each section describes a set of related screen-management tasks and the DM commands you use to perform them.

You can execute a DM command either from a DM script or interactively by specifying the command in the DM input window. In some cases, you can also execute a DM command by typing a function key or control key sequence.

The command summary tables, at the beginning of each section, list the DM commands, and related function keys and control key sequences, used to perform a specific set of tasks. The predefined control keys that appear in this chapter are the UNIX keys.

Chapter 4 explains how to specify DM commands from the keyboard and from scripts, and how to use function keys and control key sequences. For a complete description of all the DM commands described in this chapter, refer to the *Domain Display Manager Command Reference*.

Controlling Cursor Movement

Moving the cursor is the most basic of all display management operations; it's also the one you'll perform most frequently. You use the cursor to move to a location on the display where you want to perform a specific operation. For example, you can move the cursor to point to the location where you want a DM command to operate, or you can move the cursor into the DM input window to type the name of a command.

This section summarizes the DM commands and control key sequences used to control cursor movement. Table 5–1 lists the commands used to control the cursor. It also shows the predefined directional keys on low–profile keyboards.

> **NOTE:** In this command summary table, the symbols enclosed in parentheses are the unique DM keynames. Refer to Chapter 4 for more information on key names and defining keys. This note applies to all command summary tables in this chapter.

Table 5-1. Cursor Control Commands

Task	DM Command	Predefined Key
Move left one character	**al**	← (LA)
Move right one character	**ar**	→ (LC)
Move up one line	**au**	↑ (L8)
Move down one line	**ad**	↓ (LE)
Set arrow key scale factors	**as** *x y*	None
Move to the beginning of line	**tl**	|← (L4)
Move to end of line	**tr**	→| (L6)
Move to top line in window	**tt**	SHIFT/ ↑ (LDS)
Move to bottom line in window	**tb**	SHIFT/ ↓ (LFS)
Tab to window borders	**twb** [**l, r, t, b**]	None
Move to the beginning of next line	**ad; tl**	CTRL/K
Tab left	**thl**	CTRL/<TAB>
Tab right	**th**	TABS
Set tabs	**ts** [*n1 n2 ...*]	None
Move to DM input pad	**tdm**	<CMD> (L5)
Move to next window on screen	**tn**	<NEXT_WNDW> (LB)
Move to next window in which input is enabled	**ti**	None
Move to previous window	**tlw**	CTRL/~

Creating Processes

When you execute a program on an Apollo node, you run it in a computing environment called a process. Each process that you create is unique, providing a separate computing environment. Since Domain/OS enables you to create multiple processes on your node, you can run several programs simultaneously. You can create and run up to 56 simultaneous processes.

The system associates each process that you create with a **subject identifier (SID).** The SID identifies the owner of a process and consists of the user's name, group, and organization. SIDs enable the system to control user access to processes and other objects on the system. Chapter 14 describes how the system uses SIDs and Access Control Lists (ACLs) to control access to system objects. By default, the system assigns the same SID to each process that you create.

You can create processes that have pads and windows that let you enter data and view program output. Or, you can create processes that run without the use of the display. The type of process you create depends on the program and its application.

To run an interactive program, for example, you create a process with pads and windows. The shell programs that we supply with your system are interactive programs. Each shell that you invoke prompts you for input (shell commands) and displays output.

We also supply a set of special programs called **daemons** that provide you, or a program, with access to some service, such as the use of a peripheral device. These daemons run in processes called servers that you can create using any of the process creation commands described in this chapter. Many of these servers run as background processes without pads or windows.

Table 5–2 summarizes the commands used to create processes.

Table 5–2. Commands for Creating Processes

Task	DM Command	Predefined Key
Createnew process, pads, and windows	**cp** *command*	None
Create new process without pads or windows	**cpo** *command*	None
Create a server process	**cps** *command*	None

Creating a Process with Pads and Windows

To create a process with input and output pads and windows to view these pads, use the **cp** (create process) command in the following format:

[*region*] **cp** [*options*] *command* [*arguments*]

The *region* argument specifies the coordinates of the process window and *command* specifies the pathname of the program you want the process to execute. The process pads and windows that the **cp** command creates enable you to supply input to programs and view program output.

The command in the following example creates a process that executes an interactive program called **counter**. The program prompts for program input and displays its output to the process's transcript pad.

cp -n counter /horace/progs/counter

The **-n** option assigns the process the name **counter**. When **counter** completes (or if you stop the program or process), the input and transcript pads close. To delete the remaining process window, press <EXIT>. Note that in this example, since no region is specified, the DM uses its default window coordinates to create the window. (See the "Defining Default Window Positions" section later in this chapter.)

One process that you'll create frequently is a process that runs a shell program that we supply. You can create a process running your default shell by pressing <SHELL>. You can also run a specific shell by typing the **cp** command with the appropriate pathname at the DM prompt. For example, to run a Bourne shell, type the following:

```
Command:  cp /bin/sh
```

This command creates an input pad and a transcript pad, and opens the input pad as standard input. (**Standard input** is where, by default, a program gets user input.) Figure 5-1 shows a process running the Bourne shell.

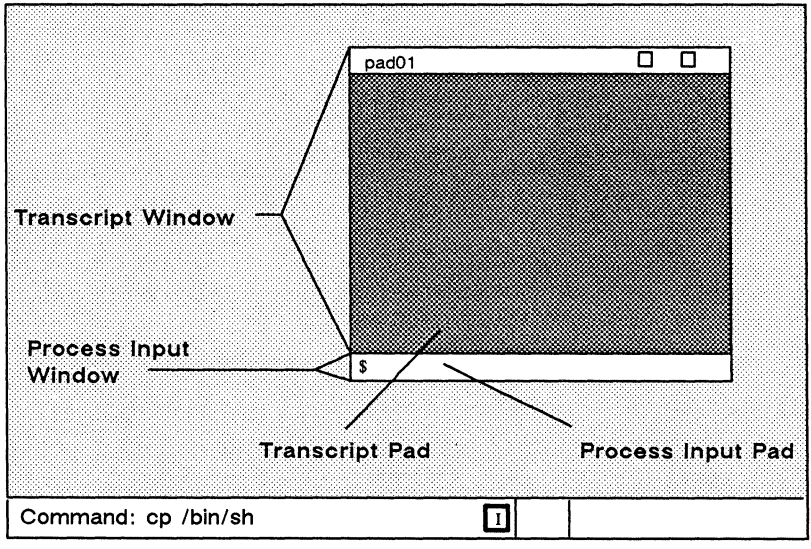


Figure 5-1. A Process Running the Bourne Shell

To stop both the Bourne shell program and its process, press the end-of-file (EOF) key CTRL/D in the shell's process input pad. To close all the windows associated with the shell's process, press <EXIT>. The section "Controlling A Process" describes how to stop programs and processes. The section "Closing Pads and Windows" describes how to close windows.

In many cases it is desireable to run a script for each new pad. This can be done by defining the ENV variable as follows in the DM input window:

```
Command: cp /bin/sh -DENV=~/.shrc
```

This will cause the script **.shrc** in the user home directory to be run when the shell starts up. It is important, however, to reset or unset the ENV variable at the end of the script. This will prevent the shell created in this pad from rerunning the script later on.

Creating a Process without Pads and Windows

To create a background process without associated pads and windows, specify the **cpo** (create process only) command in the following format:

cpo *command* [*options*]

The *command* argument specifies the pathname of the file that you want the process to execute.

When you invoke the **cpo** command, the system assigns the created process the SID of the process that invoked the **cpo** command. The created process runs until the owner of the process logs out.

Suppose you wanted to create a process running the alarm server program to monitor your disk usage, and to warn you when your disk becomes 90% full. To create the process and start the alarm server, specify the following command:

```
cpo /sys/alarm/alarm_server -disk 90
```

In this example, the alarm server runs as a background process on your node. When you log out, the process is killed. *Managing BSD System Software* provides detailed information about the alarm server and other servers.

If you include the **cpo** command in the DM start-up script, **'node_data/startup**, the system assigns the created process the SID **user.server.none**. In this case, the created process continues to run regardless of who logs in or out. You can perform this same function by executing the **cps** command from the DM input window.

Creating a Daemon (Server Process)

You can create a daemon (server process) without pads and windows that runs continually on your node by specifying the **cps** (create process server) command in the following format:

cps *command* [*options*]

The *command* argument specifies the pathname of the program you want the process to execute.

Use the **cps** command when you want to create a server that has an SID of **user.server.none** and runs regardless of whether anyone is logged in. For example, the following command starts the mailbox server **mbx_helper**:

```
cps /sys/mbx/mbx_helper -n mbx_helper
```

In the example above, the **-n** option assigns the process the name **mbx_helper**.

You can invoke **cps** commands from your node's DM start-up script (**startup**). (Chapter 3 describes the start-up script files the system uses when you start your node.) You can also invoke the **cps** command from the DM input window.

Controlling a Process

Once you create a process, you can use the DM's process control commands to stop, suspend, or restart it. Table 5-3 summarizes the DM commands used to control processes, and the predefined UNIX keys for those commands.

Table 5–3. Commands for Controlling a Process

Task	DM Command	Predefined Key
Quit a process (abnormal termination)	**dq -c 9010003**	CTRL/\
Interrupt a running process (normal termination)	**dq -i**	CTRL/C
Stop or blast a process	**dq** [**-s**|**-b** *n*]	None
Suspend execution of a process	**dq -c 120028**	CTRL/Z
Resume execution of a suspended process	**dc**	None

Interrupting and Stopping a Process

To interrupt a process, position the cursor inside the window of the associated process and use the **dq** (debug quit) command in the following format:

dq -i

Using the **-i** option generates an interrupt signal to the running process. This causes normal termination of the current process and returns the process to the calling program (usually the shell). It produces the same effect as using CTRL/C.

To stop a process, causing abnormal priority termination, position the cursor inside the window of the process and specify the **dq** command without any options. This is equivalent to typing CTRL/\.

To stop the current process and close any open streams, files, and pads, specify the following DM command:

dq -s

To delete the remaining window, move the cursor inside the window and press <EXIT>.

If you want to end a UNIX shell process, move the cursor to the shell's process input window and press the end-of-file (EOF) key, CTRL/D. Typing CTRL/D in the shell's process input window signals the completion of input and usually stops both the shell and the process. You may find this method easier than using **dq -s**.

Suspending and Resuming a Process

You can temporarily suspend a process and then restart it using the **ds** (debug suspend) and **dc** (debug continue) commands.

To suspend a process, position the cursor inside the process window; then specify the **ds** command. Later, to restart the process, position the cursor inside the process window and specify the **dc** command.

Although these DM commands may be helpful, you may find it more convenient and appropriate to use the job control feature available in the C shell (see Chapter 8 for further detail).

Creating Pads and Windows

To read or edit a file, you must create a pad to hold it and a window to view it. Table 5-4 lists the DM commands and predefined keys used to create pads and windows for editing and reading files.

Table 5-4. Commands for Creating Pads and Windows

Task	DM Command	Predefined Key
Create an edit pad and window	**ce** *file*	<EDIT> (R4)
Create a read-only window	**cv** *file*	<READ> (R3)
Create a copy of an existing pad and window	**cc**	None

Before you can use the commands that create pads and windows, you should understand just how the DM determines what boundaries to assign to a new window.

When a window's size or position on the screen is changed in any way, the DM calculates the new boundaries of the window based on a pair of points on the screen called a **point pair**. (Usually, you define the first point in the pair with the **dr** command, and the second point by the current cursor position. You may also provide absolute point coordinates as described in the "Defining Points and Regions" section in Chapter 4.)

Each point in a point pair may specify either a new or existing edge of a window, or a new or existing corner of a window. The DM creates a new window based on the relationship between the x- and y-coordinates of the two points.

DM Rules for Defining Window Boundaries

The relationship between the two points in the point pair affects the actions of the DM window-creation commands, **cp**, **ce**, **cv**, **cc**, and the window-movement commands, **wm**, **wme**, **wg**, and **wge** (see the "Managing Windows" section). The list below shows how the DM defines window boundaries according to the points given for window-creation and window-movement commands.

For points that differ in both x- and y-coordinates:

Create	Each set of coordinates form opposing corners of the window.
Move	The first point selects the nearest unobscured corner (this corner must be visible) and the DM repositions the corner at the second point.

For points that are equal:

Create	Create a 512 by 512 window centered as nearly as possible to the given cursor position.
Move	Select unobscured corner nearest the given point, and move the corner to that point.

For points that have equal y-coordinates:

Create Create a window bounded by the given x-coordinates, the top of the display, and the DM command window. In other words, create a full vertical window.

Move Select the unobscured vertical edge nearest to the first point and change the x-coordinate of that edge to that of the second point.

For points that have equal x-coordinates:

Create Create a window bounded by the given y-coordinates and each side of the display. In other words, create a full horizontal window.

Move Select the unobscured horizontal edge nearest to the first point, and change the y-coordinate of that edge to that of the second point.

When only one point is given (no **dr** is specified):

Create The DM uses one of its five default window regions (see the "Defining Default Window Positions" section), or it determines the position by the last window creation or deletion command as follows:

- If the last command was window deletion (**wc**), the default region is the same as that for the deleted window.
- If the last command was a successful window-creation command, the default region is the next third of the screen
- If the last command was an unsuccessful window-creation command, the default region is the same as that specified in the unsuccessful command.

Move Grow is illegal; move acts as if both points are equal.

Creating an Edit Pad and Window

To create an edit pad and window, specify the **ce** (create edit) command in the following format:

[*region*] **ce** *file*

The *file* argument specifies the name of the file you want to edit. If the file you specify exists, the **ce** command opens the file for editing. If the file does not exist, the **ce** command creates a new file, assigns it the pathname you specified, and opens it for editing. Note that the **ce** command does not create a process; it opens a file for editing within the current DM process.

Once you create an edit pad, you can use the DM edit commands to manipulate the text that appears on the pad. Chapter 6 describes how to use the DM edit commands to edit pads.

You can also create an edit pad and window using <EDIT>. When you press <EDIT>, an "edit file: " prompt appears in the DM input window, and the DM moves the cursor next to the prompt. To edit a specific file, type the file's pathname next to the prompt (as shown in Figure 5–2), and press <RETURN>.

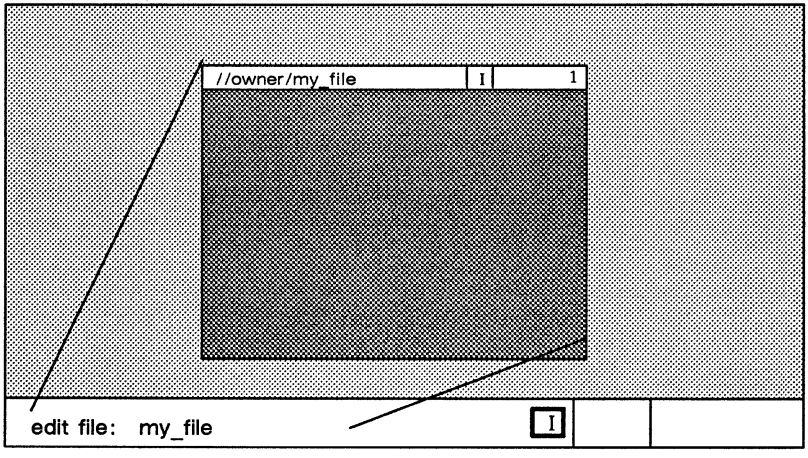


Figure 5–2. Creating an Edit Pad and Window

Creating a Read-Only Pad and Window

A read-only pad and window is identical to an edit pad and window with one exception: you cannot make changes to a read-only pad; you can only read it. (Note, however, that you can copy text from a read-only pad.)

To create a read-only pad and window, specify the cv (create view) command in the following format:

[*region*] **cv** *file*

The *file* argument specifies the name of the file you want to read. If the file you specify exists, the **cv** command opens the file and displays its contents. If the file does not exist, the DM displays the following error message:

```
(cv)  filename - Name not found
```

Note that the **cv** command does not create a process; it opens a file for reading within the current DM process.

If the file you want to read is currently active in another window, you can create another new pad and window to read it. You cannot, however, edit a file while anyone else on the network has it open for editing.

On occasion, you may create a read-only pad and window and decide that you would like to make changes to the file. Instead of creating a new edit pad and window for the file, you can specify the DM command, **ro** (set read/write mode), to change the read-only pad to an edit pad. Chapter 6 describes how to use the **ro** command to set a pad's read/write mode.

You can also create a read-only pad and window using <READ>. For a description of how to use <READ>, see Chapter 4.

Copying a Pad and Window

With the **cc** (create copy) command, you can create a copy of an existing pad and window and display it at a specific area on the screen. Figure 5–3 illustrates how to use the **cc** command to copy a pad and window.

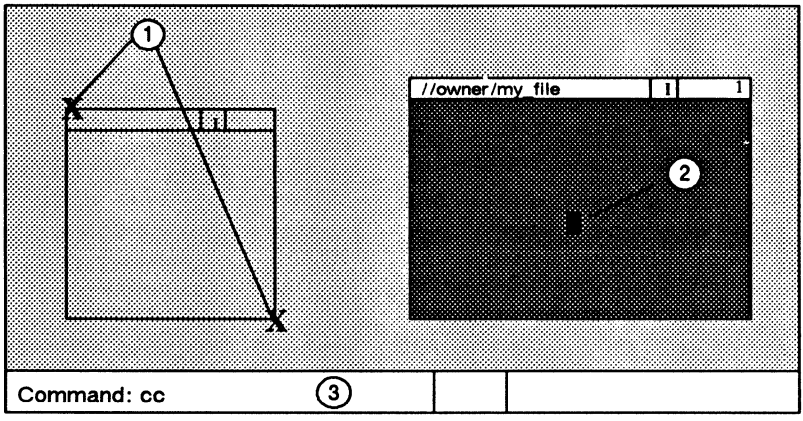


Figure 5–3. Copying a Pad and Window

The numbers in Figure 5–3 correspond to the following steps:

1. Mark opposite corners of the new window. To mark each corner: first move the cursor to the point on the screen where you want the corner to appear, then either press <MARK> or specify the **dr** command. (Chapter 4 describes how to use **dr** <MARK> to mark regions on the display.)

2. Move the cursor inside the window you want to copy.

3. Specify the **cc** command.

This procedure creates a copy of the pad and window and displays it at the location on the screen that you marked. If you issue the **cc** command without marking the display region, the DM determines the location according to the rules described earlier in the "Creating Pads and Windows" section.

Closing Pads and Windows

When you finish reading or editing a pad, you can close the pad and window using any of the commands listed in Table 5-5.

Table 5-5. Commands for Closing Pads and Windows

Task	DM Command	Predefined Key
Close window and pad; update file	**pw; wc -q**	<EXIT> (R5)
Close window and pad; no update	**wc -q**	<ABORT> (R5S)
Close (delete) a window	**wc [-q|-f]**	None

To delete (quit) a read-only or edit pad and associated windows, position the cursor inside the window and press <ABORT> or specify the following command:

wc -q

The **-q** option causes **wc** to delete the pad and window without saving the contents of the pad. If you modified the edit pad, you'll receive the following message in the DM input window asking you to confirm your request to quit:

```
File Modified. OK to quit?
```

If you respond by typing **y** or **yes** followed by <RETURN>, the **wc** command deletes the pad and window without saving the contents of the pad. If you respond **n** or **no**, the system ignores the quit request and returns the cursor to the edit pad.

If you modify an edit pad and want to save its contents (write its contents to a file), press <EXIT> or specify the **pw** command without any arguments.

The **pw** (pad write) command copies the edited pad to a file that has the same name as the original file. The system saves the contents of the original pad in a file with the same name and the added suffix **.bak**. Once you've saved the pad, use **wc** to close the edit window.

Managing Windows

Window control commands enable you to change the size, position, and characteristics of windows on the screen. You can use window control commands to manage edit pad windows, or process windows. Table 5–6 summarizes the window control commands.

Table 5-6. Commands for Managing Windows

Task	DM Command	Predefined Key
Change window size	**wg**	None
Change window size with rubberbanding	**wge**	<GROW> (LA3)
Move a window	**wm**	None
Move a window with rubberbanding	**wme**	<MOVE> (LA3S)
Set scroll mode	**ws [-on|-off]**	None
Set autohold mode	**wa [-on|-off]**	None
Scroll and autoholf mode	**wa; ws**	CTRL/A
Set hold mode	**wh**	<HOLD> (R6)
	wh -on	CTRL/S
	wh -off	CTRL/Q
Define position of default window*n*	**wdf** [*n*]	None
Acknowledge alarm	**aa**	None
Acknowledge alarm and pop window	**ap**	None

Changing Window Size

Once you create a window on your screen, you can enlarge or shrink it with the **wge** (window grow echo) command.

As shown in Figure 5–4, the **wge** command displays a flexible border, or rubberband, that changes as you move the cursor to enlarge or shrink the window. The rubberband shows you the size and shape the window will become when you complete the operation.

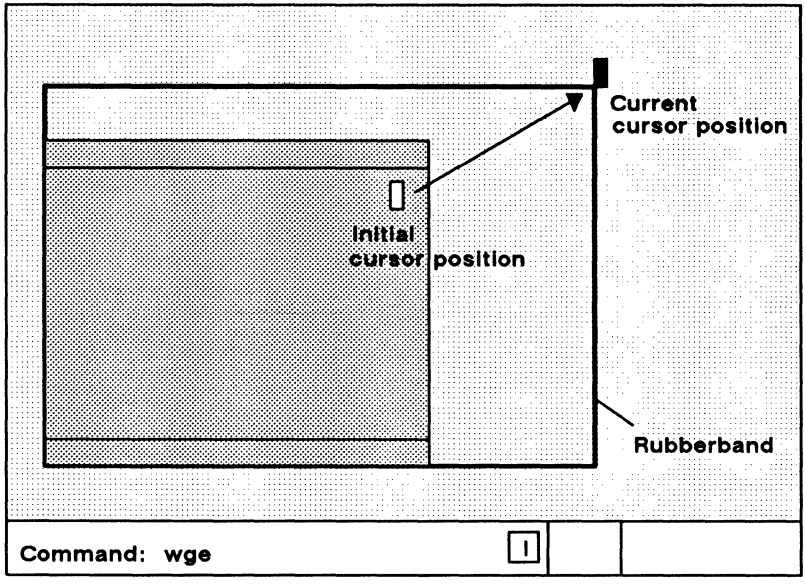


Figure 5–4. Growing a Window Using Rubberbanding

Use the following procedure to change the size of a window:

1. Move the cursor to the window corner or edge you want to move.

2. Press <GROW> or specify the **wge** command. A rubberband border appears.

3. Move the cursor to stretch or shrink the rubberband until the rubberband matches the new size you want for the window.

4. Either press <MARK> or specify the **dr** command to complete the operation.

To cancel the procedure at any time, press CTRL/X or specify the **abrt** command.

If you have a mouse, you can change the size of a window by using the left mouse key. To use the mouse to change the size of a window, perform the following procedure:

1. Move the cursor to the window corner or edge you want to move.
2. Press and hold the left mouse key. A rubberband border appears.
3. Holding the left key down, move the cursor to grow or shrink the window.
4. When the rubberband matches the new size you want for the window, release the left mouse key.

Moving a Window

To move a window to another location on the display, use the **wme** (window move echo) command. The **wme** command, like the **wge** command, uses a rubberband border to show you the exact position the new window will occupy.

Use the following procedure to move a window:

1. Move the cursor to any corner of the window you want to move.
2. Press <MOVE> or specify the **wme** command. A rubberband border appears.
3. Move the cursor until the rubberband is at the new window position.
4. Either press <MARK> or specify the **dr;echo** command sequence to complete the operation.

To cancel the procedure at any time, press CTRL/X or specify the **abrt** command.

Pushing and Popping Windows

As you create multiple windows on your screen, you may begin to stack windows one on top of another. Some windows will partially obscure or completely hide others. To view hidden windows, use the **wp** (window pop) command in the following format:

wp [*options*] [*window_name*]

The **wp** command either pops a window to the top of the stack or pushes a window to the bottom of the stack, depending on where you position the cursor. Figure 5–5 illustrates how to push and pop windows.

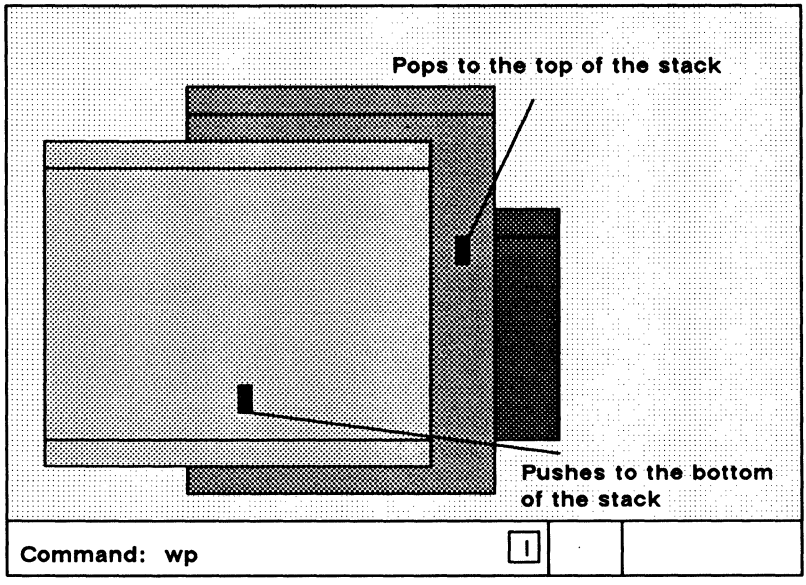


Figure 5–5 Pushing and Popping Windows

If you position the cursor in a partially obscured window, the **wp** command pops the window to the top of the stack. If you position the cursor in a completely visible window (the window on top), **wp** pushes the window to the bottom of the stack.

Use the following procedure to push or pop windows:

1. Position the cursor inside the window you want to push or pop.

2. Pop or push the window by either pressing <POP> (on low-profile type keyboards only), specifying the **wp** command.

You can also refer to a window you want to push or pop by specifying the name of the window. To specify a window name, either enter it as an argument to the **wp** command, or point to window name as follows:

1. Use the cursor to point to a text string that contains the name of the window you want to push or pop.

2. Press <MARK>, or specify **dr** to mark the window name.

3. Specify the **wp** command.

This second method is useful when you're displaying a list of all windows that you currently have open (see the description of the **cpb** command in the "Displaying the Members of a Window Group" section later in this chapter).

Changing Process Window Modes

The DM provides several modes that control how the DM inserts text into process input windows, and how process transcript windows display program output. Table 5–7 describes these modes.

You control window modes by positioning the cursor inside the process window and specifying window mode control commands. If you specify a command without any options, the command toggles the mode setting (turns it on or off depending on its current state).

Table 5–7. Process Window Modes

Mode	Description
Insert	Insert text in the input window rather than overstrike
Scroll	Output scrolls one line at a time.
Hold	Content of the window does not change when the program sends output to the pad.
Autohold	Window automatically enters hold mode.

The window legend at the top of the process window displays a letter code that indicates which modes are on. Figure 5–6 shows the mode indicators and other components that make up the process window legend.

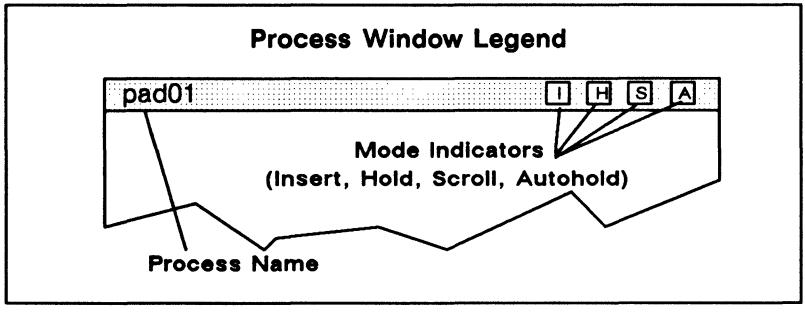


Figure 5–6. Process Window Legend

By default, the window legend displays the letter I to show that the process input window is in **insert mode**. In insert mode, the DM inserts characters you type at the current cursor position. The remainder of the line moves to the right to make room for new characters.

With insert mode turned off, the process input window is in overstrike mode, in which characters you type replace those under the cursor.

To turn insert mode on or off, specify the **ei** command in the following format:

ei [-on | -off]

If you do not specify an option, **ei** toggles the current mode.

To turn scroll mode on or off, specify the **ws** (window scroll) command in the following format:

ws [-on | -off]

With scroll mode turned on, the window displays output one line at a time as the transcript pad moves beneath the window. With scroll mode turned off, output does not appear a line at a time. Instead, when the program finishes sending output to the transcript pad, the window automatically displays the end of the pad and any new output.

Initially, all transcript pad windows have scroll mode turned on. The window legend at the top of the window displays the letter **S** when scroll mode is on.

To turn hold mode on or off, specify the **wh** (window hold) command in the following format:

wh [-on | -off]

When you turn hold mode on, the DM freezes the position of the transcript pad beneath the window. The window will not display new program output until you release the pad by turning hold mode off. When you turn hold mode off again, the window automatically displays the end of the transcript pad and any new program output.

Initially, all transcript pad windows have hold mode turned off. With hold mode turned off, the window automatically displays new output as the pad moves beneath it. The window legend displays the letter **H** when hold mode is on. You can also turn hold mode on or off by pressing <HOLD>, or by pressing CTRL/S (hold mode on) and CTRL/Q (hold mode off).

To turn autohold mode on or off, specify the **wa** (window autohold) command in the following format:

wa [**-on** | **-off**]

With autohold mode turned on, the window automatically turns hold mode on under either of the following conditions:

- A full window of output is available and none of it has been displayed.
- A form feed or create frame operation is output to the pad. In this case, the window displays the output preceding the form feed. When the window exits from hold mode, the output following the form feed or create frame operation starts at the top of the window.

To continue displaying output, turn hold mode off.

Initially, all transcript pad windows have autohold mode turned off. The window legend contains an A when autohold mode is on. You can also turn autohold mode on or off by pressing CTRL/A (which invokes the commands **wa;ws**).

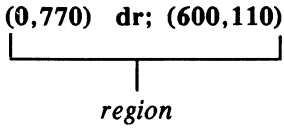
Defining Default Window Positions

The DM uses default window positions to determine where to display the first five windows you create. To define any of the DM's five default window positions, specify the **wdf** (window default) command in the following format:

[*region*] **wdf** [*n*]

The *region* argument specifies the position that the window will occupy on the screen (see the "Specifying Points on the Display" section in Chapter 4), and *n* specifies the indentification number of the default window you are defining. If you omit *n*, the **wdf** command causes the DM to discard any current window information and begin creating windows using its default window boundaries.

The command in the following example defines the window position for default window four. Note the format of the region definition.



If you want to use your own default positions for each log-in session, include **wdf** commands in your DM start-up script (**startup_dm**). Once you've defined your default window positions, you should add the command **wdf;cms**. This command instructs the DM to use the first **wdf** command to set up the default position for the first window you create. Otherwise, the DM uses the last **wdf** command in your script to determine the default position of the first window you create. For more information on DM start-up scripts, see "Understanding the System at Login" in Chapter 3.

Responding to DM Alarms

Whenever the DM writes output to a partially obscured or hidden window, it sounds an alarm and displays a small pair of bells in the alarm window. To respond to an alarm, specify either the **aa** or **ap** commands.

The **aa** command acknowledges the DM alarm by turning off the current alarm and enabling further alarms (which may already be waiting).

The **ap** command acknowledges the DM alarm and pops to the top of the stack, the window to which the alarm pertains. This command is particularly useful when the window is completely hidden, and you can't point to it.

Moving Pads Under Windows

The DM pad control commands enable you to move a pad under a window. Table 5-8 summarizes the pad control commands.

Table 5–8. Commands for Moving Pads

Task	DM Command	Predefined Key
Move top of pad into window	**pt**	None
Move cursor to first character in pad	**pt;tt;tl**	CTRL/T
Move bottom of pad into window	**pb**	None
Move cursor to last character in pad	**pb;tb;tr**	CTRL/B
Move pad *n* pages	**pp** [**-**]*n*	↑ ↓ (LD, LF)
Move pad *n* lines	**pv** [**-**]*n*	SHIFT/ ↑ (L8S) SHIFT/ ↓ (LES)
Move pad *n* characters	**ph** [**-**]*n*	← → (L7, L9)
Save transcript pad in a file	**pn**	None

Moving to the Top or Bottom of a Pad

Two DM commands enable you to move from the current position in a pad to the top or bottom of a pad. The **pt** (pad top) command moves the top line of a pad to the top of the current window. The **pb** (pad bottom) command moves the bottom line of a pad to the bottom of the current window. Neither command accepts arguments or options.

We also provide two predefined control key sequences that perform the same functions as the **pt** and **pb** commands; they also move the cursor to either the first or last character in the pad. To move the cursor to the first character in the pad, press CTRL/T (defined as

the command sequence **pb;tt;tl**). To move the cursor to the last character in the pad, press CTRL/B (defined as the command sequence, **pb;tb;tr**).

Scrolling a Pad Vertically

You can scroll a pad up or down by a specified number of lines or pages using the vertical scroll commands or associated function keys. To scroll a pad by pages, specify the **pp** (pad page) command in the following format:

pp [-]*n*

The *n* argument specifies the number (or fraction) of pages you want to scroll. A positive *n* (*n*) scrolls the pad up *n* pages; a negative *n* (*-n*) scrolls the pad down *n* pages. The DM considers a page the smaller of the following values:

- The number of lines that fit in a window.
- The number of lines between the bottom of the window and the next form feed or frame.

The command in the following example scrolls the pad down one and one-half pages:

pp -1.5

We also provide two predefined keys that scroll a pad either up or down one-half page at a time. Figure 5-7 shows the location of these keys.

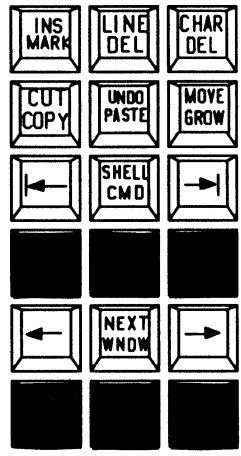


Figure 5–7. Location of Pad Scroll Keys

To scroll a pad by lines, specify the **pv** (pad line) command in the following format:

pv [–]*n*

The *n* argument specifies the number of lines you want to scroll. A positive *n* (*n*) scrolls the pad up *n* lines; a negative *n* (–*n*) scrolls the pad down *n* lines.

You can also use the two predefined function keys shown in Figure 5–7 to scroll a pad either up or down one line at a time. To scroll one line at a time, press <SHIFT> and the pad scroll key simultaneously.

Scrolling a Pad Horizontally

To scroll a pad horizontally by a specified number of characters, use the **ph** (pad horizontal) command or its associated function keys. The **ph** command has the following format:

ph [–]*n*

The *n* argument specifies the number of characters you want to scroll. A positive *n* (*n*) scrolls the pad to the left n characters; a negative *n* (*-n*) scrolls the pad to the right *n* characters.

You can also use two predefined function keys to scroll a pad either right or left 10 characters. Figure 5-7 shows the location of these keys.

Saving a Transcript Pad in a File

Normally, the DM deletes a transcript pad when you stop the pad's process and delete all windows. To keep a log of the current transcript pad and save the log in a file, specify the **pn** (pad name) command in the following format:

pn *file*

The *file* argument specifies the name of the file where the DM saves the contents of the pad. You must specify a file on your node; you cannot use a name cataloged on another node.

The **pn** command stores the current transcript pad in a file that remains opened and locked until you stop the process and delete all windows. Once you specify the **pn** command, the DM saves all current and subsequent output written to the pad.

Using Window Groups and Window Icons

The DM provides several commands that enable you to create **window groups**, make these groups invisible, or use **icons** to represent them. Table 5-9 summarizes the commands used to control window groups and icons.

Table 5-9. Commands for Controlling Window Groups and Icons

Task	DM Command	Predefined Key
Create or add to a window group	**wgra** *grp_name* [*entry_name*]	None
Remove a window from a window group	**wgrr** *grp_name* [*entry_name*]	None
Make windows invisible	**wi** [*entry_name*]	None
Change windows to icons	**icon** [*entry_name*] [*options*]	SHIFT/<POP> (R1S)
Set icon positioning and offset	**idf**	None
Display list of windows in group	**cpb** *grp_name*	None

Creating and Adding to Window Groups

When you create a window group, you establish a group name and assign windows to the group. You can then make the window group invisible or represent the group with icons by specifying the group name. Groups can contain individual windows, as well as other groups of windows.

To create a window group or add a window to an existing group, specify the **wgra** (window group add) command in the following format:

wgra *group_name* [*entry_name*]

The *group_name* argument specifies the name of the group you want to create or add to, and *entry_name* specifies the name of the window or window group you want to add. For process windows, *entry_name* specifies the process name that appears in the window legend; for edit pad windows, *entry_name* specifies the pathname that appears in the window legend.

You must specify the *group_name* argument when you use this command. If you omit the *entry_name* argument, **wgra** uses the name of the window where you last positioned the cursor.

The commands in the following example create a window group:

```
wgra shell_windows pad01
wgra shell_windows pad02
wgra shell_windows pad03
```

The first command creates a window group named **shell_windows** and adds the window named **pad01** to the group. The remaining commands add additional windows (**pad02** and **pad03**) to the **shell_windows** group.

Removing Entries from Window Groups

To remove an entry (window or window group) from a window group, specify the **wgrr** (window group remove) command in the following format:

wgrr *group_name* [*entry_name*]

The *group_name* argument specifies the name of the group that contains the entry you want to remove, and *entry_name* specifies the window name or window group name you want to remove. You must specify the *group_name* argument when you use this command. If you omit the *entry_name* argument, **wgrr** uses the pathname of the window where you last positioned the cursor.

The command in the following example removes a window named **pad01** from the group named **shell_windows**:

```
wgrr shell_windows pad01
```

Making Windows Invisible

To control whether a window or window group is visible or invisible, specify the **wi** (window invisible) command in the following format:

wi [*entry_name*] [**-w**] [**-i**]

The *entry_name* argument specifies the name of the window or window group you want to make visible or invisible. If you omit the *entry_name* argument, **wi** uses the pathname of the window where you last positioned the cursor.

The **-w** option forces the window or group to appear as a window; the **-i** option forces the window or group to become invisible. If you specify the **wi** command without either of these options, **wi** toggles the setting (makes the window or group visible or invisible, whichever is the opposite of its current state).

The command in the next example makes the window group **shell_windows** invisible:

```
wi shell_windows -i
```

Using Icons

You use icons to represent a window or group of windows on your display. Because icons are small, they enable you to keep windows and window groups easily accessible without having them open on the display.

Icons are very similar to the windows they represent. For example, you can move icons with the **wme** command (see the "Moving a Window" section discussed earlier), or you can set the position on the screen where icons will appear by default. You cannot, however, change the size of an icon on the display.

The DM displays an icon as a small window containing a specific icon symbol. The icon symbol describes the type of information the related window or group contains. Figure 5-8 shows the default icon for shell process windows.

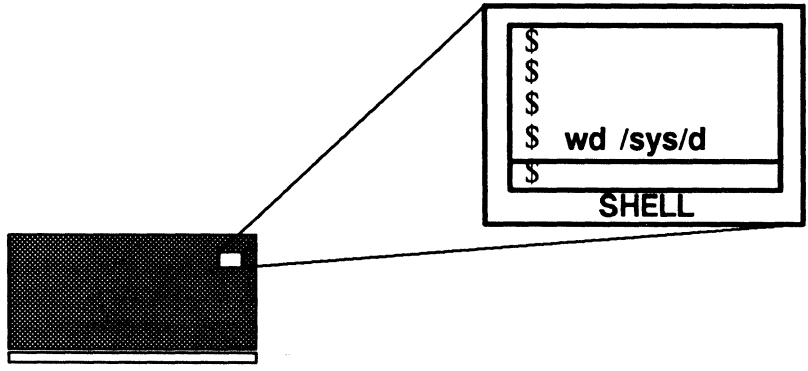


Figure 5-8. Default Icon for Shell Process Windows

To change a window or window group into an icon, or to change an icon into the window or group it represents, use the predefined key SHIFT/<POP> or specify the **icon** command as follows:

icon [*entry_name*] [**-i**] [**-w**] [**-c** '*char*']

The *entry_name* argument specifies the name of the window or window group you want to change into an icon, or change back into a window. If you specify the name of a window group as the entry name, the **icon** command changes each window in the group. If you omit the *entry_name* argument, **icon** uses the window where you last positioned the cursor.

The **-w** option forces the specified window or window group to appear as a window; the **-i** option forces the specified window or group to change to an icon. If you specify the **icon** command without either of these options, **icon** toggles the setting (changes the window or group to the opposite of its current state). The easiest way to change individual windows and icons is to position the cursor inside the window or icon and specify the **icon** command.

The **icon** command also accepts the **-c** option that allows you to specify which icon you want to use. Before we look at an example, let's look at how the system uses icons, and where it stores them.

The system uses certain default icons that we supply to represent specific types of windows. For example, whenever you change a shell process window into an icon, the system, by default, uses the

icon shown in Figure 5–8. Similarly, the system uses a special edit icon to represent read/edit windows. Many application programs that we supply also represent their specific process windows with their own specific default icons.

The system stores default icons in a font file, **/sys/dm/fonts/icons**. (This file is not an ASCII file; you cannot read it.) You can examine this file by using the **edfont** (edit font) program described in the *Domain Display Manager Command Reference*. Use **edfont** to create your own icons or change those the system uses by default.

Each icon in the font file **icons** is associated with a specific keyboard character. For example, the default shell icon is associated with the lowercase **s** character. When you create an icon, you first choose a character, and then use **edfont** to transform the character into an icon symbol. (This is how we created the default icons that the system and various application programs use.) To use your own icon once you've created it, specify its associated character name with the **-c** option.

The **-c** option allows you to specify the character associated with the icon you want to use. For example, suppose you used **edfont** to create your own icon associated with the uppercase **F** character in the **icons** file. To use this icon to represent the read/edit window **june_report**, use the following command:

```
icon june_report -i -c 'F'
```

In this example, the **icon** command directs the DM to change the read/edit window **june_report** into an icon. Normally, the DM uses the default icon for read/edit windows. The **-c** option directs the DM to use the icon that is associated with the character **F** in the file **/sys/dm/fonts/icons** instead of the default read/edit icon.

Setting Icon Default Position and Offset

The DM allows you to set the position of an icon on your screen and specify an **offset** that the DM uses to determine the positions of the next icons you create. The offset value specifies the position of new windows relative to the position of the previous icon.

By default, the DM displays icons in a vertical line along the right side of landscape displays. The default offset is the width of one icon (60 bits), vertically for landscape displays.

With the **idf** (icon default) command, you can change the default positioning and offset of an icon, or to establish the position of an icon you create in a script. You can use the **idf** command in any of the following ways:

- Move the cursor to the desired default icon position. Press <MARK> or specify the **dr** command to mark the position. Specify the **idf** command to set the new position. Since you did not specify an offset value, the DM places any new icons that you create at this one position.

- Move the cursor to the desired default icon position. Press <MARK> or specify the **dr** command to mark the position. Move the cursor to indicate the offset vector for the next icon. Specify the **idf** command to set the new position and offset.

- Specify the icon position and offset explicitly in the following command line format:

  (*position*) **dr**; (*offset*) **idf**

  The *position* argument specifies the x- and y-coordinates of the icon position and *offset* specifies the coordinates of the offset vector. For example, the following command line sets an icon position and offset:

  (800,10) dr; (850,60) idf

  This command sets the position for the first icon at bit position **(800,10)**. The next icon will appear at bit position **(850,60)**, an offset of **(50,50)** from the original position. Refer to "Defining Points and Regions" in Chapter 4 for further information.

Displaying the Members of a Window Group

To display a list of windows in a specific group, use the **cpb** (create paste buffer) command in the following format:

cpb *group_name*

The *group_name* argument specifies the name of the window group you want to list. The *group_name* refers to a paste buffer that contains the names of the windows in the group. The **cpb** command creates a window to the paste buffer you specify as the *group_name* and displays the paste buffer's contents. For example:

cpb my_group

This command displays the names of all the windows in the window group **my_group**. A paste buffer named **my_group** contains these window names.

The DM automatically creates three special paste buffers to help you manage your windows and icons. Table 5–10 describes these paste buffers.

To list the contents of one of these special paste buffers, specify the **cpb** command with the special *group_name* as follows:

cpb invis_group

This command opens the paste buffer **invis_group** that contains the names of all the windows you've made invisible.

Table 5–10. Window Paste Buffers

Mode	Description
invis_group	Contains the pathnames of all the windows that you've made invisible.
icon_group	Contains the pathnames of all the windows represented by icons.
all_group	Contains the pathname of every window open on your node, including: shell process windows, DM windows, visible and invisible windows, and windows represented by icons.

Chapter 6

Editing a Pad

Chapter 5 describes how to create pads and windows to read and edit files. This chapter describes how to use the DM to control the characteristics of edit pads, and how to edit text.

Each section in this chapter describes a set of editing tasks and the DM commands you use to perform them. You can execute a DM command either from a DM script or interactively by specifying the command in the DM input window. In many cases, you can execute a DM editing command by typing a function key or control key sequence.

The command summary tables at the beginning of each section list the DM commands, related function keys, and control key sequences used to perform a specific set of editing tasks. Note that the predefined keys listed in these tables apply only to low–profile keyboards.

Chapter 4 explains how to specify DM commands from the keyboard and from scripts, and how to use function keys and control key sequences. For a complete description of all the DM editing commands described in this chapter, refer to the *Domain Display Manager Command Reference*.

Setting Edit Pad Modes

All edit pads are controlled by a very important feature of the DM: the modes in which the DM currently operates. The modes determine whether you can make changes to the material in the pad, and whether the DM either inserts characters that you type or overstrikes them. Table 6–1 summarizes the DM commands used to change edit pad modes.

Table 6–1. Commands for Setting Edit Modes

Task	DM Command	Predefined Key
Set read/write mode	**ro [-on|-off]**	SHIFT/<AGAIN>
Set insert/overstrike mode	**ei [-on|-off]**	<INS> (L1S)

Figure 6–1 shows the window legend for edit pads. The edit pad window legend provides information about a window's characteristics, such as the pathname of the file and current window modes. The edit pad window legend also displays the **line number** of the line at the top of the window and the **horizontal offset** (column number), which indicates the number of columns the window has been scrolled sideways over the pad. The horizontal offset number appears only when you scroll the window sideways over the pad.

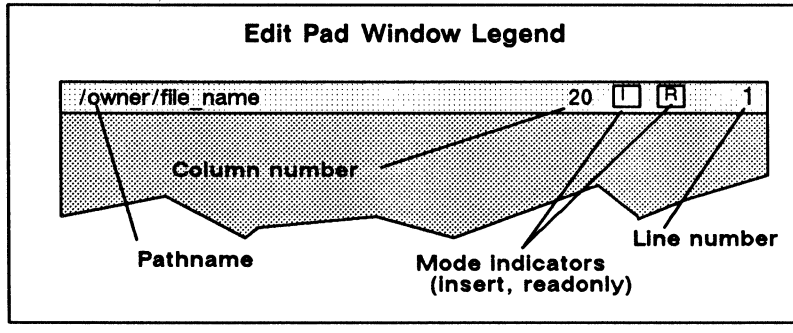


Figure 6–1. The Edit Pad Window Legend

Setting Read/Write Mode

Edit pads can be in **read-only mode** or **write mode**. In read-only mode, you cannot write to or make changes to the text in a pad. However, you can copy, search for, and scroll the text. In write mode, you can write to a pad and change text using all of the editing commands described in this chapter.

When a pad is in read-only mode, the letter **R** appears in the window legend as shown in Figure 6-1. The **R** disappears in write mode.

To turn read-only mode either on or off, specify the **ro** (set read only mode) command in the following format:

ro [-on | -off]

The **-on** option instructs **ro** to set the pad to read-only mode. The **-off** option causes **ro** to set the pad to write mode (i.e., it turns read-only mode off). If you do not specify an option, the **ro** command toggles the current mode setting.

If you've modified the text in a pad, you cannot change the pad to read-only mode without first writing the changes to a disk file (saving the file). The **pw** command, described in the "Updating an Edit File" section, allows you to write your changes to a disk file without closing the pad and window.

Setting Insert/Overstrike Mode

The DM has two modes to control how text is added to a pad: insert mode or overstrike mode. In insert mode, the DM inserts characters you type at the current cursor position. The remainder of the line moves right to make room for the new characters.

In **overstrike mode**, characters you type replace, or "overstrike," those under the cursor. Overstrike mode is useful when you want to enter text into a preformatted file without disrupting the file's format.

When a pad is in insert mode, the letter **I** appears in the window legend as shown in Figure 6–1. The **I** disappears in overstrike mode. All pads are initially in insert mode, although this is irrelevant if the pad is also read–only.

To turn insert mode either on or off, specify the **ei** (set edit/overstrike mode) command in the following format:

ei [–on | –off]

The **–on** option instructs **ei** to set the current pad to insert mode. The **–off** option causes **ei** to set the pad to overstrike mode (i.e., it turns insert mode off). If you do not specify an option, the **ei** command toggles the current mode.

You can also toggle the current mode by pressing <INS>. This key invokes the **ei** command without options.

Inserting Characters

Any pad that is in write mode automatically accepts anything that you type at the keyboard as input to that pad. The commands listed in Table 6–2 perform special insertion functions.

Table 6–2. Commands for Inserting Characters

Task	DM Command	Predefined Key
Insert string at cursor	**es** *'string'*	Default DM operation
Insert newline character	**en**	<RETURN>
Insert tab character	None	<TAB>
Insert new line after current line	**tr;en;tl**	<F1>
Insert raw (noecho) character	**er** *nn*	None
Insert end–of–file mark	**eef**	CTRL/D

Inserting a Text String

When a pad is in write mode, the DM inserts any text character you type at the current cursor position. This is the default Display Manager action. If you try to type text into a read-only pad, the DM displays an error message in the DM output window.

To insert a text string at the current cursor position, specify the **es** (edit string) command in the following format:

es *'string'*

The *'string'* argument is the text that you want to insert. Enclose the text in single quotes (').

The **es** command inserts a string of text at the current cursor position. Since text insertion is the default action, you'll probably find this command most useful in key definition commands where you want some text written out when the key is pressed. Chapter 4 describes how to define keys to perform DM functions.

Inserting a Newline Character

The newline character marks the end of a line. To insert a newline character at the current cursor position, press <RETURN> or specify the **en** (edit newline) command. When you insert a newline character, the cursor moves to the beginning of the next line.

Inserting a New Line

To insert a new blank line following the current line, specify the following command sequence:

tr;en;tl

The **tr** (to right) command moves the cursor to the end of the line, **en** inserts (or overstrikes) a newline character, and **tl** (to left) moves the cursor to the beginning of the next line.

By default, <F1> invokes the **tr;en;tl** command sequence.

Inserting an End-of-File Mark

To insert an end-of-file mark (EOF) in a pad, type CTRL/D or specify the **eef** (edit end-of-file) command. If the line containing the cursor is empty, the DM inserts the end-of-file mark on that line. Otherwise, the DM inserts the end-of-file mark following the current line.

It is a common (although not universal) convention for programs to terminate execution and return to the process that called them when they receive an end-of-file mark on their standard input stream.

Whether or not the DM also deletes the transcript window depends on the setting of its auto-close mode. If auto-close mode is disabled (the default setting), then you must manually delete any windows associated with the closed transcript pad by using the DM command line **wc -q**, or by pressing <ABORT> .

Chapter 5 describes the **wc -q** command line. See the **wc** (window close) command description in the *Domain Display Manager Command Reference* for more information about auto-close mode.

Deleting Text

The commands listed in Table 6-3 delete characters, words, or lines of text. To delete a larger block of text, refer to the "Cutting Text" section.

Table 6-3. Commands for Deleting Text

Task	DM Command	Predefined Key
Delete character at cursor	**ed**	<CHAR DEL> (L3)
Delete character before cursor	**ee**	<BACK SPACE> or <DELETE>
Delete "word" of text	**dr;/[~ a-z0-9! $_]/xd**	<F6>
Delete previous "word"	**dr;\[~ @@t@@n]\; [@@t@@n]\; ar;xd**	CTRL/W
Delete text from cursor to beginning of line	**dr;tl;xd**	CTRL/U
Delete from cursor to end of line	**es ";ee;dr;tr;xd;tl;tr**	<F7>
Delete entire line	**cms;tl;xd**	<LINE DEL> (L2)

Deleting Characters

To delete a character under the cursor, press <CHAR DEL> or specify the **ed** (edit) command. If the character under the cursor is a newline, **ed** joins the current line and the following line.

To delete a character left of the cursor, press <BACK SPACE> or <DELETE>, or specify the **ee** (edit erase) command. If the pad is in overstrike mode, the **ee** command replaces the character with a blank. <CHAR DEL>, <BACK SPACE>, and <DELETE> are repeat keys. You can repeat the operation by holding down the key.

Deleting Words

To delete a word of text at the current cursor position, press the predefined function key <F6>. In this case, a "word" consists of a string of characters that may include a tilde (~) in the first position of the word, and includes upper or lowercase letters, numbers, dollar signs ($), or underscores (_). The deletion stops at the next

space, punctuation mark, or special character (other than a dollar sign or underscore). Here are some examples of character strings that <F6> will delete: **$file, my_file3, ~report**. The <F6> function key invokes the command sequence

dr;/[~ a-z0-9!- $_]/xd

The DM writes the deleted word to its default paste buffer (a temporary file). You can reinsert the word elsewhere by moving the cursor to the desired location and either pressing <PASTE> or specifying the **xp** (paste) command. For more about **paste buffers** and the **xp** command, see "Copying, Cutting, and Pasting Text".

To delete text to the beginning of a line, use CTRL/U. To delete the previous word, use the CTRL/W control key sequence.

Deleting Lines

To delete text from the current cursor position to the end of the line (excluding the newline character), press the predefined function key <F7>. The <F7> key invokes this command sequence:

es ' ';ee;dr;tr;xd;tl;tr

The DM writes the deleted line to its default paste buffer. You can reinsert the line elsewhere by either pressing <PASTE> or specifying the **xp** command. For more information about paste buffers and the **xp** command, see "Copying, Cutting, and Pasting Text".

Defining a Range of Text

The editing commands that perform cut (delete), copy, and substitute functions operate on a range, or block, of text. Mark a range of text just as you would any other region in a pad (see "Defining Points and Regions" Chapter 4). However, you may not declare a range as an argument to an editing command. Use the **dr** (define region) command or <MARK> before using the editing command.

To use **dr** to define a range of text, define two points as follows:

[*point*] **dr;** [*point*]

The first *point* defines the beginning of the range, and the **dr** command marks it. The second *point* defines the end of the range. If you do not specify literal points, **dr** places the marks at the current cursor position.

An easy way to define a range of text is to indicate a position with the cursor and use <MARK>, which invokes the **dr** and **echo** (text echo) commands that in turn mark the first point and begin highlighting the text. Figure 6–2 illustrates how the DM highlights the text as you move the cursor to the end of the range.

To define a range of text using the cursor and <MARK>:

1. Move the cursor to the first point (the beginning of the range of text).
2. Press <MARK>.
3. Move the cursor to the second point (end of the range).
4. Specify the appropriate DM editing command.

Please note that the character under the cursor at the end of the range is not included within the range.

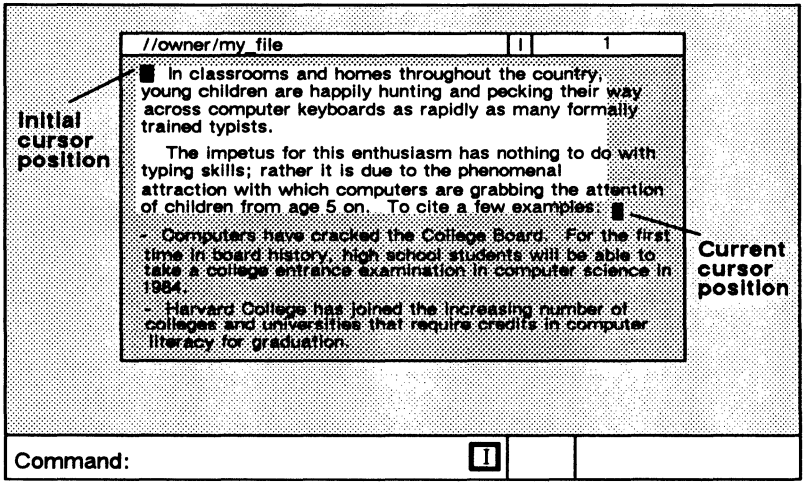


Figure 6–2. Defining a Range of Text with <MARK>

Copying, Cutting, and Pasting Text

The commands listed in Table 6–4 copy, cut, and paste a range of text. They allow you to move blocks of text from one place to another in a pad (or between pads).

Before specifying the commands that copy or cut text, use the **dr** (define region) command or <MARK> to define the range of text to be copied or cut (see the previous section). If you do not define a range, the DM copies or cuts the text from the current cursor position to the end of the line.

Table 6–4. Commands for Copying, Cutting, and Pasting Text

Task	DM Command	Predefined Key
Copy text to a paste buffer or file	**xc** [*name* | **-f** *file*] [**-r**]	<COPY> (L1A)
Cut (delete) text and write it to a paste buffer or file	**xd** [*name* | **-f** *file*] [**-r**]	<CUT> (L1AS)
Paste (write) text from a paste buffer or file into a pad	**xp** [*name* | **-f** *file*] [**-r**]	<PASTE> (L2A)

Using Paste Buffers

To perform copy, cut, and paste operations, the DM uses temporary files called paste buffers. Paste buffers hold text you've copied or cut so that you can paste it in elsewhere.

You can create up to 100 paste buffers, each containing different blocks of text. To create a paste buffer, you specify a name for the paste buffer as an argument to the commands that copy or cut text

(**xc** and **xd**). To insert the contents of a paste buffer you have created, specify the name of the paste buffer as an argument to the command that pastes text (**xp**). We describe the **xc**, **xd**, and **xp** commands in the following sections.

When you log off, the DM deletes all paste buffers you have created during the session. If you want to save the copied or cut text for use during another session, you can write it to a permanent file (see the **xc** and **xd** command descriptions in the following sections).

If you do not specify the name of a paste buffer or permanent file when you specify the commands that copy or cut text, the DM writes the text to its **default (unnamed) paste buffer**. The DM also uses this default paste buffer when you press the predefined function keys and control key sequences that copy, delete, and paste text.

> **NOTE:** In a paste buffer, the DM saves only the text copied or deleted during the last DM operation. Therefore, do not write anything else to the paste buffer until you have reinserted its contents. Otherwise, you will lose the text you're attempting to move.

Copying Text

To copy a defined range of text from any pad into a paste buffer or file, specify the **xc** (copy text) command in the following format:

xc [*name* | **-f** *file*] [**-r**]

The *name* argument specifies the name of a paste buffer that the DM creates to hold the copied text. The **-f** *file* option specifies the name of a permanent file for the text. For example, the following copies a defined range of text into a paste buffer named **copy_text**:

```
xc copy_text
```

As another example, the following command line copies a defined range of text into a permanent file named **copy_text**:

xc -f copy_text

If you supply the name of an existing paste buffer or file, **xc** overwrites its contents with the newly copied text. If you omit the name of a paste buffer or permanent file, **xc** writes the copied text to the default (unnamed) paste buffer.

The **-r** option instructs **xc** to copy a rectangular block of text that you have defined by marking the upper left and lower right corners of a text region. To define the region, use the cursor and the **dr** command or <MARK> to specify the left corner, then move the cursor to specify the right corner. If you specify a column (the left and right corners in the same column), **xc** copies all characters to the right of the column.

Figure 6–3 shows the two cursor positions used to mark the column. The dotted rectangle shows the block of text that the **xc -r** command line copies. (The dotted rectangle is only for the purpose of illustration; it does not appear on your display.)

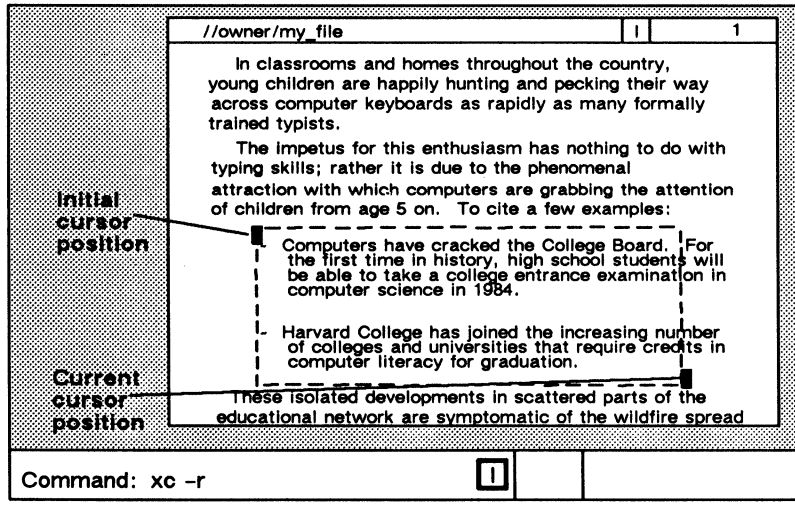


Figure 6–3. Copying Text with the xc -r Command

By default, <COPY> invokes the **xc** command using the default (unnamed) paste buffer. You must specify the **xc** command with the *name* argument or the **-f** *file* option if you want to copy text to a named paste buffer or permanent file.

Once you have copied a range of text, you can use the **xp** command to paste the text in elsewhere (see "Pasting Text").

Copying a Display Image

To copy a display image into a GMF, use the **xi** (copy image) command in the following format:

xi [**-f** *file*]

The **-f** *file* option specifies the name of the file you want to store the display image. If you omit the **-f** option, the system writes the image to the file **'node_data/paste_buffers/default.gmf**. Once you copy the image to a file, you can print the file using the **prf** command with the **-plot** option as follows:

prf my_file.gmf -plot

To use the **xi** command, mark the range of the display you want to copy. If you do not specify a range, **xi** copies the entire window in which the cursor is positioned. (On a color node, the **xi** command only copies the text plane, not the full color image.) Note that if you want to copy the whole screen, use the shell command **cpscr** (copy screen). Chapter 11 describes the **cpscr** command.

Cutting Text

When you cut text from a pad, the DM copies the text into a paste buffer or file and then deletes it from the pad. To cut a defined range of text, specify the **xd** (cut text) command in the following format:

xd [*name* | **-f** *file*] [**-r**]

The *name* argument specifies the name of a paste buffer that the DM creates to hold the deleted text. The **-f** *file* option specifies the name of a permanent file for the text. You can use this command only in pads created with <EDIT> or the **ce** (create edit) command.

If you supply the name of an existing paste buffer or file, **xd** overwrites its contents with the newly deleted text. If you omit the name of a paste buffer or permanent file, **xd** writes the deleted text to the default (unnamed) paste buffer.

The **-r** option instructs **xd** to delete a rectangular block of text that you have defined by marking the upper left and lower right corners of a text region. To define the region, use the cursor and the **dr** command or <MARK> to specify the left corner, then move the cursor to specify the right corner. If you specify a column (the left and right corners in the same column), **xd** deletes all characters to the right of the column.

By default, <CUT> invokes the **xd** command using the default (unnamed) paste buffer. You must specify the **xd** command with the *name* argument or the **-f** *file* option to write deleted text to a named paste buffer or permanent file, respectively.

Once you have cut a range of text, you can use the **xp** command (described in the next section) to paste the text in elsewhere.

Pasting Text

To insert the contents of a paste buffer or file into a pad at the current cursor position, specify the **xp** (paste) command in the following format:

xp [*name* | **-f** *file*] [**-r**]

The *name* argument specifies the name of an existing paste buffer that contains the text you want to insert. The **-f** *file* option specifies the name of an existing file that contains the text you want to insert. If you do not specify the name of a paste buffer or permanent file, **xp** inserts the contents of the default (unnamed) paste buffer.

You can use this command only in pads created with <EDIT> or the **ce** (create edit) command.

The **-r** option instructs **xp** to insert a rectangular block of text that you have copied or deleted using the **xc** or **xd** command and the **-r** option. The **xp** command uses the current cursor position as the origin (upper left corner) of the block.

By default, <PASTE> invokes the **xp** command using the contents of the default (unnamed) paste buffer. You must specify the **xp** command with the *name* argument or the **-f** *file* option to insert the contents of a named paste buffer or permanent file, respectively.

Using Regular Expressions

The DM search and substitute operations (described in the next several sections) allow you to use special notation, called regular expressions, to specify patterns for search and substitute text strings. You also use UNIX regular expressions with the shell commands **ed** (edit), **sed** (stream editor), and **grep** (pattern search). Note, however, that the UNIX regular expressions used by these commands are very different from the DM regular expressions discussed in this chapter. See the *BSD Command Reference* for descriptions of these commands.

Regular expressions permit you to concisely describe text patterns without necessarily knowing their exact contents or format. You can create expressions to describe patterns in particular positions on a line, patterns that always contain certain characters and at times may include others, or patterns that match text of indefinite length.

Following is a a list of the characters used to construct regular expressions and a brief description of their functions. Remember that these special characters apply *only* to regular expression operations. Some of these characters also have meanings (often radically different) in shell commands and other software products. If you want to use a regular expression as a part of one of those shell commands or products, be sure to enclose the expression in quotation marks so that it will not be misinterpreted.

ASCII Characters

Any standard ASCII character (except those listed in this section) matches one and only one occurrence of that character. By default, the case of the characters is insignificant. Use the **sc** (set case) command to control case significance (see the "Setting Case Comparison" section). The following examples are all valid expressions:

SAM
fred12
Joe(a&b)

Beginning of Line (%)

A percent sign (%) at the beginning of a regular expression matches the empty string at the beginning of a line. If a % is not the first character in the expression, it simply matches the percent character. Use this special feature to mark the beginning of a line in a regular expression. For example:

%Print matches the string in line **a** but not line **b** because, in line **b**, **Print** is not at the beginning of the line.

(a) Print this file
(b) This Print file

End of Line ($)

A dollar sign ($) at the end of a regular expression matches the end-of-line character (null) at the end of a line. If $ is not the last character in the expression, it simply matches the dollar sign character. Use this special feature to mark the end of a line in a regular expression. For example:

file$ matches the string in line **a**, but not line **b** because, in line **b**, **file** is not followed by an end-of-line marker.

(a) Print this file
(b) This file is permanent

Single Character Wildcard (?)

A question mark (?) matches any single character except a newline character. The only exception to this is when the question mark appears inside a character class (see "Strings and Character Classes"), in which case it represents the question mark character itself. For example:

?OLD??? matches the strings in lines **a** and **b**, but not line **c** because, in line **c** the letters "OLD" are alone on the line:

(a) **HOLDING**
(b) **FOLDERS**
(c) **OLD**

Expression Wildcard (*)

An asterisk (*) following a regular expression matches zero or more occurrences of that expression. The only exception to this is when the * appears inside a character class (see the "Strings and Character Classes" section), in which case it represents the asterisk character itself. Matching zero or more occurrences of some pattern is called a closure. An expression used in a closure will never match a newline character. Here are some examples:

a*b matches the strings **b**, **ab**, **aab**, etc.

%a?*b matches any string that begins with **a** and ends with **b**, and that is also the first string in the line. Any number of other characters can come between **a** and **b**.

Strings and Character Classes

A string of characters enclosed in square brackets, [*string*], is a **character class**. This pattern matches any one character in the string but no others. Note that the other regular expression characters % $? * lose their special meaning inside square brackets, and simply represent themselves. For example:

[sam] matches the single character **s**, **a**, or **m**. (If you want to match the word **sam**, omit the square brackets.)

A string enclosed in square brackets whose first character is a tilde [~string] matches any single character that does not appear in the string. If a tilde (~) is not the first character in the string, it simply matches the tilde character itself. For example:

[~sam] matches any single character except **s**, **a**, or **m**.

Within a character class, you can specify any one of a range of letters or digits by indicating the beginning and ending characters separated by a hyphen (-). For example:

[A-Z] matches any single uppercase letter in the range **A** through **Z**.

[a-z] matches any single lowercase letter in the range **a** through **z**.

[0-9] matches any single digit in the range **0** through **9**.

The range can be a subset of the letters or digits. However, the first and last characters in the range must be of the same type: uppercase letter, lowercase letter, or digit. For example, **[a-n]** and **[3-8]** are valid expressions. **[A-9]** is invalid.

Note that a hyphen (-) has a special meaning inside square brackets. If you want to include the literal hyphen character in the class, it must be either the first or last character in the class (so that it does not appear to separate two range-marking characters), or you can precede the hyphen with the escape character @ (see the @ description below).

The right bracket ( ]) also has special meaning inside a character class; it closes the class descriptor list. If you want to include the right bracket in the class, precede it with the escape character @ (see the @ description below). For example:

[a-d] matches any single occurrence of **a**, **b**, **c**, or **d**.

%[A-Z] matches any uppercase letter that is also the first character on the line.

5-[1-9][0-9]* matches any of the page numbers in this chapter.

[0A-Z] matches any string containing a zero or an uppercase letter.

[-a-z0-9] matches any uppercase letter or punctuation mark (i.e., no lowercase letter or digit).

Escape (@)

The at sign (@) is an escape character. Characters preceded by the @ character have special meaning in regular expressions, as indicated in the following list:

@n matches a newline character.

@t matches a tab character. In a regular expression, **@t** matches only tab characters that have been inserted with **@t**.

@f matches a form feed character.

In addition, you can use the escape character inside a character class to specify literal occurrences of a hyphen (-) or a right bracket (]). You may also use the @ character to specify a literal occurrence of the other special characters used in regular expressions: % $? * @. For example:

[A-Z@-@]] matches any uppercase letter, a hyphen, or a right bracket.

@?@* matches a question mark followed by an asterisk, rather than zero or more occurrences of any character (?*).

Text Pattern Matching with {expr}

You can "tag" parts of a regular expression to help rearrange pieces of a matched string. The DM remembers a text pattern surrounded by braces {*expr*} so that you can refer to it with @*n*, where *n* is a single digit referring to the string remembered by the *n*th pair of braces, e.g.,

s/{???}{?*}/@2@1/

The **s** command is the DM command for substituting strings of text (see "Substituting All Occurrences of a String"). This example of the **s** command moves a 3-character sequence from the beginning of a line to the end of the line. **???** matches the first three characters of the line, and **?*** matches the rest of the line. The **@2** expression refers to the string **?*** inside the second pair of braces, and **@1** refers to the string **???** inside the first pair of braces. For example:

```
so/{?}{?}/@2@1/
```

The **so** command is also a DM command for substituting strings of text, but it only substitutes the first occurrence of the first pattern on a line (see "Substituting the First Occurrence of a String"). This example of the **so** command transposes two characters beginning with the one under the cursor. This can be a handy key definition if you often type **ie** for **ei**, etc.

Searching for Text

The search operations shown in Table 6-5 locate strings of characters in a pad. You describe the string pattern using regular expressions (see the previous section).

Table 6–5. Commands for Searching for Text

Task	DM Command	Predefined Key
Search forward for string	*/string/*	None
Search backward for string	*\string*	None
Repeat last forward search	//	CTRL/N
Repeat last backward search	\\	CTRL/P
Cancel search or any action involving the echo command	**abrt**	CTRL/X
Set case comparison for search	**sc [-on] [-off]**	None

To search forward from the current cursor position, enclose the regular expression in slashes as follows:

/string/

To search backward from the current cursor position, enclose the regular expression in backslashes as follows:

\string

A search operation moves the cursor to the first character in the pattern specified by *string*. If necessary, the pad moves under the window to display the matching string. If the search fails, the cursor position does not change, and the DM displays the message "No match" in its output window.

Searches do not wrap around the end or beginning of the file. Therefore, to search an entire pad, position the cursor at the beginning of the pad.

By default, searches are not case-sensitive. This means, for example, that **/mary/** will locate **mary**, **MARY**, and even **maRy**. To perform a case-sensitive search, use the **sc** (set case) command.

A search is not syntactically a command; it's a positioning operation. One way to specify a point in a pad is by matching a regular expression. This means that the search operation is really a positioning action followed by a null command. Consequently, you should not think of search operations as operating on a text range, but rather searching from the initial cursor position to the end (or beginning) of the file in order to properly position the cursor.

If the DM scans more than 100 lines in a search operation, it displays a "Searching for /string/ ..." message in its output window. Then it polls for keystrokes every 10 lines it processes. At this point, you may:

- Wait for the DM to complete the operation.
- Cancel the search by typing CTRL/X, or by pressing a key that has been defined to invoke the **abrt** (abort) or **sq** (search quit) command (see "Cancelling a Search Operation").
- Use the keyboard; it works as it normally does. You can type into any pad except the one being searched. You can specify any DM command except another search or substitute command. The DM executes these commands when it completes the search. You can type input to another shell or program (if it was previously waiting for input). The process executes these commands when the DM finishes the search.

Repeating a Search Operation

To repeat the last search forward, specify the // command or type the CTRL/N ("next occurrence") sequence.

To repeat the last search backward, specify the \\ command or type the CTRL/P ("previous occurrence") sequence.

The DM saves the most recent search instruction, so you may repeat it even if you have specified other (non-searching) commands since then.

Canceling a Search Operation

To cancel the current search operation, type CTRL/X. The CTRL/X sequence invokes the **abrt** (abort) command. Since you cannot type DM commands for the pad being searched, you must use CTRL/X or define a key to invoke the **abrt** command (see "Defining Keys" in Chapter 4).

The DM command **sq** (search quit) also cancels a search operation. As with the **abrt** command, you must define a key to invoke **sq** during a search.

When you type CTRL/X or press a key defined to invoke **abrt** or **sq**, the DM displays the message "Search aborted" in its output window.

Setting Case Comparison

As we said earlier, a search operation is not case sensitive by default. In a case-insensitive search, upper- and lowercase letters are equivalent. In a case-sensitive search, the characters must match in case (that is, **/mary/** will not locate **/MARY/**).

To set case comparison for a search, specify the **sc** (set case) command in the following format:

sc [-on | -off]

The **-on** option specifies a case-sensitive search, and the **-off** option specifies a case-insensitive search. The **sc** command without options toggles the current case comparison setting.

Substituting Text

The commands shown in Table 6-6 allow you to search a pad or part of a pad for a text string, and to replace the string with a new string.

Before specifying a substitute command, use the **dr** (define region) command or <MARK> to define the range of text in which you want the substitution to occur (see the "Defining a Range of Text" section earlier in this chapter). If you do not define a range, the substitution occurs from the current cursor position to the end of the line.

Unlike searches, which ignore case unless told otherwise, all substitutions are case-sensitive. You cannot make a substitution case-insensitive.

Table 6-6. Commands for Substituting Text

Task	DM Command	Predefined Key
Substitute *string2* for all occurences of *string1* in a defined range	**s**/*string1*/*string2*	None
Substitute *string2* for the first occurence of *string1* in each line of a defined range	**so**/*string1*/*string2*	None
Change case of each letter in a defined range	**case [-s] [-u] [-l]**	None

If the DM scans more than 100 lines while processing a substitute command, it displays a "Substitute in progress..." message in its output window. Then it polls for keystrokes every 10 lines it processes. At this point, you may:

- Wait for the DM to complete the substitute operation.
- Type into any pad except the one where the substitution is occurring. You can specify any DM command except another search or substitute command. The DM executes these commands when it completes the substitution.

Substituting All Occurrences of a String

To replace all occurrences of a text string with a new text string, specify the **s** (substitute) command in the following format:

s[[/[*string1*]]/*string2*/]

The *string1* argument specifies the string to be replaced. Use a regular expression to describe *string1*. If you supply the first delimiter (/) but omit *string1* (that is, **s**//*string2*/), *string1* defaults to the string used in the last search operation. If you also omit the delimiter (that is, **s**/*string2*/), then *string1* defaults to the string used in the last substitute operation.

The *string2* argument specifies a literal replacement string (not a regular expression). If you supply *string1*, then *string2* is required.

You can use an ampersand (&) to instruct the **s** command to use *string1* as part of *string2*. For example:

s/Tom/& Smith/

This command replaces all occurrences of **Tom** with **Tom Smith** over the defined range of text.

The **s** command does not move the cursor or the pad, but does update the pad when the substitution is complete.

Substituting the First Occurrence of a String

The **so** (substitute once) command is like the **s** (substitute) command except that **so** replaces only the first occurrence of a string in each line of a defined range of text. Specify the **so** command in the following format:

so[[/[*string1*]]/*string2*/]

The *string1* argument specifies the string to be replaced. Use a regular expression to describe *string1*. If you supply the first delimiter (/) but omit *string1* (that is, **so**//*string2*/), *string1* defaults to the string used in the last search operation. If you also omit the delimiter (that is, **so**/*string2*/), then *string1* defaults to the string used in the last substitute operation.

The *string2* argument specifies a literal replacement string (not a regular expression). If you supply *string1*, then *string2* is required.

You can use an ampersand (&) to instruct the **so** command to use *string1* as part of *string2*. For example:

so/Tom/& Smith/

This command replaces the first occurrence of **Tom** with **Tom Smith** in each line of the defined range of text.

The **so** command does not move the cursor or the pad, but does update the pad when the substitution is complete.

Changing the Case of Letters

To change the case of letters in a defined range of text, specify the **case** command in the following format:

case [-s] [-u] [-l]

The **-s** option swaps all uppercase letters for lowercase and all lowercase letters for uppercase. The **-u** option changes all letters in the defined range to uppercase, and **-l** changes all letters to lowercase. The **case** command without options swaps all uppercase letters for lowercase and all lowercase letters for uppercase.

Undoing Previous Commands

To undo the most recent DM command you entered, use the **undo** command. You can also undo the previous command by pressing <UNDO>.

NOTE: The **undo** command only applies to DM operations, not shell commands.

The **undo** command works by compiling a history of DM operations in input and edit pads in reverse chronological order. It reverses the effect of the most recent DM command you specified. Successive **undo** commands reverse DM commands further back in history.

To compile its history of activities, the DM uses undo buffers (one per edit pad and one per input pad). The undo buffers are circular lists that, when full, eliminate the oldest entries to make room for new ones.

The DM groups entries together in sets. For example, an **s** (substitute) command may change five lines. While the DM considers this to be five entries, the five entries are grouped into a single set so that one **undo** will change all five lines back to their original state. When a buffer becomes full, the DM erases the oldest set of entries. This means that **undo** will never partially undo an operation; it will either completely undo the operation or do nothing.

An undo buffer for an edit pad can hold up to 1024 entries. An undo buffer for an input pad can hold up to 128 entries.

Updating an Edit File

To update a file that you are currently editing, specify the **pw** (pad write) command. This command is valid for edit pads only. It requires no arguments or options.

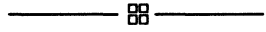
The first time you specify **pw** during an editing session, the DM writes the contents of the edit pad to the file that is being edited, without closing the edit pad. The DM writes the previous contents of the file to a file with the same name and the added suffix *.bak*. Subsequent **pw** or **wc** (window close) commands rewrite the new file and leave the .bak version unchanged. (For more about the **wc** command, see "Closing Pads and Windows" in Chapter 5.)

The **pw** command is similar to **wc** with two exceptions:

- The **pw** command leaves the edit pad open so that you can continue editing the file.
- The **pw** command writes the new version of the file even if other windows are viewing the edit pad.

If, for example, you want to try compiling a program you are editing, **pw** will prove to be useful. If you decide to make additional changes to the program, you can just go back to the edit pad and continue editing, since updates made by **pw** leave the edit pad open and active.

You can also update an edit file by pressing <SAVE> (see the "Closing Pads and Windows" section in Chapter 5).



Chapter 7

Introduction to Shell Usage

The BSD enironment supports several types of shells, including the Bourne shell, C shell, and Korn shell. Although each shell provides for I/O redirection, pipes, shell scripts, and wildcards, the implementation of these features frequently varies. This chapter highlights the important differences between shells, alerting you to shell characteristics that, while similar on the surface, may produce somewhat different results. Chapters 8, 9, and 10 provide in-depth information about the C, Bourne, and Korn shells, respectively.

Opening a Default UNIX Shell

The Bourne shell is the default UNIX shell in the BSD environment. If your system administrator has not specified a log-in shell for your account, you will see a Bourne shell when you log in. However, since the C shell is the preferred UNIX shell in the BSD environment, you may have a C shell appear instead. Use **chsh** (change shell) to change your log-in shell (see Chapter 3 for further information on this procedure).

You may arrange to have the DM (Display Manager) open a UNIX shell whenever any user logs in to the node, or only when you log in to the node. If you want every user to get a UNIX shell when they log in to a node, specify one of the UNIX environments (BSD or SysV) in the **/etc/environ** file. The **/etc/environ** file establishes the

log–in shell and default SYSTYPE for the node. Chapter 3 describes, in detail, how the system determines which shell to run when you log in.

If you would like to get a UNIX shell only when you log in to the node, specify one of the UNIX environments in the file **~/.environ** and set the desired shell using the **chsh** (change shell) command. The file **~/.environ** has the same format as the **/etc/environ** file (see Chapter 3 for further details).

The default BSD Bourne and Korn shell prompt is a dollar sign ($) followed by a space. The default C shell prompt is a percent sign (%) followed by a space. Any of these prompts can be changed from within the shell.

Opening Additional UNIX Shells

In addition to the shells created at login, you may need to create (and remove) other shells while you are logged in. If you press (shifted) <SHELL>, your log–in shell will be run in a new pad.

> **NOTE:** You can change your default shell by using the **chsh** (change shell) command. See Chapter 3 for further information.

As an alternative to using <SHELL>, you can simply tell the DM to create a process and run a shell in it. To create a process that runs a Bourne shell, press <CMD> and enter the DM command

```
Command: cp /bin/sh
```

To create a process that runs a C shell, press <CMD> and enter the DM command

```
Command: cp /bin/csh
```

The Display Manager creates the specified shell process in a window with a transcript pad and input pad. The SYSTYPE set at login determines which **/bin** is used. You may also specify **/sys5.3/bin** to force creation of a shell with the given SYSTYPE of **sys5.3**.

Shell Start-Up Files

When you log in, the DM runs /sys/**dm/login_sh**, which executes one of the UNIX shells as a log-in shell. This shell looks for a command file in your home directory. If the file exists, the shell executes it to set up its initial environment. Each of the three UNIX shells has its own command file that it executes when you log in.

You can create additional command files that are executed whenever you start a new shell process. For example, the C shell (that is, **/bin/csh**) looks for two files when you log in: a file named **.cshrc** and then a file named **.login**. It executes the **.login** file only when you log in. It executes the **.cshrc** file whenever you create a new shell.

The Bourne shell (**/bin/sh**) and Korn shell (**/bin/ksh**) look for a file named **.profile** when you log in. The environment variable **ENV** determines whether the Bourne and Korn shell run a start-up file for every new shell. If you define **ENV**, the Bourne and Korn shells take its value as the name of a start-up script to execute. The **.profile** file may assign a value to **ENV** and that script will be run after **.profile** has finished.

Table 7-1 lists the names of the different shell start-up files. By convention, the script name **.kshrc** is used for the Korn shell and **.shrc** for the Bourne shell.

Table 7-1. Shell Start-Up Files

Shell	At Login	New Shell
/bin/csh	~/.login	~/.cshrc
/bin/ksh	~/.profile	$ENV
/bin/sh	~/.profile	$ENV

We have added the following option to the UNIX shells

-D*variable* = *value*

This option allows you to define a variable, then use the value of that variable to do special processing at shell startup. For example, you can use this option in your <SHELL> key definition to run the ~/.shrc script whenever you create a Bourne shell:

```
kd l5s cp /bin/sh -DENV=~/.shrc ke
```

This example sets the value of the **ENV** variable to ~/.shrc, then passes that value to the shell's environment. The Bourne shell then uses ~/.shrc as a start-up script, and attempts to execute it. We assume you have already created a file named ~/.shrc, which contains shell commands. Refer to chapters 8, 9, and 10 for information about creating shell scripts for the C, Bourne, and Korn shells.

It may be useful to have more than one script to which ENV may point. For new pads, you may want **ENV=~/.shrc.pad** specified. This script is the first thing to be run in the new pad and may print text to the pad. At the end of the script, you may want to either unset ENV or set it to another pathname. Since each time a new subshell is created (to run a shell script, for example) the ENV is checked, you may want something other than your new pad script to be run.

Using a Terminal

To access Domain/OS via a **tty** device (terminal), on either a hard-wired or phone line connection to an Apollo node's SIO (Serial Input Output) line, you use a different procedure for creating a UNIX shell.

As described in Chapter 3, the **init** program reads the file **/etc/ttys** as part of system startup. The **/etc/ttys** file contains a list of "terminal lines" (including **tty** devices) and the program to run for each "terminal line". The **/etc/ttys** file usually contains a line invoking the **/etc/getty** program to initialize tty devices. The **/etc/getty** program determines the terminal characteristics, and lets users log in.

When you log in, **/etc/getty** reads your username and executes the **/bin/login** command.

When you log in on a tty line, the initial shell that appears is determined by the log-in shell field of your account registry. If your registry account doesn't specify a shell, **login** executes the default shell for the node which you are logging in to. The values for SYSTYPE and shell are determined in exactly the same manner as for the DM.

> **NOTE:** Be aware that Domain/OS serial line architecture sometimes causes unpredictable results if you attempt to use a terminal that doesn't expect eight-bit characters.

When the **login** program is used to start a UNIX shell on a tty line, it binds various functions (signals) to control characters as noted in Table 7-2. You can change these characters using the **tset** (terminal-dependent initialization) or **stty** (set terminal options) commands. For more information about these commands, see the *BSD Command Reference*.

Table 7-2. Control Characters Defined in a UNIX Shell

Function	Control Character
erase	<DELETE>
kill	CTRL/U
interrupt	CTRL/C
suspend	CTRL/Z
eof	CTRL/D
quit	CTRL/\

The last close of the tty line causes the node's serial I/O hardware to drop the DTR (Data Terminal Ready) signal. This causes most modems to hang up the phone. For more concerning tty line characteristics, see the **tset** command in the *BSD Command Reference*.

Search Path

Chapter 2 describes environment variables and multiple version support. The search path for shells is modified by the SYSTYPE environment variable. Each shell has a built-in command search path. The exact path depends on the shell. UNIX shells look for commands in the following places, in this specific order:

1. current directory
2. **/usr/ucb**
3. **/bin**
4. **/usr/bin**
5. **/usr/apollo/bin**

You can change the default search path in any of our UNIX shells by setting the shell variable called **PATH** (or **path**, for C shell users). See Chapters 8, 9, and 10 for further detail.

Shell Program Execution

A shell script is a text file that contains a series of UNIX commands. You can specify which shell (C, Bourne, or Korn) is to interpret and execute a shell program by starting the first line of each shell script with the character sequence **#!** followed by the pathname of the desired shell, as shown here:

#!/bin/sh	Specifies a Bourne shell script. In this case, the Bourne shell used is the one found in **/$SYSTYPE/bin.** If you need to be more specific, you may say:
	#! /sys5.3/bin/sh Specifies a **sys5.3** Bourne shell.
#!/bin/csh	Specifies a C shell script.
#!/bin/ksh	Specifies a Korn shell script.

The following shows how this line is used in a Bourne shell script:

```
#!/bin/sh
#
for i do
          case . . .
               . . .
               . . .
               . . .
               . . .
          esac
done
```

The shell interpreter directive **#!** must appear as the first line of the file in order to be interpreted correctly (remember that this information is case–sensitive), and it must comprise the first two characters of the line. Any amount of white space may appear between the exclamation point and shell pathname.

The C shell invokes **/bin/sh** (the Bourne shell) to interpret shell scripts when there is no explicit **#!** shell designation. In other shells, a script with no shell specification line is interpreted (with unpredictable results) by the shell in which it was invoked.

Wildcards

Every shell has its own metacharacters (wildcards). Chapters 8, 9, and 10 detail the wildcard–handling mechanisms of the C, Bourne, and Korn shells. Differences among the various UNIX shells can be important considerations.

All UNIX shells perform some type of wildcard expansion, and UNIX commands expect a command line that has already been expanded by the shell.

> **NOTE:** If you are using both the BSD and the Aegis environments, be aware that Aegis shell commands perform wildcard expansion with rules that differ from those used by UNIX shells. For this reason, use of commands found in **/com** is not recommended from UNIX shells.

Chapter 8

Using the C Shell

The primary purpose of any shell is to translate command lines typed at a terminal into useful work, something the shell usually accomplishes by invoking another program. The C shell (**/bin/csh**) is one of several shells available to users of the BSD environment.

This chapter introduces the more commonly-used features of the C shell. We recommend that you try all the examples shown, to develop a variety of experiences with the C shell. The **csh** (C shell) documentation in the *BSD Command Reference* provides a full description of all features of this shell. Appendix E also provides a summary of valid C shell metacharacters.

Starting the Shell

To start a C shell on an Apollo node, log in and type the DM command

```
Command: cp /bin/csh
```

In this command line, **/bin** resolves to **/$(SYSTYPE)/bin**.

The DM opens a window and runs the C shell in it. With the **csh** command, you may supply the coordinates where the DM will locate the upper left and lower right corners of the window. You may even give the process a name, as in this line:

```
Command: (0,200)dr; (540,600)cp /bin/csh -n c_shell
```

This command line opens up a small window near the left side of the screen and displays the name **c_shell** in the window legend.

The Basic Notion of Commands

A shell acts primarily as a medium through which you invoke other programs. While the shell has a set of built-in functions that it performs directly, most commands to the shell cause execution of programs that reside elsewhere (are not part of the shell).

A command consists of a word or words that the shell interprets as a command name followed by optional arguments. Thus, the command

```
% mail kate
```

consists of a command name (**mail**), followed by an argument (**kate**). The shell looks through the directories in its search path for an executable file named **mail**.

The rest of the words on the command line are assumed to be arguments and are passed to the command when it is executed. In this case, we specified the argument **kate** which **mail** interprets as the name of a user to whom mail is to be sent. In normal usage, you might invoke **mail** as follows:

```
% mail kate
Is there a meeting today?  And is it at 1:00?
bob
*** EOF ***
EOT
%
```

Here we typed a message to send to **kate** and ended this message with a CTRL/D, which sent an end-of-file (EOF) to the **mail** program.

The **mail** program, in turn, echoed EOT (end-of-transmission), transmitted the message to **kate**, and exited. The shell, noticing that **mail** was finished, prompted for input by displaying a percent sign (%), which indicated its readiness for further orders.

This is the essential pattern of all interactions with BSD software via the C shell. You type a complete command, and the shell executes it. When command execution completes, the shell prompts for a new command. If you run, for example, the **vi** (visual display editor) editor for an hour, the shell waits for you to finish editing, and then prompts you for further orders.

Flag Arguments

While many arguments to commands specify objects such as filenames, some arguments invoke optional capabilities of the command. By convention, such arguments begin with a dash (-). Thus, the following command produces a list of the files in the current working directory:

```
% ls
```

The **ls** command has many options, including **-s**, the size option. If you include **-s** on a **ls** (list directory) command line,

```
% ls -s
```

ls lists the size of each file in blocks of 1024 bytes. Refer to the *BSD Command Reference* for available options for each command.

Output to Files

Commands that normally read input or write output on the screen can optionally be told to get their input from a file or to send their output to a file. Suppose you wish to save the current date in a file called **now**. This command

```
% date
```

prints the current date on the transcript pad of the shell into which **date** (print date) was typed, because the screen (transcript pad) is the default standard output, and **date** always prints the date on the standard output.

The shell lets you redirect the standard output of a command through a notation using the greater-than (>) metacharacter and the name of the file where output is to be placed.

Thus, the command

```
% date > now
```

runs the **date** (print the date) command and redirects the standard output to a file called **now** rather than to the default standard output (the screen). The current date and time are written to the file **now**. No output appears on the screen. It is important to know that **date** is unaware that its output is going to a file rather than to the screen. The shell performs this redirection before the command begins executing.

The file **now** need not have existed before the **date** command above was executed; the shell would have created the file (in the current working directory) if it did not exist.

NOTE: If you redirect standard output into an existing file, that file is overwritten unless the shell variable **noclobber** has been set. See the discussion of **noclobber** later in this chapter.

Input From Files Using Pipelines

The standard input of a command can be redirected so that it is taken from a file, instead of the keyboard (default standard input). This is often unnecessary, since most commands read from a file whose name is given as an argument. You could use this

```
% sort < data
```

to run to run the **sort** (sort or merge files) command with standard input, where the command normally reads its input, from the file **data**. But, it is easier and just as legal to type this

```
% sort data
```

letting the **sort** command open the file **data** and sort it.

NOTE: If you merely type the **sort** command without an argument, lines are sorted from its standard input, the keyboard. Since you are not redirecting the standard input, the program sorts lines as you type them on the terminal, until you type a CTRL/D to indicate an end-of-file.

Another useful feature of the C shell is its ability to connect the standard output of one command to the standard input of another using a mechanism known as a pipeline. For instance, the following command line

```
% ls -s
```

normally produces a list of the files in the current directory and lists the size of each file in blocks of 1024 characters.

To help determine which of your files is largest, you may want to have the list sorted by size rather than by name. Although **ls** has no such option, you can pipe the output of **ls** (list directory) to the **sort** command and use some of **sort**'s options to get a list of files sorted in size order.

The **-n** option of **sort** specifies a numeric sort rather than an alphabetic sort. Thus,

```
% ls -s | sort -n
```

tells the C shell to run the **ls** command with the **-s** option, and then pipe the resulting output to the **sort** command run with the **-n** (numeric sort) option. The output of this combination of commands is a list of files sorted by size, with the smallest file first. You could then use the **-r** reverse sort option and the **head** (give first few lines) command combined with the previous command, as follows:

```
% ls -s | sort -nr | head -5
```

This sequence takes a list of files sorted alphabetically, each with the size in blocks, and pipes this list to the standard input of **sort**. The **sort** command, in turn, sorts the list numerically in reverse order (largest first). The sorted list is piped to the command **head** which then displays the first five lines of the list, giving you names and sizes of the five largest files in the current directory.

Commands separated by pipe (|) characters are connected together by the shell. The standard output of the command to the left of the pipe is connected to the standard input of the command to the right of the pipe. The leftmost command in a pipeline normally takes its standard input from the keyboard. The rightmost places its standard output on the screen.

Metacharacters in The C Shell

The C shell uses a number of characters to perform special functions. In general, many characters that are neither letters nor digits have special meaning to the shell. Since these special characters may also be used literally, the shell provides a means of quoting that lets you strip these metacharacters of any special meaning.

Metacharacters normally have effect only when the shell is reading input. You needn't worry about placing shell metacharacters in a letter you are sending via **mail**, or when supplying text or data to some other program. Note that the shell is only reading input when it is displaying its prompt.

Filenames

Many commands need the names of files as arguments. Domain/OS pathnames consist of a number of components separated from each other by the slash (/). Each component except the last names a directory in which the next component resides, in effect specifying the path of directories to follow to reach the file.

Thus, the pathname **/usr/apollo/bin/systype** specifies a file in the directory **/usr/apollo/bin**. Within this directory the file named is **systype**, a program that examines binaries for SYSTYPE flags. A pathname that begins with a slash is said to be an absolute pathname, since it is specified from the absolute top of the node's directory hierarchy.

> **NOTE:** A node's directory hierarchy begins one level below the network root, or "double slash" (//) directory, so a truly "absolute" pathname must always begin with two slashes followed by the name of the node's entry directory as in the following: **//ice/usr/apollo/bin/systype**.

When the shell sees a pathname that does not begin with a slash, it assumes that it should start looking in the current working directory. When you log in, the working directory is set to your home directory. From there, you can move to (and work in) other directories by using the **cd** command. Pathnames not beginning with a slash are said to be relative to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname contains no slashes, the shell assumes that the pathname is the name of a file contained in the current working directory. Absolute pathnames, by contrast, are unrelated to the working directory.

Most filenames consist of a number of alphanumeric characters and periods. While all characters except a slash (/) and null may appear in UNIX filenames, it is inconvenient to have most non-alphabetic characters in filenames, since many of them have special meaning to the shell. The period or dot (.), while not a C shell metacharacter, is often used to separate the extension of a filename from the base of the name. Thus

prog.c prog.o prog.errs prog.output

are four related files. Their names share a common base portion (that part of the name which is left when a trailing period and following characters that are not periods are stripped off). The file **prog.c** might be the source for a C program, the file **prog.o** the corresponding object file, the file **prog.errs** the errors resulting from a compilation of the program and the file **prog.output** the output of the program itself.

To refer to all four of these files in a command, use the notation:

prog.*

The shell expands **prog.*** into a list of names that begin with **prog.** before the command to which it is an argument is executed. The asterisk (*) here matches any sequence (including the empty sequence) of characters in a filename, except a leading dot (.). The names that match are alphabetically sorted and placed in the argument list of the command. Thus,

```
% echo prog.*
```

echoes the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we list them above. The **echo** (echo arguments) command receives four words as arguments, even though only one argument is supplied to the shell. The shell generates the four words by filename expansion of the one input word.

The C shell also expands other characters. The question mark (?) matches any single character in a filename, except a leading dot (.). Thus,

echo ? ?? ???

echoes a line of filenames; first those with 1-character names, then those with 2-character names, and finally those with 3-character names. The filenames of each length are sorted independently (i.e., the output to the screen is a list of 1-character filenames, followed by a list of 2-character filenames, followed by a list of 3-character filenames).

The shell also matches any single character from a sequence of characters delimited by brackets. Thus,

prog.[co]

matches both **prog.c** and **prog.o**. You can also place two characters around a dash (-) in this notation to denote a range. Thus, to **troff** five chapters of a book that exists in the files **chap.1**, **chap.2** and so on, type the command line

% troff chap.[1-5]

which would pass the names

chap.1 chap.2 chap.3 chap.4 chap.5

to **troff** for processing. The above notation is equivalent to

chap.[12345]

> **NOTE:** If a list of argument words to a command (an argument list) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing filenames, then the shell considers this to be an error and prints the diagnostic message "No match" and does not execute the command.

Files beginning with a period (.) are treated specially. This prevents accidental matching of the filenames "." and ".." in the working directory, where they have special meaning to the system. It also prevents matching of other files such as **.cshrc** which are not normally visible in a directory listing. (We discuss **.cshrc** in a later section.)

Another filename expansion mechanism gives access to the pathname of the *home directory* of other users. Normally, this notation consists of a tilde (~) followed by a user's login name. For instance, the word **~kate** maps to the absolute pathname of user **kate**'s home directory, as shown here:

```
% cd ~kate
% pwd
//ice/kate
```

A special case of this notation consists of a tilde alone, which expands to the pathname of your home directory. For example, the command

```
% ls -a ~
```

lists all the files in your home directory. Likewise, the command

```
% cp thatfile ~
```

expands to

```
% cp thatfile your_home_directory/thatfile
```

The shell also has a mechanism that uses left and right brace characters ({ }) for abbreviating a set of words that have common parts but can't be abbreviated by other mechanisms because they are not files (or are files that, while created by the program being invoked, do not exist yet). This mechanism is described in a later section.

Quotation

We have already described a number of the metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus, the command

```
% echo *
```

does not echo the asterisk (*). It either echoes a sorted list of all filenames in the current working directory, or prints the message "No match" if no files exist in the working directory.

The recommended mechanism for placing a character with special meaning to the shell in an argument word to a command is to enclose it in single quotes ('), as in the following example:

```
% echo '*'
```

One special character, the exclamation point (!), is used by the history mechanism of the shell and cannot be escaped by the normal means of placing it within single quotes. The exclamation point and the single quote should be preceded by a single backslash (\) to escape their special meaning. Thus,

```
% echo \'\!
```

prints

```
' !
```

These two mechanisms let you include any printing character in an argument to a shell command. They can be combined, as in

```
% echo \''*'
```

which prints '* since the first backslash escaped the first single quote and the asterisk was enclosed in single quotes.

Terminating Commands

When you are executing a command and the shell is waiting for it to complete, there are several ways you can force it to stop executing. For instance, if you type the following the system prints a list of all users on the system:

```
% cat /etc/passwd
```

This is likely to continue for several minutes unless you stop it. You can send an interrupt signal to the **cat** (catenate and print) command by typing CTRL/C.

Since **cat** doesn't try to avoid or to otherwise handle this signal, the interrupt terminates **cat**. The shell notices its termination and prompts you again. If you hit interrupt again, the shell repeats its prompt since it is designed to effectively ignore interrupt signals.

Many programs terminate when they get an end-of-file from their standard input. The **mail** program in an earlier example terminated when it received a CTRL/D (which generates an end-of-file) from the standard input. The C shell normally terminates when it receives an end-of-file. When this happens, the messages

```
% *** EOF ***
logout
*** Pad Closed ***
```

are left on the transcript pad and the window is closed. Since this means that typing CTRL/D one too many times can accidentally log you out of a window, the shell has a mechanism for preventing this. This **ignoreeof** option is discussed in the next section.

If a command has its standard input redirected to come from a file, it normally terminates when it reaches the end of this file. If you execute the following command line,

```
% mail kate < prepared.text
```

the **mail** command terminates when it sees the EOF at the end of the file **prepared.text** from which it is getting input. Another way to accomplish the same thing is to type

```
% cat prepared.text | mail kate
```

since the **cat** command then writes the text through the pipe to the standard input of the **mail** (send and receive mail) command. When the **cat** command completes, it terminates, closing down the pipeline, and the **mail** command receives an end-of-file from **cat** and terminates. You can also stop these commands by typing CTRL/C.

If you write or run programs that are not fully debugged, it may be necessary to stop them somewhat ungracefully. This can be done by typing CTRL/\, which sends a quit signal.

Commands running in the background ignore interrupt and quit signals. To kill them, use the **kill** (terminate process) command.

Starting, Exiting, and Modifying the C Shell

This section includes information on starting the C shell and arranging for it to set certain variables to convenient values every time you log in.

Opening a C Shell When You Log In

The Bourne shell is the default UNIX shell in the BSD environment. Chapter 3 describes how the system determines which shell to run when you log in. You may arrange to have the system open a C shell as your log-in shell by specifying **/bin/csh** in the shell field of your registry account, using the **chsh** (change shell) command.

When you log in to an Apollo node, the DM looks in several places for information about what windows to open and what processes to start. It normally opens a default shell, then looks for the file

your_home_directory/**user_data/startup_dm.***display_type*

The *display_type* argument matches the type of display in use (for example, 1280bw). If you include a command line such as the following:

```
(0,200)dr; (540,600) cp /bin/csh
```

in your **startup_dm** file, the DM automatically opens a C shell when you log in.

You may also define a key or function key to open a C shell. The following DM command defines the shifted L5 key (L5 is labeled <SHELL>) so that when you press SHIFT/ <SHELL>, a C shell is opened:

```
kd l5s cp /bin/csh ke
```

> **NOTE:** By default, <SHELL> starts up a new pad with your log-in shell, so if it is **csh**, you don't need to do the above.

Log-In and Log-Out Scripts

When you log in, the C shell sets the working directory to your home directory and begins reading commands from a file **.cshrc** in this directory. Every C shell reads from this file. In addition, you may create a file called **.login** in your home directory that the C shell reads (after it reads **.cshrc**) if it is started as a log-in shell. Neither of these files is required. If neither exists, the shell uses its own defaults. As an example of a **.cshrc** file, consider the following:

```
set history=10
set path = (. ~/bin /usr/ucb /bin /usr/bin /usr/apollo/bin)
set noclobber
set ignoreeof
set inprocess
alias cd 'cd \!* ; ls'
```

This file begins with a series of **set** commands that the shell interprets directly. These particular **set** commands establish the following conditions in the C shell:

- The shell maintains a "history list" of the last 10 commands.
- The shell searches for a command in the following places, in this order:

  1. . (current directory)
  2. *home_directory*/**bin**
  3. **/usr/ucb**
  4. **/bin**
  5. **/usr/bin**
  6. **/usr/apollo/bin**

- The variable **noclobber** is set, forcing the shell to notify you whenever you redirect output into a file that already exists.

  NOTE: You may override **noclobber** if it is set by using the >! syntax. For example, to overwrite the contents of a file named **now** with the current date, you can do so even if **noclobber** is set. Typing the command line **date >! now** does it. The >! combination is a special metasyntax indicating that clobbering the file is allowed. Note that the space between the exclamation point and the **now** is critical here, as **!now** is an invocation of the history mechanism, and has a totally different effect.

- The variable **ignoreeof** is set. The shell does not terminate (close the window or, if you are using a terminal, log you off) when it receives an end-of-file from standard input.

The next two commands are **alias** commands that, in effect, rename command sequences. Here, the command **cd** is aliased to change to the specified directory, then list its contents. And, since the variable **ignoreeof** is set, the string "lo" is defined as having the alias **logout**, allowing the closing up of the shell window with a minimum of typing.

Shell Variables

The shell maintains a number of variables. In the **.cshrc** file shown in the previous section, the variable **history** was set to a value of 10. In fact, each shell variable has as its value an array of zero or more strings. The **set** command assigns values to variables. **Set** has several forms, the most useful of which is

set *name=value*

Shell variables let you store values that can then be made available, via the substitution mechanism, to commands. The shell variables most commonly referenced are, however, those to which the shell itself refers. By changing the values of these variables, you can directly affect the behavior of the shell.

One of the most important variables is **path**. It contains a sequence of directory names where the shell searches for commands. If you execute the **set** command with no arguments, the shell displays the values of all variables currently set.

The shell examines each directory in the specified **path** and determines what commands are contained there. Except for the current directory, which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they are not necessarily found by the shell.

To use a command that has been added in this manner, specify the **rehash** command. This command causes the shell to recompute its internal table of command locations, so that it finds the newly added command. Since the shell has to look in the current directory for each command, placing **rehash** at the end of the path specification works equally well and reduces overhead.

Other useful built–in variables are **home**, which has your home directory, and **cwd**, which contains your current working directory. The **ignoreeof** variable is one of several variables only examined by the shell to see if it is set or unset (it can have a value, but its value is not examined by the shell). Thus, to set this variable, type:

```
set ignoreeof
```

To unset it, type the following:

```
unset ignoreeof
```

The variable **noclobber** is another such Boolean variable.

History

The shell can maintain a history list into which it places the words of previous commands. This history mechanism lets you reuse commands or words from them in forming new ones. Use this mechanism to repeat commands or to correct minor typing mistakes in them. The following example shows how the C shell's history mechanism is typically used.

```
% cat bug.c
main()
{
     printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4:newline in string or char constant
"bug.c", line 5:syntax error
% ex !$
ex bug.c
"bug.c"
4s/);/"&
printf("hello");
wq
"bug.c"
% !c
cc bug.c
% a.out
hello% !e
ex bug.c
ex bug.c
4s/lo/lo\\n
printf("hello\n");
wq
"bug.c"
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 kate eng 3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 kate eng 3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ^spp^ssp
num bug.c | ssp
1 main()
2 {
3 printf("hello\n");
4 }
% !! | prf
num bug.c | ssp | prf
```

This example shows a very simple C program with some bugs. To begin, we use **cat** to print the file **bug.c** onto the screen. Then, we attempt to run the C compiler, **cc**, referring to the file again as **!$**, which is an invocation of the **history** mechanism that means "use the last argument to the previous command." The exclamation point is the metacharacter that invokes the history mechanism and the dollar sign stands for the last (most recent) argument read by the shell. The shell echoes the command, as it would have been typed without using the **history** mechanism, and then executes it.

Since the compilation yielded error diagnostics, we invoke the line editor, **ex** to fix the bug. Then the file is recompiled, this time referring to the **cc** command simply as **!c**. The notation *!x* tells the shell to repeat the most recently submitted command that begins with character *x*. If specificity is necessary (for example, if other commands starting with *c* had been used recently), we can invoke the **history** mechanism by typing **!cc**. If further caution is needed, the form **!cc:p** prints the last command that started with **cc**, without executing it so you can see what it would run. This command is entered into the history list as the previous command.

After this recompilation, a run of the resulting **a.out** file reveals that a bug still exists, so we reinvoke the editor, and then the C compiler. This time, we add the **-o bug** switch to the **cc** command line, telling the compiler to place the resultant binary in the file **bug** rather than **a.out**. In general, the **history** mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then run the **size** command to see how large the object files were, and then an **ls -l** command with the same argument list, denoted by the argument list *\!**. Finally, we run the **bug** program to see that its output was indeed correct.

To make a numbered listing of the program we run the **num** program on the file **bug.c**. To eliminate multiple blank lines in the output, we run it through the filter **ssp**, but misspell it as **spp**. To correct this, we use a shell substitute, placing the old text and new text between circumflex (^) characters. This is similar to the substitute command in the editor. We then repeat the same command with **!!**, but send its output to the line printer.

NOTE: On Apollo nodes, <AGAIN> is often defined to copy all text between the cursor position and the next EOL into the "next input window." In fact, the DM's cut-and-paste facilities may be more effective than the **history** mechanism in certain situations.

You can repeat a command from the history list by other means. The **history** command prints out a number of previous commands accompanied by the numbers with which they can be referenced. You can also refer to a previous command by searching for a string that appeared in it. See **csh** in the *BSD Command Reference* for a complete description of these mechanisms.

Aliases

The shell has an alias mechanism that helps in transforming input commands. It can be used to simplify the commands you type, to supply default arguments to commands, or to do transformations on commands and their arguments. The alias mechanism is similar to a macro facility. Some of the features obtained by aliasing can also be obtained using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment or involve commands such as **cd** (change directory), which must be done in the current shell. For example, if you'd like the command **ls** (list directory) to always show sizes of files (i.e, do **-s**), use the following alias:

```
% alias ls ls -s
```

Or, you can create a "new" command called **dir** that does the same thing, by typing

```
% alias dir ls -s
```

Thus, the alias mechanism can be used to provide short names for commands, to supply default arguments, and to define new short commands in terms of other commands. You can also define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. For example, the alias for **cd** in our **.cshrc** example shown earlier in this chapter,

% alias cd 'cd \!* ;ls'

causes the shell to automatically do an **ls** after every **cd**. We enclose the entire alias definition in single quotes (') to prevent most substitutions from occurring and to prevent the semi-colon (;) from being recognized as a metacharacter. The exclamation point (!) here is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing **cd** command, without giving an error message if no arguments are supplied. The semi-colon is used to indicate that one command is to be done first, followed by the next. Similarly, the definition

% alias whois 'grep \!^ /etc/passwd'

defines a command which looks up its first argument in the password file.

> **NOTE:** The C shell reads the **.cshrc** file each time it is invoked. If you put many commands there, shells tend to start slowly. We recommend that you limit the number of aliases in this file. Ten aliases cause no perceived delay. Fifty aliases cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

More Redirection Using >> and >&

In addition to the standard output, commands also have a diagnostic output (or "error output") that is normally directed to the screen even when the standard output is redirected to a file or a pipe. If you need to redirect the diagnostic output to the same place as you redirect standard output (e.g., if you want to redirect the output of a long-running command into a file and need to have a record of any error diagnostics produced while the command was running), use the notation

command **>&** *file*

The >& here tells the shell to route both the diagnostic output and the standard output into *file*. Similarly, you can give the command

command |& prf

to route both standard and diagnostic output through the pipe to the **/usr/apollo/bin/prf** print spooler.

You can use this notation when **noclobber** is set and *file* already exists:

command >&! *file*

Finally, it is possible to use the following form to place output at the end of an existing file:

command >> *file*

If **noclobber** is set, an error results if *file* does not exist; otherwise, the shell creates *file* if it doesn't exist. A form such as the following one can be used if it's necessary to override **noclobber**'s error message:

command >>! *file*

Background, Foreground, and Suspended Jobs

When one or more commands are connected via pipes or as a sequence of commands separated by semicolons, the shell creates a single job consisting of all commands so connected. A single command without pipes or semicolons is, of course, the simplest job. Usually, every line typed to the shell creates a job.

If you type the ampersand (&) metacharacter at the end of a command line, the job generated by that command line is started as a background job. Thus, the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs "in the background" at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time.

Thus, the following command line runs the **du** (summarize disk usage) program, which reports on the disk usage of your working directory (as well as any directories below it), puts the output into the file **usage**, and returns immediately with a prompt for the next command without waiting for **du** to finish:

```
% du > usage &
```

The **du** program continues executing in the background until finished, and the shell continues accepting input from you. When a background job terminates, the shell types a message before the next prompt, telling you that the job is completed. In the following example, the **du** job finishes sometime during the execution of the **mail** command. Its completion is reported just before the prompt after the **mail** job is finished.

```
% du > usage &
[1] 503
% mail kate
How can I tell when a background job is finished?
bob
*** EOF ***
EOT
[1] - Done du > usage
%
```

If the job hadn't terminated normally, you might have gotten a message such as "Killed". To have terminations of background jobs reported at the time they occur (possibly interrupting the output of other foreground jobs), set the **notify** variable. If you had done this for the previous example, the "Done" message might have appeared in the middle of the message to **kate**. Background jobs are unaffected by any signals from the keyboard (e.g., stop, interrupt, quit).

Information about all running jobs is recorded in a table maintained by the C shell. In this table, the shell stores the names, arguments, and process numbers of all commands in the job. It also notes the working directory in which the job was started. Each job in the table is either running in the foreground with the shell waiting for it to terminate, running in the background, or suspended.

Only one job can be running in the foreground. Simultaneously, several jobs can be either running in the background or suspended. As each job is started, it is given a **job number**. This number is used in conjunction with the commands below to suspend or kill the job. The job number assigned to a job remains the same until the job terminates, at which time the job number is available for reuse.

When a job is started in the background, the shell displays the job's number, as well as the process numbers of all its (top level) commands. This job, for example, runs the **ls** program with the **-s** option, and pipes this output into the **sort** program with the **-n** option, which puts its output into the file **usage**:

```
% ls -s | sort -n > usage &
[2] 65 66
%
```

Since an ampersand appears at the end of the line, these two programs start together as a background job. After starting the job, the shell prints the job number (e.g., 2) in brackets followed by the job's process numbers, then prompts for a new command.

To suspend a foreground job, send a stop signal (CTRL/Z) to the shell process currently running in the foreground. To suspend a background job, use the **stop** command. When jobs are suspended, they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job stops and reports this fact, much like it reports the termination of background jobs. For foreground jobs, this looks like

```
% du > usage
CTRL/Z
Stopped
%
```

The shell displays the "Stopped" message when it notices that a job (in this case, the **du** program) has stopped. When you use the **stop** command on a background job, the shell prints a slightly different message:

```
% sort usage &
[1] 23
% stop %1
[1] + Stopped (signal) sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended, then continued as background jobs using the **bg** command, allowing you to continue other work and stop waiting for the foreground job to finish. In this sequence, we start **du** in the foreground, stop it before it finishes, then continue it in the background:

```
% du > usage
CTRL/Z
Stopped
% bg
[1] du > usage &
%
```

All job control commands can take an argument that identifies a particular job. All job name arguments must begin with a percent (%), since some of the job control commands also accept process numbers. To get the numbers of all running or suspended processes, use the **jobs** command.

The default job (when no argument is given) is called the **current job** and is identified by a plus sign (+) in the output of the **jobs** command. When only one job is stopped or running in the background, it is always the current job. No argument is needed in this case. If you stop a job running in the foreground, it becomes the current job and the existing current job becomes the previous job, identified by a dash (-) in the output of **jobs**. When the current job terminates, the previous job becomes the current job.

When given, the argument to **jobs** is one of the following:

%-	the previous job
%n	where *n* is the job number

%pref	where *pref* is some unique prefix of the command name and arguments of one of the jobs
%?string	where *string* is a string found in only one of the command lines that set up a job.

The **jobs** command lists the table of jobs, giving the job number, commands, and status ("Stopped" or "Running") of each background or suspended job. With the **-l** option, the process numbers are also given.

```
% du > usage &
[1] 33
% ls -s | sort -n > myfile &
[2] 34
% mail ers
CTRL/Z
Stopped
% jobs
[1] - Running du > usage
[2]   Running ls -s | sort -n > myfile
[3] + Stopped mail ers
% fg %ls
ls -s | sort -n > myfile
% more myfile
```

The **fg** moves a job into the foreground. If the job is suspended, it is restarted. If the job is already running in the background, it continues to run, but becomes the foreground job; consequently, it can accept signals or input from the terminal. In the above example, we use **fg** to change the **ls** job from the background to the foreground since we want to wait for it to finish before looking at its output file.

The **bg** command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the stop signal. The combination of the stop signal and the **bg** command changes a foreground job to a background job. The **stop** command suspends a background job.

The **kill** command terminates a background or suspended job immediately. In addition to jobs, **kill** may be given process numbers as arguments. Thus, in the example above, the running **du** command can be terminated as shown here:

```
% kill %1
[1] Terminated du > usage
%
```

The **notify** command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes, instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal, it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the **s** (substitute) command in the text editor might take a long time:

```
% ex bigfile
"bigfile"
1,$s/thisword/thatword/
CTRL/Z
Stopped
% bg
[1] ex bigfile &
%
  . . . some foreground commands . . .
[1] Stopped (tty input) ex bigfile
% fg
ex bigfile
wq
"bigfile"
%
```

After we issue the **s** command, we stop the **ex** job with CTRL/Z, and then put it in the background using **bg**. Some time later when the **s** command is finished, **ex** tries to read another command and is stopped because jobs in the background cannot read from the terminal. The **fg** command returns the **ex** job to the foreground where it can once again accept commands from the terminal.

> **NOTE:** The **jobs** command only prints jobs started in the currently executing shell. It knows nothing about background jobs started in other shells. Use **ps** to find out about background jobs not started in the current shell.

Working Directories

The shell is always in a particular working directory. The "change directory" command, **cd**, changes the working directory of the shell. It's useful to make a directory for each project you work on, then place all files related to that project in that directory. The "make directory" command, **mkdir**, creates a new directory. The "print working directory" command, **pwd**, reports the absolute pathname of the working directory of the shell, i.e., the directory in which you are located. Thus, in this example, we create the directory **newdocs** and then move to it:

```
% pwd
//ice/kate
% mkdir newdocs
% cd newdocs
% pwd
//ice/kate/newdocs
%
```

No matter where you move to in a directory hierarchy, you can return to your home directory by typing the **cd** command with no arguments:

```
% cd
```

The name .. ("dot dot") always means the directory above the current one. Thus,

```
% cd ..
```

changes the shell's working directory to the parent of (the directory immediately above) the current directory. The name can be used in any pathname; thus,

```
% cd ../programs
```

moves you to the directory **programs** contained in the directory above the current one. If you have several directories for different projects under your home directory, this shorthand notation makes it easier to switch between them.

The shell always remembers the pathname of its current working directory in the variable **cwd**. The shell can also be requested to remember the previous directory when you change to a new working directory. If the "push directory" command, **pushd**, is used in place of the **cd** command, the shell saves the name of the current working directory on a directory stack before changing to the new one. You can see this list at any time by typing the "directories" command **dirs**.

```
% pushd newpaper/references
/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
% dirs
/usr/lib/tmac ~/newpaper/references &
% popd
~/newpaper/references ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde as shorthand for your home directory. The directory stack is printed whenever more than one entry is on it and it has changed. It is also printed by a **dirs** command, which is usually faster and more informative than **pwd**, since it shows the current working directory as well as any other directories remembered in the stack.

The **pushd** command with no argument alternates the current directory with the first directory in the list. The "pop directory" command, **popd**, used without an argument, returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it).

Typing **popd** several times in a series takes you backward through the directories you had been in (changed to) via the **pushd** command. Other options to **pushd** and **popd** manipulate the contents of the directory stack and change to directories not at the top of the stack. See **csh** in the *BSD Command Reference* for details.

Since the shell remembers the working directory in which each job was started, it warns you when it thinks you might be restarting a foreground job that has a different working directory than the current working directory of the shell. Thus, if you start a background job, change the shell's working directory, then bring a background job into the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs -l
//ice/kate
% cd myproject
% dirs
/myproject
% ex prog.c
"prog.c"
CTRL/Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ex prog.c
```

This way the shell warns you of an implied change of working directory, even though no **cd** command was issued. In our example, the **ex** job is still in **//ice/kate/myproject** even though the shell changes to **//ice/kate/**. A similar warning is given when such a foreground job terminates or is suspended (using the stop signal) since a return to the shell implies a change of working directory.

```
% fg
ex prog.c
. . . after some editing
q
working dir is now: ~
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell assumes that a job stays in the same directory where it started. The **-l** option of **jobs** types the working directory of suspended or background jobs when it is different from the current working directory of the shell.

Useful Built-In Commands

The **alias** command is used to assign new aliases and to show existing aliases. With no arguments, it prints a list of the current aliases. With a single argument, such as

```
% alias ls
```

alias shows the current alias for that argument (i.e., **ls**).

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions produce.

The **history** command shows the contents of the history list. The numbers given with the history events help to reference previous events that are difficult to reference using the contextual mechanisms introduced above. Also, a shell variable called **prompt** tells the C shell to use a specific character or string as the prompt. Thus, if you type

```
% set prompt='\! % '
```

the shell prepends the number of the current command in the history list to the percent sign. Note that the exclamation point had to be escaped here even within single quotes (').

The **exit** command can be used to terminate a shell in which **ignoreeof** is set.

The **rehash** command causes the shell to recompute a table of command locations. You must use **rehash** if a command is added to a directory in the current shell's search path. If a command isn't in the search path when the hash table is computed, the shell won't know that it exists.

The **repeat** command can be used to repeat a command several times. Thus, to make five copies of the file **one** in the file **five**, you could do this:

```
% repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the C shell environment. Thus,

setenv TERM vt100

sets the value of the environment variable **TERM** to **vt100**. To print out the environment, use **setenv** as shown here (the **setenv** command, with no arguments, prints out the current values):

```
% setenv
USER=kate
LOGNAME=kate
PROJECT=none
ORGANIZATION=doc
NODEID=1054
PATH=://ice/kate/bin:/usr/ucb:/bin:usr/bin
SYSTYPE=bsd4.3
TERM=apollo_1280bw
NODETYPE=DN3000
TZ=EST5EDT
HOME=//ice/kate
```

The **source** command forces the current shell to read commands from a file. Thus, use

source .cshrc

after making a change to the **.cshrc** file to have the change take effect immediately. The **unalias** command cancels aliases.

The **unset** command removes shell variables, and **unsetenv** removes environment variables.

Shell Control Structures and Shell Scripts

This section describes how to place commands in special files (**shell scripts**) that invoke shells for reading and executing commands.

Invocation and the argv Variable

To run a C shell script, you may type

% csh *scriptname args*

The *scriptname* argument is the name of the file containing a group of **csh** commands and *args* denotes a a sequence of optional arguments. The shell places these arguments in the variable **argv** and then begins to read commands from the script. These arguments placed in **argv** are made available as if they were ordinary shell variables. If you make the file *scriptname* executable by typing

% chmod 755 *scriptname*

or by typing

% chmod +x *scriptname*

or even

% chacl +x *scriptname*

and place the line

#!/bin/csh

as the first line of the file *scriptname*, a C shell is automatically invoked to execute *scriptname* when you type

scriptname

In general, you should always start a shell script with a line of the the form

#!shell

The *shell* argument is the name of the shell that is to execute the script. Common *shell*s are:

/bin/csh	the C shell
/bin/sh	the Bourne shell
/bin/ksh	the Korn shell

If the file does not begin with the #! notation, the shell in which you invoked the script tries to execute it, with unpredictable results.

Variable Substitution

After each input line is broken into words and history substitutions are made, the input line is parsed into distinct commands. Before each command is executed, the shell does variable substitution on these words. Variable substitution is keyed by the dollar sign ($), and is a procedure by which the shell replaces the names of variables by their values. Thus,

echo $argv

when placed in a command script causes the current value of the variable **argv** to be echoed to the output of the shell script. It is an error for **argv** to be unset at this point.

The C shell provides a number of notations for accessing components and attributes of variables. The notation $?*name* expands to 1 if *name* is set and to 0 otherwise. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation $#*name* expands to the number of elements in the variable *name*. To illustrate this, consider the following:

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $#argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus, **$argv[1]** gives the first component of argv (in the example above **a**). Similarly, **$argv[$#argv]** gives **c**, and **$argv[1-2]** gives **a b**. Other notations useful in shell scripts are **$***n* (where *n* is an integer) as a shorthand for **$argv[***n***]** the *n*th parameter and **$*** which is a shorthand for **$argv**.

The form **$$** expands to the process number of the current shell. This process number is unique on the node, and can be used in generation of unique temporary filenames. The form **$<** is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus, the sequence

```
#!/bin/csh -f
#
echo -n 'yes or no?'
set a=($<)
```

writes out the prompt "yes or no?" without a newline and then reads the answer into the variable **a**. In this case, **$#a** is 0 if either a blank line or end-of-file (CTRL/D) is typed. The form **$argv[***n***]** yields an error if *n* is not in the range **1-$#argv** while **$***n* never yields an out-of-range subscript error. It is never an error to give a subrange of the form *n*-.

Expressions

It's important to be able to evaluate expressions in the shell based on the values of variables. All the arithmetic operations of C are available in the shell with the same precedence that they have in C. In particular, the operations == and != compare strings and the operators && and || implement the boolean and/or operations. The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (e.g., *, ?, or [ ]), and the test is whether the string on the left matches the pattern on the right.

The shell also allows file inquiries of the form

-? *filename*

where the question mark is replaced by a number of single characters. For instance, the expression primitive

-e *filename*

tells whether the file *filename* exists. Other primitives test for read, write, and execute access to the file, whether it is a directory, or has non-zero length. You can test whether a command terminates normally, by a primitive of the form

{ *command* }

which returns true (i.e., 1) if the command succeeds (exits normally with exit status 0), or 0 if the command terminates abnormally or with exit status nonzero. If you need more detailed information about the execution status of a command, execute it, then examine the **$status** variable.

> **NOTE:** Since **$status** is set by every command, you must save a particular command's **$status** if you can't examine it immediately following the command's execution.

A Sample Shell Script

The following shell script, called **copyc**, uses the C shell's expression mechanism and some of its control structures:

```
#!/bin/csh -f
# copyc copies those C programs in the
# specified list to the directory ~backup if
# they differ from the files already in ~backup
#
set noglob
foreach i ($argv)
     if ($i !~ *.c) continue
     # not a .c file so do nothing
     if (! -r ~backup/$i:t) then
          echo $i:t not in backup... not cp\'ed
     continue
     endif
cmp -s $i ~backup/$i:t # to set $status
     if ($status != 0) then
          echo new backup of $i
          cp $i ~backup/$i:t
          endif
     end
```

First, we specify that this is a **/bin/csh** script. The **-f** option prevents the user's **.cshrc** file from being read. This is recommended for all scripts, not only because it makes them faster, but also because it prevents interference from unexpected aliases or settings of a particular user.

This script uses the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching **end** for each of the values given between the left and right parentheses with the named variable (i.e., **i** set to successive values in the list).

Within this loop, you may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (**i** in this case) has the value it was assigned at the last iteration.

Here, we set the variable **noglob** to prevent filename expansion of the members of **argv**. This is recommended if the arguments to a shell script are filenames that have already been expanded or if arguments may contain filename expansion metacharacters. You can also quote each use of a dollar sign variable expansion, though this is rather tedious.

The other control construct used here is a statement of the form

if (*expression*) **then**
    command
    ...
endif

The placement of the these keywords is not flexible. The following two formats are unacceptable to the C shell:

#this won't work
if (*expression*)
then
    command
    ...
endif

#nor will this
if (*expression*) **then** *command* **endif**

The shell does have another form of the **if** statement:

if (*expression*) *command*

For the sake of appearance, this can be written with an escaped newline:

if (*expression*) \
    command

The *command* must not involve a pipe (|), ampersand (&), or semi-colon (;). It must not be another control command. In the second form, the final backslash (\) must immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**, as shown here:

```
if ( expression ) then
    commands
else if (expression ) then
    commands
    ...

else
    commands
endif
```

Another important mechanism used in shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract a root of a filename, or **:e** to extract the extension. Thus, if the variable **i** has the value **/mnt/foo.bar**, then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
```

shows how the **:r** modifier strips off the trailing **.bar** and the the **:e** modifier leaves only the **bar**. Other modifiers take off the last component of a pathname and leave the head **:h**, or all but the last component of a pathname and leave the tail **:t**. (These modifiers are fully described under **csh** in the *BSD Command Reference*.) You can also use the command substitution mechanism, described in the next major section, to perform modifications on strings.

The C shell allows only one colon modifier on a dollar sign substitution. Thus, the following doesn't produce the results that one would otherwise expect:

```
% echo $i $i:h:t
/a/b/c /a/b:t
```

Finally, we note that the pound sign (#) lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a pound sign are discarded by the shell. This character can be quoted using a single quote (') or backslash (\) to place it in an argument word.

Other Control Structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

while (*expression*)
    commands
end

and

switch (*word*)
    case str1:
        commands
        breaksw
...
    case strn:
        commands
        breaksw
    default:
        commands
        breaksw
endsw

> **NOTE:** The C shell uses **breaksw** to exit from a **switch**, while **break** exits a **while** or **foreach** loop. A common mistake in C shell scripts is the use of **break** rather than **breaksw** in switches.

Finally, **csh** allows a **goto** statement, with labels that look the same as they do in C:

loop:
    commands
    goto loop

Supplying Input to Commands

By default, commands run from shell scripts receive the standard input of the shell that is running the script. This allows shell scripts to participate in pipelines, but mandates extra notation for commands that are to take in-line data.

Thus, we need a metanotation for supplying in-line data to commands in shell scripts. As an example, consider this script that runs the editor to delete leading blanks from the lines in each argument file:

```
#!/bin/csh -f
# deblank, a script to remove leading blanks
foreach i ($argv)
     ed - $i << 'EOF'
     1,$s/"[ ]*//
     w
     q
     'EOF'
    end
%
```

The **<< 'EOF'** notation means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly **'EOF'** itself. The fact that the EOF is quoted causes the shell to forego variable substitution on the intervening lines. In general, if any part of the word following the "<<" that the shell uses to terminate the text to be given to the command is quoted, then variable substitutions are not performed. In this case, since we used the form "1,$" in our editor script, we needed to ensure that this dollar sign did not trigger a substitution. We could also have ensured this by preceding the dollar sign here with a backslash, as in the following line, but quoting the EOF terminator is a more reliable way of achieving the same result.

```
1,\$s/"[ ]*//
```

NOTE: Although it is good practice to indent your scripts to show blocks of commands, leading spaces and tabs are copied inside "<<"s, so they cannot be indented without consequences.

Catching Interrupts

If your shell script creates temporary files, you may wish to catch interruptions of the shell script so that you can clean up these files. To do this, use

onintr *label*

The *label* argument is a label in the program. If an interrupt is received, the shell does a **goto** *label*. You can then remove the temporary files and use **exit** (built in to the shell) to exit the shell script. To exit with a nonzero status, status 1, type the following:

exit(1)

Additional Options

Other features of the shell are useful to writers of shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can help trace the actions of the shell. The **-n** option causes the shell to read commands but not to execute them, something which may be useful during debugging.

The double quote (") mechanism allows only some of the expansion mechanisms discussed thus far to occur on the quoted string, and serves to make this string into a single word as a single quote (') does.

Other Shell Features

The C shell features discussed in this section are less commonly used. In particular circumstances, it may be necessary to know the exact nature and order of different substitutions performed by the shell. The precise meaning of certain combinations of quotations is also important at times. Furthermore, the shell has many command line option flags mostly used in the writing of UNIX programs, and debugging of shell scripts. See **csh** in the *BSD Command Reference* for more information.

Loops at the Terminal and Variables as Vectors

Occasionally, the **foreach** control structure may be used at a terminal to aid in performing a number of similar commands. For instance, to count the number of files in several directories (**dir1**, **dir2**, and **dir3**) that had the characters **.TS** or **.EQ** at the beginning of a line, you can use several command lines:

```
% grep -c '^\.TS|.EQ' dir1
3
% grep -c '^\.TS|.EQ' dir2
5
% grep -c '^\.TS|.EQ' dir3
6
```

or you can use **foreach** to do this more easily:

```
% foreach i (dir1 dir2 dir3)
? grep -c  '^\.TS|.EQ'  $i
? end
3
5
6
%
```

Here, the shell prompts for input with a question mark (?) when reading the body of the loop. Variables containing lists of filenames or other words are also useful in loops. You can, for example, do the following:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The **set** command here gave the **a** variable a list of all the filenames in the current directory as value. You can then iterate these names to perform any chosen function.

The output of a command within backquotes (`) is converted by the shell to a list of words. You can also place the backquoted string within double quotes to take each (nonempty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier **:x** can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

Braces { ... } in Argument Expansion

Another form of filename expansion involves the brace characters ({ }), which specify that the delimited strings separated by a comma (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus,

A{str1,str2,...strn}B

expands to:

Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames need not exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments that are not filenames, but have common parts. For example,

% mkdir ~/{hdrs,retrofit,csh}

makes subdirectories **hdrs**, **retrofit**, and **csh** in your home directory. This is most useful when the common prefix is longer than shown in the following example:

chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}

Command Substitution

A command enclosed in backquotes (`) is replaced, just before filenames are expanded, by the output from that command. You may type the following to save the current directory in the variable **pwd**:

```
% set pwd=`pwd`
```

You may type this line to run the editor **ex**, supplying as arguments filenames ending in **.c** and that contain the string **TRACE**:

```
% ex `grep -l TRACE *.c`
```

NOTE: Command expansion also occurs in input redirected with double right angle brackets (<<) and within double quoted (") notations. See the *BSD Command Reference* for details.

Chapter 9

Using the Bourne Shell

The Bourne shell (named for its author, S. R. Bourne) is a language that provides a programmable interface to the SysV environment. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **case**, **if-then-else**, and **for** are supported, as is 2-way communication between the shell and commands. String-valued parameters (i.e., filenames or flags) may be passed to a command. Also, commands set a return code that may be used to determine flow of control. The standard output from a command may also serve as shell input.

The Bourne shell can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through pipes can be invoked. Commands are found by searching directories in the file system in a user-defined sequence. Commands can be read either from the keyboard, or from a file, which allows command scripts to be stored for later use.

The first part of this chapter covers most of the everyday requirements of shell users. Later, we describe those features of the shell primarily intended for use within shell scripts, including control-flow primitives and string-valued variables provided by the shell. Knowing another programming language might help you understand this better. Finally, we describe the more advanced features of the shell. Appendix C contains a summary of Bourne shell grammar; Appendix D summarizes valid Bourne shell metacharacters and reserved words.

Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
$ who
```

is a command that prints the names of everybody currently logged in to a node in the network. The command

```
$ ls -l
```

prints a list of files in the current directory. The -l argument tells the **ls** (list directory) command to give a long listing of the directory (i.e., print status information, size, and the creation date of each file).

Background Commands

When the Bourne shell executes a command, it normally runs it from within the shell process, waits for it to finish, then prompts for more input. You may also have the shell run a command and then accept additional input before the command finishes. Thus,

```
$ cc pgm.c &
```

calls the C compiler to compile the file **pgm.c**. The trailing ampersand (&) is an operator that instructs the shell not to wait for the command to finish. To help you keep track of such a process, the shell reports its process number following its creation. Use the **ps** (process status) command to get a list of currently active processes.

Input/Output Redirection

Most commands produce output on the standard output (normally, the transcript pad of the window in which the Bourne shell is running).

This output may be redirected to a file by writing, for example,

```
$ ls -l > file
```

The shell interprets the notation > *file* and does not pass it as an argument to the **ls** (list directory) command. If *file* doesn't exist, the shell creates it; otherwise, the original contents of *file* are used. Then the output from **ls** is put in *file*. You may also append output to a file by using this notation:

```
$ ls -l >> file
```

Here too, *file* is created if it does not already exist; however, if it does, the output will be added to the end of *file*.

The standard input of a command may be taken from a file instead of the node by writing, for example,

```
$ wc < file
```

The command **wc** (word count) reads its standard input (in this case, redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then this could be used:

```
$ wc -l < file
```

Pipelines and Filters

The standard output of one command may be connected to the standard input of another by using the "pipe" operator (|), as in

```
$ ls -l | wc
```

Two commands connected in this way constitute a "pipeline" and the overall effect is the same as

```
$ ls -l > file; wc < file
```

except that no *file* is used. Instead, the two processes are connected by a pipe and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

Many UNIX commands are called "filters." A **filter** is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, the **grep** (search a file for a pattern) command, selects from its input those lines that contain some specified string. Thus,

```
$ ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string **old**. Another useful filter is the **sort** (sort or merge files) command, which can be used, for example, to print an alphabetically sorted list of users logged-in. The command line for doing this would be as follows:

```
$ who | sort
```

A pipeline may consist of more than two commands. For example,

```
$ ls | grep old | wc -l
```

prints the number of filenames in the current directory containing the string **old**.

Generating Filenames

Many commands accept filenames as arguments. For example, this prints information relating to the file **main.c**:

```
$ ls -l main.c
```

The shell provides a means of generating a list of filenames that match a pattern; for example,

```
$ ls -l *.c
```

generates, as arguments to **ls**, all filenames in the current directory that end in **.c**. In this context, the asterisk is a metacharacter "pattern" that matches any string including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string, but not a leading dot (.).
?	Matches any single character.
[...]	Matches any one of the enclosed characters.

A pair of characters separated by a dash (-) matches any character lexically between the pair. For example, consider the following:

[a-z]*	Matches all names in the current directory beginning with one of the letters **a** through **z**.
/usr/tom/no/?	Matches all one-character names in the directory **/usr/tom/no**. If no filename matches the pattern, then the pattern is passed, unchanged, as an argument.

This expansion of metacharacters to pathnames is called **globbing**. Globbing is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
$ echo /usr/fred/*/*.bin
```

finds and prints the names of all files of the form *file*.**bin** in sub-directories of **/usr/fred**. The **echo** (echo arguments) command simply prints its arguments, separated by blanks. Using this last feature can be time-consuming, requiring, in this case, a scan of all sub-directories of **/usr/fred**.

There is one exception to the general rules given for patterns. The dot (.) at the start of a filename must be explicitly matched. Therefore,

```
$ echo *
```

echoes all filenames in the current directory not beginning with a dot. This echoes all those filenames beginning with a dot:

```
$ echo .*
```

It avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory," respectively. (Notice that, by default, the **ls** (list directory) command suppresses listing of information for the files "." and "..".) Filenames starting with "." are also used to hide from common use start-up and other files which are normally ignored.

Quotation

As we have mentioned, characters that have a special meaning to the shell are called metacharacters. A complete list of Bourne shell metacharacters appears in Appendix D, but some of the more common ones are shown in Table 9-1.

Table 9–1. Some Common Bourne Shell Metacharacters

Character	Description
<	Redirects input
>	Redirects output
*	Matches any set of characters
?	Matches any single character
&	Designates a background command
|	Designates a pipe

A character preceded by a backslash (\) is said to be "quoted"or "escaped" and loses any special meaning it may otherwise have had. Since the backslash is elided, **echo**, used as shown, returns the following strings

```
$ echo \?
?
$ echo \\
\
```

To allow long strings to be continued over more than one line, the shell ignores the sequence \newline. The backslash is convenient for quoting single characters. When more than one character needs quoting, we recommend the easier method of enclosing the string between single quotes. For example,

```
$ echo xx'****'xx
xx****xx
```

The quoted string may not contain the single quote character ('), but it may contain newlines, which are preserved. We recommend this simple quoting for casual use. A third quoting mechanism, which uses double quotes to prevent interpretation of some, but not all, metacharacters is discussed in another section.

Prompting

The shell issues a prompt when it is ready for more input. The default Bourne shell prompt is a dollar sign ($) followed by a space. The prompt may be changed. For example, to set the prompt to the string "yesdear ", type this:

```
$ PS1="yesdear "
```

If a newline is typed and further input is needed, the shell issues the secondary prompt, a greater-than symbol (>) followed by a space. If this happens unexpectedly, type an interrupt to return the main shell prompt. You may also change this prompt.

For example,

```
$ PS2="moredear "
```

sets the prompt to the string "moredear ".

Starting the Bourne Shell

When you log in to a Domain node, the DM (Display Manager) looks in several places for information about what windows to open and what processes to start (see Chapter 3 for more detailed information). It normally opens a specified shell, then looks for the file:

~/user_data/startup_dm.*display_type*

The *display_type* argument matches the type of display in use (for example, 1280bw). If you include a command line such as:

```
(0,200)dr; (540,600)cp /bsd4.3/bin/sh -n bourne_shell
```

in your **startup_dm** file, the DM automatically opens a BSD4.3 Bourne shell when you log in. Since we included the **-n** option, the process is named **bourne_shell**.

You may also define a key or function key to open a Bourne shell. This DM command defines the shifted L5 key (labeled <SHELL>) so that pressing SHIFT/<SHELL> opens a Bourne shell:

kd l5s cp /bin/sh ke

When you log in, the shell sets the working directory to your home directory and begins reading commands from a file named **.profile** before reading commands from the node's keyboard or any other file. Every Bourne shell you start as a log-in shell reads from this file.

The environment variable **ENV** determines whether the Bourne shell runs a start-up file for every new shell. If you define **ENV**, the Bourne shell takes it value as the name of a start-up script to execute. The **.profile** file may assign a value to **ENV** and that script will be run after **.profile** is finished. See "Shell Start-Up Files" in Chapter 7 for further information.

Shell Scripts

The shell may be used to read and execute commands contained in a file, for example,

$ sh *file* [*argument* ...]

starts a new shell which reads commands from *file*. Such a file is called a shell script. Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1**, **$2**, ... **$9**. For example, if the file **wg** contains

```
who | grep $1
```

then the following command line

$ sh wg fred

is equivalent to

```
$ who | grep fred
```

Files have three independent attributes: read, write, and execute. You can use **chmod** (change mode) or **chacl** (change ACL) to make a file executable. For example,

```
$ chmod +x wg
```

ensures that the file **wg** has execute status. Following this, the command

```
$ wg fred
```

is equivalent to

```
$ sh wg fred
```

This allows programs and shell scripts be used interchangeably. Besides providing names for positional parameters, the number of positional parameters in the call is available as $#. The name of the file being executed is available as $0.

A special shell variable **$*** is used to substitute for all positional parameters except $0. Typically, this is used to access the whole list of arguments, as in the following shell script named **list_links**, which simply prepends some arguments to those already given:

```
#!/bin/sh
# list_links - command to list links
ls -S  $*
```

Control Flow Using for Statements

shell scripts are frequently used to loop through the arguments ($1, $2, ...) executing commands once for each argument. For example, consider the following program that searches a file of corporate phone numbers containing lines of the form

```
tony 8756
bob 9934
sherry 4368
...
richard 5335
```

If this file is called **/usr/lib/telnos**, then the text of the shell script **tel** is

```
#!/bin/sh
for i
do
     grep $i /usr/lib/telnos;
done
```

This command line prints those lines in **/usr/lib/telnos** that contain the string **sherry**

```
$ tel sherry
```

while this prints those lines containing **sherry** followed by those for **richard**:

```
$ tel sherry richard
```

The **for** loop notation of the shell has the general form

for *name* **in** *w1 w2 ...*
do
    command-list
done

A *command-list* is a sequence of one or more simple commands separated by a newline or semicolon. Furthermore, the shell only recognizes reserved words like **do** and **done** when they follow a newline or semicolon. The shell variable *name* is set to the words *w1 w2 ...* in turn each time the *command-list* following **do** is executed. If **in** *w1 w2 ...* is omitted, then the loop is executed once for each positional parameter; that is, **in** **$*** is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
#!/bin/sh
for i
do
     > $i;
done
```

The command line

```
$ create alpha beta
```

ensures that two empty files **alpha** and **beta** exist and are empty. The notation > *file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

Control Flow Using case Statements

The Bourne shell's **case** statement provides a multiway branching mechanism. For example, the following is an **append** command:

```
#!/bin/sh
case $# in
          1) cat >> $1 ;;
          2) cat >> $2 <$1 ;;
          *) echo 'usage: append [ from ] to' ;;
esac
```

When called with one argument as in

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the **cat** command.

When called with two arguments as in

append *file1 file2*

the contents of *file1* are appended to *file2*. If the number of arguments supplied to **append** is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

case *word* **in**
pattern) *command-list*;;
...
esac

The shell attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, then the associated *command-list* is executed and execution of the **case** is complete. Since a single asterisk (*) is the pattern that matches any string, it can be used for the default case.

> **NOTE:** The shell doesn't check to see that only one pattern matches the **case** argument. The first match found by the shell defines the set of commands to be executed.

In this example, the commands following the second asterisk (*) are never executed.

```
#!/bin/sh
case $# in
        *) ... ;;
        *) ... ;;
esac
```

The **case** construction may also be used to distinguish between different forms of an argument. The following example is a fragment of a **cc** (C compiler) command:

```
#!/bin/sh
for i
do
   case $i in
         -[ocs]) ... ;;
         -*) echo 'unknown flag $i' ;;
         *.c) /lib/c0 $i ... ;;
         *) echo 'unexpected argument $i' ;
   esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a pipe character (|). Thus,

```
case $i in
         -x|-y) ...
esac
```

is equivalent to

```
case $i in
         -[xy]) ...
esac
```

The usual quoting conventions apply, so that

```
case $i in
         \?) ...
```

matches a question mark (?).

Here Documents

The shell script **tel**, illustrated previously, uses the file **/usr/lib/telnos** to supply the data for **grep**. Alternatively, this data may be included within the shell script as a "here document." For example,

```
#!/bin/sh
for i
do
     grep $i << !
...
richard 5335
sherry 4368
...
!
done
```

In this case, the shell takes the lines between <<! and ! as the standard input for **grep**. The exclamation point (!) is arbitrary. The here document is terminated by a line that consists of the character (or string) following the lesser-than characters (<<).

Parameters are substituted in the document before it is made available to one command as illustrated by the following script called **edg**:

```
#!/bin/sh
ed $3 << %
g/$1/s//$2/g
w
%
```

The call

```
$ edg string1 string2 file
```

is then equivalent to the **ed** commands

```
ed file << %
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. To prevent substitution, use a backslash (\) to quote the special dollar sign character ($) as in

```
$ ed $3 << EOF
1,\$s/$1/$2/g
w
EOF
```

This version of **edg** is equivalent to the first except that **ed** prints a question mark if no occurrences of the string $1 appear. You can entirely prevent substitution within a here document by quoting the terminating string. For example,

```
grep $i << \EOF
...
EOF
```

The document is presented without modification to **grep**. If parameter substitution is not required in a here document, this latter form is more efficient.

Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. You can use the **set** command to examine all currently set variables.

Variables may be given values by writing, for example,

user=fred box=m000 acct=mh0000

assigns values to the variables **user**, **box**, and **acct**. A variable may be set to the null string. The following line sets the variable **null** to the null string:

null=

The value of a variable is obtained by preceding its name with a dollar sign ($). For example, the following line echoes **fred**:

$ echo $user

Variables may be used interactively to provide abbreviations for frequently used strings. For example, this moves (using the **mv** command) the file **pgm** from the current directory to the directory **/usr/fred/bin**:

```
$ b=/usr/fred/bin
$ mv pgm $b
```

A more general notation is available for parameter (or variable) substitution, as in

```
$ echo ${user}
```

which is equivalent to

```
$ echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
$ tmp=/tmp/ps
$ ps a > ${tmp}a
```

directs the output of **ps** to the file **/tmp/psa**, whereas this causes the value of the variable **tmpa** to be substituted:

```
$ ps a > $tmpa
```

Except for **$?**, which is set after every command, the Bourne shell sets these variables when invoked:

$? The exit status (decimal string return code) of the most-recently-executed command. Most commands return a zero if they execute successfully, and a nonzero status otherwise. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). This is used, for example, in the **append** command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes on the same node, this string is frequently used to generate unique temporary filenames. For example,

```
$ ps a > /tmp/ps$$
...
$ rm /tmp/ps$$
```

$! The decimal process number of the last process run in the background.

$- The current shell flags, such as **-x** and **-v**.

Some variables have special meaning to the shell; avoid their use elsewhere.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since last examined, the shell prints the message "you have mail" before prompting for the next command. This variable is typically set in the file **.profile** in your home directory, e.g., **MAIL=/usr/mail/***name*, where *name* is your user ID.

$HOME The user's log-in or home directory. This is the default argument for the **cd** command.

$PATH A list of directories that contain commands. Each time a command is executed by the shell, a list of directories is searched for an executable file of that name. **$PATH** consists of directory names separated by colons (:), for example,

```
$ PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first colon), **/usr/fred/bin**, **/bin** and **/usr/bin** are to be searched in that order.

Thus, individual users can have "private" commands that are accessible independently of the current directory. If the command name contains a slash (/), this directory search is not used. The shell makes a single attempt to execute the command.

$PS1 The primary shell prompt string; by default, a dollar sign ($).

$PS2 The shell prompt when further input is needed; by default, a greater–than character (>).

$IFS The set of characters used by blank interpretation.

$ENV The pathname of a shell start–up script to execute whenever you start a new shell.

$SHELL When the shell is invoked, it scans the environment for this name (in this case, **/bin/sh**).

$SYSTYPE The environment as set by "systype".

The test Command

The **test** (evaluate condition) command has a number of uses in shell programs. For example,

$ test –f *name*

returns zero exit status if *name* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given here. See **test** for a complete specification.

test *s* true if *s* is a nonnull string

test –f *name* true if *name* is a file that exists

test –r *name* true if *name* is a readable file

test –w *name* true if *name* is a writable file

test -d *name* true if *name* is a directory that exists

test -L *name* true if *name* is a soft link

NOTE: In determining whether an object is a soft link, **test -d** *name* also returns true if *name* is a soft link that points to a directory. Furthermore, **test -f** *name* returns true if *name* is a soft link that points to a file. If *name* is a soft link that points to a nonexistent object, then **test -f** *name* returns false while **test -L** *name* returns true.

Control Flow Using while Statements

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if-then-else** branch are also provided. The actions of **while**, **until**, and **if-then-else** are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*
do
    loop-command-list
done

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time around the loop, *command-list* is executed. If a zero exit status is returned, then *command-list* is executed; otherwise, the loop terminates. Thus,

```
#!/bin/sh
while test $1
do ...
   shift
done
```

is equivalent to the following:

```
#!/bin/sh
for i
do ...
done
```

Shift is a shell command that renames the positional parameters $2, $3, ... as $1, $2, ... and loses **$1** (that is, it shifts them to the left).

You can also use the **while/until** loop to make the shell wait until an external event occurs, before running commands. An **until** loop reverses the termination condition. For example, this does a loop every five minutes until **file** exists (presumably another process creates the file):

```
#!/bin/sh
until test -f file
do
    sleep 300;
done
commands
```

Control Flow Using if Statements

The Bourne shell also provides a general conditional branch of the following form, which tests the value returned by the last simple command following **if**:

```
if command-list
then
    command-list
else
    command-list
fi
```

The **if** command may be used along with the **test** (evaluate condition) command to test for the existence of a file as in the following

```
if test -f file
then
    process file
else
   do something else
fi
```

A multiple test **if** command of the form

```
if ...
then ...
else if ...
then ...
else if ...
...
fi
fi
fi
```

may be written using an extension of the **if** notation as

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following shows the **touch** command, which changes the "last modified" time for a list of files. The command may be used along with **make** to force recompilation of a list of files.

```
#!/bin/sh
flag=
for i
do
     case $i in
          -c) flag=N ;;
          *)  if test -f $i
          then
              touch $i
          elif
              test $flag
          then
              echo file \'$i\' does not exist
          else
              > $i
          fi
      esac
done
```

The **-c** flag in this command forces subsequent files to be created if they don't already exist. Otherwise, an error message would be printed. The shell variable **flag** is set to a nonnull string if the **-c** argument is found. The "touch" updates the last modified time of a file to be the current time.

The sequence

if *command1*
then *command2*
fi

may be written as

command1 **&&** *command2*

Conversely, the following sequence

command1 || *command2*

executes *command2* only if *command1* fails. In each case, the value returned is that of the last simple command executed.

Command Grouping

Commands may be grouped in one of the following two ways:

{ *command-list* ; }
(*command-list*)

In the first example, *command-list* is simply executed; the second executes *command-list* as a separate process. For example, this command line executes **rm junk** in the directory **x** without changing the current directory of the invoking shell:

```
$ (cd x; rm junk )
```

The following commands have the same effect, but they leave the invoking shell in the directory **x**:

```
$ cd x; rm junk
```

Debugging Shell Scripts

The shell provides two tracing mechanisms to help when debugging shell scripts. The first is invoked within the script as

```
$ set -v
```

(**-v** for verbose), causing lines of the script to be printed as they are read. This helps isolate syntax errors. Invoke it without modifying the script by specifying

```
$ sh -v proc
```

where *proc* is the name of the shell script. This flag may be used along with the **-n** flag, which prevents execution of subsequent commands.

> **NOTE:** Using **set -n** renders a node's keyboard useless until you type an end-of-file (EOF).

The command **set -x** produces an execution trace. Following parameter substitution, each command is printed as it is executed. Both flags may be turned off by typing

```
$ set -
```

and the current setting of the shell flags is available as

```
$-
```

Keyword Parameters

Shell variables may be given values by assignment or when a command is invoked. If a command is preceded by statements of the form *name=value*, the assignment is done. The value of *name* in the invoking shell isn't affected. For example, this executes *command* with **user** set to **fred**:

```
user=fred command
```

The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are passed to the command.

You may also use the **set** command to set positional parameters from within a script. For example,

```
$ set - *
```

sets $1 to the first filename in the current directory, $2 to the next, and so on. The dash (-) ensures correct treatment when the first filename begins with a dash.

Parameter Transmission

When a shell script is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell script by specifying in advance that such parameters are to be exported. Thus,

```
$ export user box
```

marks the variables **user** and **box** for export. When a shell script is invoked, all exportable variables are copied for use within the invoked script. Modification of such variables within the script doesn't affect the values in the invoking shell. A shell script may not usually modify the state of its caller without an explicit request on the part of the caller. Shared file descriptors are an exception to this rule.

> **NOTE:** Any new process created that takes its context from the process in which a variable was defined and exported will recognize the new variable. Processes already created (or those created later) that don't take their context from the process where the variable was defined won't apply the variable.

Names whose values are intended to remain constant may be declared **readonly**. The form of this command is the same as that of the **export** command:

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution

In the BSD version of **/bin/sh**, the null string replaces any unset shell parameter. For example, if the variable **d** is not set, the following command echoes nothing:

```
$ echo $d
```

A default string may be given as in the following

```
$ echo ${d-.}
```

which echoes the value of the variable **d** if it is set and a dot (.) otherwise. The default string is evaluated using the usual quoting conventions so that

```
$ echo ${d-'*'}
```

echoes an asterisk (*) if the variable **d** is not set. Similarly,

```
$ echo ${d-$1}
```

echoes the value of **d** if it is set and the value (if any) of $1 otherwise. A variable may be assigned a default value using the notation

```
$ echo ${d=.}
```

which substitutes the same string as

```
$ echo ${d-.}
```

and if **d** were not previously set then it is set to the string ".". (The notation ${...=...} is not available for positional parameters.) If there is no sensible default, then the notation

```
$ echo ${d?message}
```

echoes the value of the variable **d** if it has one; otherwise, the shell prints *message* and abandons the shell script. If **message** is absent, then a standard message is printed. A shell script that requires some parameters to be set might start as follows:

```
#! /bin/sh
: ${user?} ${acct?} ${bin?}
...
```

The colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user**, **acct**, or **bin** are not set, the shell abandons execution of the script.

Command Substitution

The standard output from a command can be substituted in a way similar to that allowed for parameters. The command **pwd** prints on its standard output the name of the current directory. For example, if the current directory is **usr/fred/bin**, then the command

```
$ d='pwd'
```

is equivalent to

```
$ d=/usr/fred/bin
```

The shell takes the entire string between opening single quotes (grave accents,'...') as the command to be executed and replaces it with the output from the command. The command is written using the usual quoting conventions except that a grave accent (') must be escaped with a backslash (\). For example,

```
$ ls 'echo "$1"'
```

is equivalent to

```
$ ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including here documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell scripts. An example of such a command is **basename**, which removes a specified suffix from a string. For example, the following command line prints the string **main**:

```
$ basename main.c .c
```

Its use is illustrated by the following fragment from a **cc** command that sets **B** to the part of **$A** with the suffix **.c** stripped:

```
case $A in
        ...
        *.c) B=`basename $A .c`
        ...
esac
```

For example, the following sets the variable **i** to the names of files in time order, most recent first:

```
for i in `ls -t`;
    do
        ...
```

This prints the date

```
set `date`; echo $6 $2 $3, $4
```

in output of the following format:

```
1984 Dec 14, 23:59:59
```

Evaluation and Quoting

The shell is a macro processor that provides parameter substitution, command substitution and filename generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in the summary of Bourne shell grammar in Appendix C. Before a command is executed, the following substitutions occur:

- parameter substitution (for example, **$user**).
- command substitution (for example, **`pwd`**). Only one evaluation occurs, so that if, for example, the value of the variable *X* is the string *$y*, then the following echoes *$y*:

  $ echo $X

Following these substitutions, the resulting characters are broken into nonblank words. Thus, "blanks" are the characters of the string **$IFS**. By default, this string consists of blank, tab and newline. The null string isn't regarded as a word unless quoted, for example,

```
$ echo ''
```

passes on the null string as the first argument to **echo**, whereas

```
$ echo $null
```

calls **echo** with no arguments if the variable **null** is not set or set to the null string.

Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of filenames is generated to replace the word. Each such filename is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

In addition to the quoting mechanisms described earlier using backslash (\) and the '...' string, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitutions occur but filename generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using a backslash (\):

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example, this passes the value of the variable **x** as a single argument to **echo**:

```
$ echo "$x"
```

Similarly, the following

```
$ echo "$*"
```

passes the positional parameters as a single argument and is equal to

```
$ echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted.

```
$ echo "$@"
```

passes the positional parameters, unevaluated, to **echo** and is equivalent to the following

```
$ echo "$1" "$2" ...
```

Table 9–2 gives, for each quoting mechanism, the shell metacharacters that are evaluated. In this table,

- "t" shows a sequence used as a terminator.
- "y" shows a sequence in which a metacharacter is interpreted.
- "n" shows a sequence in which a metacharacter is not interpreted.

Table 9–2. Evaluation of Bourne Shell Metacharacters by Quoting Mechanisms

Quoting Mechanism	Metacharacters Evaluated					
'	\	$	*	`	"	'
`	n	n	n	n	n	t
"	y	y	n	y	t	n

Among other things, this table shows that the sequence \$ is not interpreted (is passed as a literal $), the sequence \' can be used to terminate a string, and the sequence "$ preserves the meta-meaning of the dollar sign ($). Where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable **X** has the value **$y**, and if **y** has the value **pqr**, then this echoes the string **pqr**:

```
$ eval echo $X
```

In general, **eval** evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. Thus,

```
$ wg='eval who | grep'
$ $wg fred
```

is equivalent to

```
$ who | grep fred
```

Here, **eval** is required since there is no interpretation of metacharacters, such as a pipe character (|), following substitution.

Error Handling

How errors detected by the shell are treated depends on the type of error and whether the shell is being used interactively. An **interactive shell** is one whose input and output are connected to a node as determined by **gtty** (get terminal state). A shell invoked with the **-i** flag is also interactive. Execution of a command may fail because:

- Input/output redirection won't work (for example, a file doesn't exist or can't be created).
- The command itself doesn't exist or cannot be executed.
- The command terminates abnormally.
- The command terminates normally but returns a nonzero exit status.

In every case, the shell goes on to execute the next command. Except in the last case, the shell prints an error message. All remaining errors cause the shell to exit from a command script. An interactive shell returns to read another command from the node's keyboard. Such errors include the following:

- Syntax errors (for example, if ... then ... done).
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the node.
- Failure of any of the built-in commands such as **cd**.

The shell flag **-e** causes the shell to terminate if any error is detected. Many of the UNIX signals used by BSD software are described in Table 9-3. The signals in this list of potential interest to shell programs are 1, 2, 3, 14, and 15. For a complete list, see **signal** in the *BSD Programmer's Reference*.

Table 9-3. UNIX Signals Commonly Used by BSD Software

Signal Number	Description
1	Hangup
2	Interrupt
3*	Quit
4*	Illegal instruction
5*	Trace trap
6*	IOT instruction
7*	EMT instruction
8*	Floating point exception
9	Kill
10*	Bus error
11*	Segmentation violation
12*	Bad argument to system call
13	Write on a pipe with no one to read it
14	Alarm clock
15	Software termination (from **kill**)
16	Domain/OS fault with no UNIX equivalent

Fault Handling

Shell scripts normally terminate when an interrupt is received from the node. The **trap** command is required for necessary clean-up activity (for example, removal of temporary files). For example, this line sets a trap for signal 2 (interrupt):

```
trap 'rm /tmp/ps$$; exit' 2
```

If this signal is received, it executes the commands

```
rm /tmp/ps$$; exit
```

Exit is another built-in command that terminates execution of a shell script. It is required to keep the shell from resuming execution of the script at the place where it was interrupted, once the trap has been taken.

UNIX signals can be ignored (never sent to the process); caught, allowing the process to decide what action to take; or left to cause process termination with no further action. If a signal is ignored on entry to a shell script, for example, by being invoked in the background, then **trap** commands (and the signal) are ignored. This modified version of **touch** shows the use of **trap** in removing a file:

```
#!/bin/sh
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do case $i in
   -c) flag=N ;;
    *) if test -f $i
       then
          touch $1
       elif
          test $flag
       then
          echo file \'$i\' does not exist
       else
          > $i
       fi
    esac
done
```

The **trap** command appears before the creation of the temporary file; otherwise, the process could die without removing the file. Since there is no signal 0, it is used by the shell to indicate the commands to be executed on exit from the shell script.

A script may elect to ignore signals by specifying the null string as the argument to trap. The following fragment, taken from the **nohup** command,

```
trap '' 1 2 3 15
```

causes hangup, interrupt, quit, and kill signals to be ignored by the script and by invoked commands. Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The shell script called **scan** (below) illustrates **trap** usage where there is no exit in the **trap** command. The shell script **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the node until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands, but cause termination when **scan** is waiting for input.

```
#!/bin/sh
d=`pwd`
for i in *
do
   if test -d $d/$i
   then cd $d/$i
      while echo "$i:"
            trap exit 2
            read x
      do trap : 2; eval $x;
   done
   fi
done
```

The **read** command is a built-in command that reads one line from the standard input and places the result in the variable **x**. The command returns a nonzero exit status if an end-of-file is read or an interrupt is received.

Command Execution

To run a command other than a built-in command, the shell first creates a new program level in the shell process. The execution environment for the command includes input, output, and the states of signals, and is established before the command is executed. A built-in command **exec** creates a new program level in the shell process. For example, a simple version of the **nohup** command looks like this:

```
trap '' 1 2 3 15
exec $*
```

The **trap** command turns off the signals specified so they are ignored by subsequently created commands; **exec** runs the specified command as a new program level in the shell process itself.

Most forms of input/output redirection have already been described. In all of the following examples, *word* is only subject to parameter and command substitution. No filename generation or blank interpretation takes place; thus,

```
echo ... > *.c
```

writes its output into a file whose name is *.c. Input output specifications are evaluated left to right as they appear in the command.

> *file*	The standard output (file descriptor 1) is sent to *file*, which is created if it doesn't already exist.
>> *file*	The standard output is sent to *file*. If the file exists, output is added to the end; otherwise, the file is created.
< *file*	The standard input (file descriptor 0) is taken from the *file*.

<< *file*	The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *file*. If *file* is quoted, no interpretation of the document occurs. If *file* isn't quoted, parameter and command substitution occur and a backslash (\) is used to quote the characters \ $ ‘ and the first character of *word*. In the latter case, *newline* is ignored (c.f. quoted strings).
>&*digit*	The file descriptor *digit* is duplicated using the system call **dup** (duplicate a descriptor), and the result is used as standard output.
<&*digit*	The standard input is duplicated from file descriptor *digit*.
<&-	The standard input is closed.
>&-	The standard output is closed.

If any of the above are preceded by a digit, the file descriptor created is that specified by the digit instead of the default 0 or 1. For example, this runs *command* with error output (file descriptor 2) redirected to *file*:

command ... 2> file

and this runs *command* with its standard output and message output merged:

command ... 2>&1

The environment for a command run in the background such as

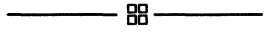
$ list *.c | lpr &

is modified in two ways. First, the default standard input for such a command is the empty file **/dev/null**. This prevents two parallel processes (the shell and the command) from trying to read the same input (a rather chaotic situation).

For example, the following allows both the editor and the shell to read from the same input at the same time:

```
$ ed file &
```

The environment of a background command is further modified by turning off the quit and interrupt signals so that they are ignored by the command. Thus, by convention, a signal set to 1(ignored) is never changed, even for a short time. Note also that the shell command **trap** has no effect on an ignored signal.



Chapter 10

Using the Korn Shell

The Korn shell, **ksh**, is a suitable for the Bourne shell and the C shell. It offers new features, better performance, and is upwardly compatible with the Bourne shell. It also provides many of the additional interactive features of the C shell. Most of the known bugs of the Bourne shell have been eliminated. Furthermore, **ksh** provides an enhanced programming environment so that users can write medium-sized programming tasks at the shell level without serious performance penalties. In most cases, scripts written for the Bourne shell can run without change under **ksh**.

The description of features in this chapter assumes that you are already familiar with the Bourne shell. If you do not have a solid understanding of how the Bourne shell works, please read Chapter 9. For further information on how to use the Korn shell, see **ksh** (Korn shell) in the *BSD Command Reference*.

Starting the Korn Shell

To start a Korn shell on an Apollo node after you've logged in, type the DM command

```
Command: cp /bin/ksh
```

In the case of the line above, **/bin** resolves to **/$(SYSTYPE)/bin**. See Chapters 2 and 3 for more information about the SYSTYPE environment variable.

The DM opens a window and runs the Korn shell in it. With the **ksh** command, you may supply the coordinates where the DM will locate the upper left and lower right corners of the window. You may even give the process a name, as in this line:

```
Command: (0,200)dr; (540,600)cp /bin/ksh
```

This command line opens up a small window near the left side of the screen.

Opening a Korn Shell When You Log In

The Bourne shell is the default UNIX shell in the BSD environment. Chapter 3 describes how the system determines which shell to run when you log in. You may arrange to have the system open a Korn shell as your log-in shell by specifying **/bin/ksh** in the shell field of your registry account, using the **chsh** (change shell) command.

When you log in to an Apollo node, the DM (Display Manager) looks in several places for information about what windows to open and what processes to start (see Chapter 3 for more detailed information). It normally opens a specified log-in shell, then looks for the file

~/user_data/startup_dm.*display_type*

The *display_type* argument matches the type of display in use (e.g., 1280bw). If you include a command line such as

(0,200)dr; (540,600) cp /bsd4.3/bin/ksh

in your **startup_dm** file, the DM automatically opens a BSD version of the Korn shell when you log in.

The log–in shell sets the working directory to your home directory and begins reading commands from the file named **.profile** in this directory. The log–in shell assumes that any file called **.profile** in your home directory contains commands, and reads it first, before reading commands from the terminal or any other file. Every Korn shell you start as a log–in shell reads from this file.

You may also define a key or function key to open a Korn shell. The following DM command defines the shifted L5 key (labeled <SHELL>) so that pressing SHIFT/<SHELL> opens a Korn shell:

```
kd l5s cp /bin/ksh ke
```

By default, <SHELL> starts up a new pad with your log–in shell, so if it is **ksh**, you don't need to do the above procedure.

Shell Variables

The ability to define and use variables to store and retrieve values is an important feature in most programming languages. The Korn shell has variables with identifiers that follow the same rules as the Bourne shell. Since all variables have string representations, there is no need to specify the type of each variable in the shell.

In the Korn shell, each variable can have one or more attributes that control the internal representation of the variable, the way the variable is printed, and its access or scope. Two of the attributes, **readonly** and **export**, are available in the Bourne shell. The **typeset** built–in command of **ksh** assigns attributes to variables. The complete list of attributes can be found under **ksh** in the *BSD Command Reference*. The **unset** built–in command of the **ksh** removes values and attributes of parameters.

Whenever a value is assigned to a variable, the value is transformed according to the attributes of the variable. Changing the attribute of a variable can change its value. There are three attributes for field justification, as might be needed for formatting a report.

For each of these attributes, the first time an assignment is made to the variable, its size is remembered. Each assignment causes justification of the field, truncating if necessary. Assignment to fixed

sized variables provides a simple way to generate a substring consisting of a fixed number of characters from the beginning or end of a string.

The attributes **-u** and **-l**, are used for uppercase and lowercase formatting respectively. Since it makes no sense to have both attributes on simultaneously, turning on either of these attributes turns the other off. The following script provides an example of the use of shell variables with attributes. This script reads a file of lines each consisting of five fields separated by a colon (:) and prints fields 4 and 2 in uppercase in columns 1–15, left justified, and columns 20–25 right-justified respectively.

```
typeset -Lu f4=123456789012345  # 15 character left justified
typeset -Ru f2=123456           # 6 character right justified
IFS=:
set -f                          # skip file name generation
while  read -r f1 f2 f3 f4 f5   # read line, split into fields
do print -r "$f4 $f2"           # print fields 4 and 2
done
```

The integer attribute, **-i**, causes the variable to be internally represented as an integer. The first assignment to an integer variable determines the output base (see the following paragraphs). This base will be used whenever the variable is printed. Assignment to integer typed variables result in arithmetic evaluation, as described below, of the right hand side.

The Korn shell allows 1–dimensional arrays in addition to simple variables. Any variable can become an array by referring to it with a subscript. All elements of an array need not exist. Subscripts for arrays must evaluate to an integer between 0 and 127, otherwise an error results. Evaluation of subscripts is described in the next section. Attributes apply to the whole array.

Assignments to array variables are made with the typeset built-in command. Referencing of subscripted variables requires a dollar sign ($), but also requires braces around the array element name. The braces are needed to avoid conflicts with the file name generation mechanism. The form of any array element reference is:

${*name*[*subscript*]**}**

A subscript value of asterisk (*) or at sign (@) can be used to generate all elements of an array, as they are used for expansion of positional parameters.

A few additional operations are available on shell variables. A

${#*name*}

is the length in bytes of $name. For an array variable, the following gives the number of elements in the array:

${#*name*[*]}

There are two parameter substitution modifiers that have been added to strip off leading and trailing substrings during parameter substitution. The pound sign (#) modifier strips off from the left, and the percent sign (%) modifier strips off from the right. For example, if the shell variable **i** has value **file.c**, then the expression

${i%.c}

has value **file**. The **substring** built-in command has been added to **ksh** to enable the user to generate a substring of a given string. This built-in allows the user to specify an expression to be deleted from the left and right ends of the string. The resulting substring is printed out. Command substitution can be used to assign the output of substring to a shell variable.

Arithmetic Evaluation

The built-in command, **let**, provides the ability to do integer arithmetic. All arithmetic evaluations are performed using long arithmetic. Arithmetic constants are written as

base#number

The *base* argument is a decimal integer between 2 and 36; *number* is any positive integer. Base ten is used if no base is specified.

Arithmetic expressions are made from constants, variables, and one or more of the 14 operators listed in the manual page. Operators are evaluated in order of precedence. Parentheses may be used for grouping. A variable does not have to have an integer attribute to be used within an arithmetic expression. The name of the variable is replaced by its value within an arithmetic expression. The statement

let x=x+1

can be used to increment a variable **x**. Note that there is no space before or after the plus (+) and equal (=) operators. This is because each argument to **let** is an expression to evaluate. The last expression determines the value returned by **let**. If the last expression evaluates to a nonzero value, the **let** returns true. Otherwise, **let** returns false.

Note that many of the arithmetic operators have special meaning to the shell and must be quoted. Since this can be burdensome, an alternate form of arithmetic evaluation syntax has been provided. For any command that begins with double left parentheses, all the characters until the matching double right parentheses are treated as a quoted arithmetic expression. The double parentheses usually avoids incompatibility with the Bourne shell's use of parentheses for grouping a set of commands to be run in a subshell. Expressions inside double parentheses can contain blanks and special characters without quoting. More precisely,

((...))

is equivalent to

let " ... "

The following script prints the first *n* lines of its standard input onto its standard output, where *n* is supplied as an argument or is 20 if omitted:

```
typeset -i n=${1-20}                          # set n
while read -r line && (( (n=n-1)>=0 ))        # at most n lines
do print -r - "$line"
done
```

Functions and Command Aliasing

Two new mechanisms help create pseudo-commands (things that look like commands, but do not always create a process). The first technique is called **command name aliasing.**

As a command is being read, the command name is checked against a list of alias names. If it is found, the name is replaced by the text associated with the alias and then rescanned. The text of an alias is not checked for aliases, so recursive definitions are not allowed.

Aliases are defined with the **alias** built-in command. The form of an alias command is:

alias *name=value*

Except for the first character, which must be printable, the alias *name* must be a valid identifier. The replacement text, *value*, can contain any valid shell script, including metacharacters such as pipe symbols and input/output redirection. Aliases can be used to redefine built-in commands so that the alias

alias test=./test

can be used to look for **test** in your current working directory rather than using the built-in **test** command. Keywords such as **for** and **while** cannot be changed by aliasing. The command **alias**, without arguments, generates a list of aliases and corresponding texts. The **unalias** command removes the name and text of an alias.

Aliases are used to save typing and to improve readability of scripts. For example, the alias

alias integer='typeset -i'

allows **integer** the variables **i** and **j** to be declared and initialized with the command integer

i=0 j=1

One frequent use of aliases is to alias a command name to the full pathname of the program. This eliminates the path search but requires knowledge of where that program will be stored. To reduce the amount of path searching, tracked aliases have been introduced. A tracked alias is not given a value. Its value is defined at the first reference by a path-search as the full pathname equivalent of the name, and remains defined until the **PATH** variable is changed. Programs found in directories that do not begin with a slash (/) that occur earlier in the path-search than the value of the tracked alias, take precedence over tracked aliases.

Tracked aliases provide an alternative to the **csh** (C shell) command hashing facility. Tracked aliases do not require time for initialization and allow for new commands to be introduced without the need for rehashing. An option to the shell allows all command names that are valid alias names to become tracked aliases.

Functions are more general than aliases but also more costly. Functions definitions are of the form

function *name*
{
    any shell script
}

The **function** is invoked by writing *name* and optionally following it with arguments. Positional parameters are saved before each function call and restored when completed. Functions are executed in the current shell environment and can share named variables with the calling program. The **return** built-in can be used to cause the function to return to the statement following the point of invocation.

By default, variables are inherited by the function and shared by the calling program. However, environment substitutions preceding the function call apply only to the scope of the function call. Also, variables defined with the typeset, built-in command are local to the function in which they are declared. Thus, for the function defined as follows

```
function name
{
    typeset -i x=10
    let z=x+y
    print $z
}
```

and invoked as

```
y=13 name
```

x and y are local variables with respect to the function name while z is global.

Alias and function names are never directly carried across separate invocations of **ksh**, but can be passed down to subshells. The **-x** flag is used with **alias** to carry aliases to subshells while the **-fx** flags of **typeset** are used to do the same for functions.

Several UNIX commands can be aliased to **ksh** built-in commands. Some of these are automatically set each time the shell is invoked. About 20 frequently used UNIX commands are also set as tracked aliases. Each user can create a file for aliases and functions.

The location of an **alias** command can be important since aliases are only processed when a command is read. A **.procedure** is read all at once (unlike **profiles**, read a command at a time) so that aliases defined there won't affect commands within this script.

A name is checked to see if it is a built-in command before checking to see if it is a function. To write a function to replace a built-in command you must define a function with a different name and alias the built-in name to this function. For example, to write a **cd** function which changes the directory and prints out the directory name, you can write the following:

```
alias cd=_cd
function _cd
{
    if      'cd' "$@"
    then    echo $PWD
    fi
}
```

The single quotes around **cd** within the function prevents alias substitution. The **PWD** variable is described below. The combination of aliases and functions can be used to do things that can't be done with either of these separately. For example, the function and aliases defined as

```
function _from # i=start to finish [ by incr]
{
          typeset var=${1%%=*}
         integer incr=${5-1} $1
         while   (( $var <= $3 ))
        do       _repeat
                 let $var=$var+incr
        done
}
alias repeat='function _repeat {'  from='}; _from'
```

allow you to write loops such as the following with the expected behavior:

```
repeat
        any script command
from i=1 to 13 by 3
```

You should put aliases and functions that are to be available for all shell invocations into your **ksh** startup file. By setting and exporting the environment variable, **ENV**, to the name of this file, the aliases and functions are defined each time **ksh** is invoked. The value of the **ENV** variable undergoes macro and command substitution prior to its use. Since the **ENV** file is not invoked automatically for a log in shell, you must include the following lines in your **.profile** file:

```
ENV=~/.kshrc
eval $ENV
```

See Chapter 7 for further information on the **ENV** environment variable.

Input and Output

An extended I/O capability enhances the use of the shell as a programming language. The Bourne shell has a built-in command (**read**) for reading lines from file descriptor 0, but does not have any internal output mechanism. As a result, the **echo** command has been used to produce output for a shell procedure. This is inefficient and restrictive.

For example, you cannot read in a line from a terminal and echo the line exactly as is. In the Bourne shell, the **read** built-in command cannot be used to read lines that end in a backslash (\), and the **echo** (echo arguments) command will treat certain sequences as control sequences. In addition, you can never have more than one file open at any time for reading.

The Korn shell has options on the **read** command to specify the file descriptor for the input. The **exec** built-in command can be used to open and close file streams. The **-r** option allows a backslash (\) at the end of an input line to be treated as a regular character rather than the line continuation character. The first argument of the **read** command can be followed by a question mark (?) and a prompt to produce a prompt at the terminal before the read. If the input is not from a terminal device then the prompt is not issued.

The **ksh** built-in command, **print**, outputs characters to the terminal or to a file. You you can specify the file descriptor number as an option to the command. Ordinarily, the arguments to this command are processed the same as for **echo**. However, the **-r** flag can be used to output the arguments without any special meaning. The **-n** flag can be used here to suppress the trailing newline that is ordinarily appended.

To improve performance of existing shell programs, an alias for **echo** is defined by the shell when it is invoked. For the BSD version, the alias is

alias echo='print -'

where the dash (-) signifies that there are no more options.

The shell is frequently used as a programming language for interactive dialogues. The **select** statement has been added to the language to make it easier to present menu selection alternatives to the user and evaluate the reply. The list of alternatives is numbered and put in columns. A user-settable prompt, **PS3**, is issued and if the answer is a number corresponding to one of the alternatives, the select loop variable is set to this value. In any case, the **REPLY** variable stores the user-entered reply.

Re-entering Commands

An interactive shell saves the commands you type at your node in a file. If the variable **HISTFILE** is set to the name of a file to which you have write access, then the commands are stored in this history file. Otherwise, the file

$HOME/.history

is checked for write access, and if this fails, an unnamed file holds the history lines. This file is truncated if this is a top level shell. The number of commands accessible to you is determined by the value of the **HISTSIZE** variable at the time the shell is invoked. The default value is 64.

A command may consist of one or more lines since a compound command is considered one command. If an exclamation point (!) is placed within the primary prompt string, it is replaced by the command number each time the prompt is given. Whenever the file is named, all shells that use this file share access to the same history.

A built-in command, **fc** (fix command), lists and/or edits any of these saved commands. The command can always be specified with a range of one or more commands. The range can be specified by giving the command number, relative or absolute, or by giving the first character or characters of the command. The **-l** option specifies listing of previous commands. When given without specifying the range, the last 16 commands are listed, each preceded by the command number.

If the listing option is not selected, then the range of commands specified, or the last command if no range is given, is passed to an editor program before being re-executed by **ksh**. The editor to be used may be specified with the **-e** option, followed by the editor name. If this option is not specified, the value of the shell variable **FCEDIT** is used as the name of the editor, providing that this variable has non-null value. If this variable is not set, or is null, and you have not selected the **-e** option, **/bin/ed** is used. When editing is complete, the edited text automatically becomes the input for **ksh**. As this text is read by **ksh**, it is echoed onto your node.

Using a dash (-) in place of an editor name bypasses the editing and simply re-executes the command. Here, you can specify only a single command as the range, and then add an optional argument of the form

old=new

This requests a simple string substitution prior to evaluation. A convenient alias,

alias r='fc -e -'

has been predefined so that the single keystroke **r** can be used to re-execute the previous command, and the key-stroke sequence

r abc=def c

can be used to re-execute the last command that starts with the letter **c** with the first occurrence of the string abc replaced with the string def. Typing

r c > file

re-executes the most recent command starting with the letter **c**, with standard output redirected to file.

In-line Editing

The Korn shell offers options that let you edit parts of the current command line before submitting the command. The in-line edit options make the command line into a single line screen edit window.

When the command is longer than the width of the terminal, only a portion of the command is visible. Moving within the line automatically makes that portion visible. Editing can be performed on this window until the return key is pressed. The editing modes have commands that access the history file in which previous commands are saved. You can copy any of the most recent **HISTSIZE** commands from this file into the input edit window. You can locate commands by searching or by position.

The in-line editing options do not use the termcap database. They work on most standard terminals. They only require that the backspace character moves the cursor left and the space character overwrites the current character on the screen.

You have a choice of editor options. You can open a DM edit window in your current pad by setting the value of the **FCEDIT** variable to **pad**, and unsetting the **VISUAL** and **EDITOR** variables. In VT100 windows or on dialup lines, you can select the **emacs**, **gmacs**, or **vi** option by turning on the corresponding option of the **set** command. If the value of the **EDITOR** or **VISUAL** variables ends with any of these suffixes, the corresponding options are turned on. A large subset of each of each of these editors features are available within the shell. Additional functions, such as file name completion, are also available.

In the **emacs** or **gmacs** mode, you position the cursor to the point needing correction and insert, delete, or replace characters as needed. The only difference between these two modes is the meaning of the command CTRL/T. Control keys and escape sequences are used for cursor positioning and control functions. The available editing functions are listed in the manual page.

The **vi** (visual display editor) editing mode starts in insert mode and enters control mode when you type <ESC>. A <RETURN>, which submits the current command for processing, can be entered from either mode. The cursor can be anywhere on the line. A subset of

commonly used **vi** commands are available. The **k** and **j** command that normally move up and down by one line, move up and down one command in the history file, copying the command **at** into the input edit window.

For efficiency, the node is kept in canonical mode until an <ESC> is typed. On some terminals, and on earlier versions of UNIX operating systems, this doesn't work correctly. The **viraw** option of the **set** command, which always uses raw or cbreak mode, must be used in this case.

Job Control

The job control feature lets you stop and restart programs, and to move programs to and from the foreground and the background.

An interactive shell associates a job with each pipeline typed in from the terminal and assigns them a small integer number called the job number. If the job is run asynchronously, the job number is printed at your node. At any given time, only one job owns the node, i.e., keyboard signals are only sent to the processes in one job.

When **ksh** creates a foreground job, it gives it ownership of the node. If you are running a job and wish to stop it, you hit CTRL/Z, which sends a stop signal to all processes in the current job. The shell is notified the processes have stopped and takes back control of the node.

Commands to continue programs in the foreground and background are available. There are also several ways to refer to jobs. A percent sign (%) introduces a job name. You can refer to jobs by name or number as described in the manual page. The built-in command **bg** allows you to continue a job in the background, while the built-in command **fg** allows you to continue a job in the foreground even though you may have started it in the background.

A job being run in the background stops if it tries to read from the node. It is also possible to stop background jobs that try to write on the node by setting terminal options appropriately.

A built-in command, **jobs**, that lists the status of all running and stopped jobs. In addition, you are notified of the change of state of any background jobs just before each prompt. When you try to leave the shell while jobs are stopped or running, you receive a message from **ksh**. If you ignore this message and try to leave again, all stopped processes are terminated.

A built-in version of **kill** makes it possible to use job numbers as targets for signals. Signals can be selected by number or name. The name of the signal is the name found in the **/usr/include/signal.h** include file with the prefix SIG removed. The list of valid signal names can be generated with the **-l** flag of **kill**.

Miscellaneous

The Korn shell has several additional features to enhance functionality and performance. This section lists most of these features.

Tilde Substitution

The tilde (~) character at the beginning of a word has special meaning to **ksh**. If the characters after the tilde up to a slash (/) or the end of a parameter match a user login name in the **/etc/passwd** file, then the tilde and the name are replaced by that user's login directory. If no match is found, the original word is unchanged.

A tilde by itself, or in front of a slash, is replaced by the value of the **HOME** parameter. A tilde followed by a plus (+) or minus (-) sign is replaced by the value of the parameter **PWD** and **OLDPWD** respectively. Tilde substitution takes place when the script is read, not while it is executed.

Built-in I/O Redirection

All built-in commands can be redirected.

Added Options

Several options have been added to the shell and all options have names that can be used in place of flags for setting and resetting options. The following command lists current option settings:

set -o

The **-f** option or **noglob** disables filename generation. It can be applied at invocation or as an option to the **set** command.

The option **ignoreeof** can be used in a top-level shell to prevent CTRL/D from logging you out. You must type "exit" to log out.

The **-h** or **trackall** option causes all commands whose name is a valid alias name to become a tracked alias.

The **job monitor** option causes a report to be printed when each background job completes. It is automatically enabled on Apollo nodes.

The **bgnice** option causes background jobs to run at lower priority.

Previous Directory

The Korn shell remembers your last directory in the variable **OLDPWD.** The **cd** built-in command can be given, followed by a dash (-), to return to the previous directory. Note that, if you do this two times in a row, you return to the starting directory, not the second previous directory. A directory stack can be implemented with shell internals by using an array and writing shell functions to push and pop directories from the stack.

Additional Variables and Parameters

Several new parameters have special meaning to **ksh**. The variable **PWD** holds the current working directory of the shell. The command, **pwd**, is aliased to print the following for better response:

```
- $PWD
```

The variable **OLDPWD** holds the previous working directory of the shell.

The variable **FCEDIT** is used by the **fc** built-in described above. The variables **VISUAL** and **EDITOR** are used for determining the edit modes as described above.

> **NOTE:** Since editing in DM transcript pads is done in cooked mode, it is awkward to use the **ksh** built-in command line editing. A good solution to the problem is to set **FCEDIT=pad** and unset the **VISUAL** and **EDITOR** variables.

The variable **ENV** defines the startup file for non-login **ksh** invocations. On Apollo systems, variables may be set on the command line. This is encouraged for opening new pads. For example,

```
Command: cp /bin/ksh -DENV=~/.kshrc.pad
```

will run the start-up script **.kshrc.pad** from the user's home directory in the new pad.

The variables **HISTSIZE** and **HISTFILE** control the size and location of the file containing commands entered at a terminal.

The parameter **MAILPATH** is a colon (:) separated list of filenames to be checked for changes periodically. You are notified before the next prompt. Each of the names in this list can be followed by a question mark (?) and a prompt to be given when a change has been detected in the file. The prompt is evaluated for macro and command substitution. The parameter **MAILCHECK** specifies the minimal interval in seconds before the system checks for new mail.

The variable **RANDOM** produces a random number each time it is referenced. Assignment to this variable sets the seed for the random number generator.

The parameter **PPID** generates the parent process id of the shell.

The value of the parameter represented by an underscore (_) is the last argument of the previous command.

The parameter **SECONDS** represents the number of seconds since shell invocation is returned. If given a value, then the value returned upon reference will be the assigned value plus the number of seconds since the assignment.

The parameter **TMOUT** can be set to be the number of seconds that the shell will wait for input before exiting.

The **COLS** variable can be used to adjust the width of the edit window for the in-line edit modes.

Modified Variables

The input field separator parameter, **IFS**, has meaning for the read built-in, for the set built-in, and while expanding for and select lists. In all other instances it is ignored.

Timing Commands

A keyword time has been added to replace the time command. Any function, command or pipeline can be preceded by this keyword to obtain information about the elapsed, user and system times. Since I/O redirection binds to the command, not to time, parenthesis should be used to redirect the timing information which is normally printed on file descriptor 2.

Command Substitution

Command substitution in the Bourne shell suffers from some complicated quoting rules. It is hard to write a **sed** pattern which contains backslashes within command substitution. Putting the pattern in single quotes is not very helpful.

The Korn shell leaves the Bourne shell command substitution alone and adds a newer and easier to use command substitution syntax. All characters between a $(and a matching) are evaluated as a command. The dollar sign means "value of" and the parentheses denote a command.

The command itself can contain quoted strings even if the substitution occurs within double quotes. Nesting is also legal. You can use unbalanced parentheses within the command, providing that they are quoted. The special command substitution of the form **$(cat file)** can be replaced by **$(< file)**, which is faster because no separate process is created.

Whence

The addition of aliases, functions, and more built-ins has made it substantially more difficult to know what a given command word really means. A built-in command, **whence**, when used with the **-v** option, has been provided to answer this question. A line is printed for each argument to **whence**, telling what would happen if this argument were used as a command name. It reports on keywords, aliases, built-ins, and functions. If the command is none of the above, it follows the path search rules and prints the full pathname, if any; otherwise, it prints an error message.

Added Traps

All traps can be given by name in the Korn shell. The names of traps corresponding to signals are the same as the signal name with the **SIG** prefix removed. The trap 0 is named **EXIT** and a new trap named **ERR** has been added. This trap is invoked whenever the shell would exit if the **-e** flag were set.

Additional Test Operators

The operators **-ot** and **-nt** can compare the modification times of two files to see which file is older than or newer than the other. The operator **-ef** checks to see if two files have the same device and i-node number, i. e., a link to the same file.

No Special Meaning for Circumflex (^)

The Bourne shell uses a circumflex (^) as an archaic synonym for the pipe character (|). The circumflex is not a special character to the Korn Shell.

Performance

The Korn shell executes many scripts faster than other Bourne shells. One major reason is that many of the functions provided by the **echo** and **expr** commands are built-in. The time to execute a built-in function is one or two orders of magnitude faster than performing a fork and execute of the shell. Command substitution of built-ins is performed without creating another process, and often without even creating a temporary file.

Another reason for improved performance is that all I/O is buffered. Output buffers are flushed only when required. Several of the internal algorithms have been changed so that the number of subroutine calls has been substantially reduced. The Korn shell uses hash tables for variables. Scripts that rely heavily on referencing variables execute faster. More processing is performed while reading the script so that execution time is saved while running loops.

Scripts that do little internal processing and create many processes run a little slower because the time to fork **ksh** is slower than for the Bourne shell.

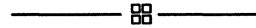
Sample Korn Shell Script

An example of a **ksh** script is shown below. This program is a variant of the **grep** (search file for pattern) program. Pattern matching for this version of **grep** means shell patterns consisting of question mark (?), asterisk (*), and left and right brackets ([ ]).

The first half examines option flags. Note that all options except **-b** have been implemented. The second half goes through each line of each file to look for a pattern match.

This program does not serve as a replacement for **grep**, only an illustration of the programming power of **ksh**. Note that no auxiliary processes are spawned by this script. It was written and debugged in under two hours. While performance is acceptable for small programs, this program runs at only one-tenth the speed of **grep** for large files.

```
# SHELL VERSION OF GREP
vflag= xflag= cflag= lflag= nflag=
set -f
while((1))                          # look for grep options
do  case"   $1" in
    -v*)    vflag=1;;
    -x*)    xflag=1;;
    -c*)    cflag=1;;
    -l*)    lflag=1;;
    -n*)    nflag=1;;
    -b*)    print 'b option not supported';;
    -e*)    shift;expr="$1";;
    -f*)    shift;expr`< $`;;
    -*) print $0: 'unknown flag';exit 2;;
    *)  if  test "$expr" = ''
        then    expr="$1";shift
        fi
        test "$xflag" || expr="*${expr}*"
        break;;
    esac
    shift                           # next argument
done
noprint=$vflag$cflag$lflag  # don't print if these flags set
integer n=0 c=0 tc=0 nargs=$# # initialize counters
for i in "$@"                       # go thru the files
do  if  ((nargs<=1))
    then    fname=''
    else    fname="$i":
    fi
    test "$i" && exec 0< $i     # open file if necessary
    while   read -r line        # read in a line
    do  let n=n+1
        case"$line" in
        $expr)                  # line matches pattern
            if      test "$noprint" = ""
            then    print -r "$fname${nflag:+$n:}$line"
            fi
            let c=c+1 ;;
        *)                      # not a match
            if      test "$vflag"
            then    print -r "$fname${nflag:+$n:}$line"
            fi;;
        esac
done
if      test "$lflag" && ((c))
then    print - $i
fi
let tc=tc+c n=0 c=0
done
test "$cflag" && print $tc      # print count if cflag is set
let tc                          # set the exit value
```



Chapter 11

Managing Files

In Chapter 1, we looked at how the system organizes objects (files, directories, and links) in a structure called a naming tree. This chapter describes how to use shell commands to manage these objects on your system. Shell commands let you move around the system naming tree and create, rename, copy, move, print, delete, and compare objects.

Since all of the commands described in this chapter require you to specify pathnames, you should understand the rules for pathnames described in Chapter 1. Commands that use the shell command line parser also allow you to perform operations on groups of objects, and therefore accept one or more pathname wildcards. Many of the examples in this chapter show you how to use pathname wildcards in specific operations. For a complete description of the pathname wildcards you can use with shell commands, refer to the shell chapters (Chapters 8, 9, and 10).

Keep in mind that this chapter describes the *basic* functions of the commands you use to manage objects. For a complete description of a particular command and *all* of its options, refer to the *BSD Command Reference*.

Moving Around the Naming Tree

Most of the commands described in this chapter require you to use pathnames to specify locations in the naming tree where you want particular operations performed. Often, you will specify pathnames that use the current working directory. To move around the naming tree, you need to know how to change your working directory.

The working directory is where the system begins its search for objects when you omit the object's full pathname. At log–in, the DM and any log–in shell sets your initial working directory to the home directory designated in your user account (see Chapter 3). Each subsequent process that you create uses the working directory of its parent process as its working directory.

To display the name of a process's current working directory, specify the **pwd** (print working directory) command as follows:

```
% pwd
```

To change a process's working directory to another directory, specify the **cd** (change directory) command in the following format:

```
% cd dir
```

The *dir* argument specifies the pathname of the directory you want to use as the working directory. For example:

```
% cd //my_node/owner/forms
```

sets the working directory for the current process to **forms**. Once set, any time you omit the full pathname of an object, the system starts its search at the directory **forms** by default.

Creating Files

To create normal text files, you may use one of the UNIX text editors, such as **vi** (visual display editor) or **ex** (text editor), or you

may specify the DM command **ce** (create edit) along with the pathname of the file you want to create. The *BSD Command Reference* and *UNIX Text Processing* describes how to use the UNIX text editors. This section describes how to use the DM to create files.

By default, <EDIT> invokes the **ce** command. The **ce** command directs the DM to create the file and open an edit pad and window for the file on the display. Using the DM editor, you can edit the file, then save its contents by pressing <EXIT>. When you save the file, the system stores it at the location in the naming tree specified by the file's pathname. Refer to the "Creating Pads and Windows" section in Chapter 4 for a description of how to use the **ce** command to create and edit files.

The following example creates a file named **memo** in the directory **/user**, and opens the file for editing:

```
Command: ce //node/user/memo
```

The previous example uses an absolute pathname to specify the name of the new file. When you use a pathname that assumes the current working directory, the system uses the working directory of the current process. (The last process to perform an operation before you specified the **ce** command is the current process.)

The following example creates a file named **memo** in the current working directory:

```
Command: ce memo
```

The command in this example specifies the filename **memo**. Since the pathname does not specify an explicit directory location, the system uses the current process's working directory to determine where to create the new file. If the current process's working directory is **//node/user**, then the system will create the new file with the pathname **//node/user/memo**.

When you run multiple shell processes, you typically move between processes, often changing the current working directory. As a result, you may find it difficult to keep track of the current working directory. In situations where you run multiple processes, you may want to specify absolute pathnames to avoid creating files at an unintended location.

When you create a file, the system assigns the file a set of default ACLs by copying the initial file ACL from the file's parent directory. After you create a file you can change its ACLs with the **chacl** (change ACL) or **dbacl** (Domain/Dialogue-based ACL editor) command. Chapter 14 explains ACLs and shows how to use these commands.

Copying Files

When you copy a file, you create a copy of the file at another location in the naming tree. To copy a file or group of files, use the **cp** (copy) command in the following format:

```
% cp [options] source target
```

The *source* argument specifies the pathname of the file you want to copy, and *target* specifies the pathname of the naming tree location where you want the copy created. The rules for pathnames described in Chapter 1 apply to both command arguments.

The **cp** command always creates a copy of the source file at the location specified by the target. For example:

```
% cp memo /user_1/new_memo
```

creates a copy of the source file **memo** in the directory **user_1**. In this example, since the target specifies the pathname of a file, **cp** assigns the copy the name specified by the target, **new_memo**.

If the target specifies the pathname of a directory, **cp** creates a copy of the source file in the target directory (the current working directory if you omit the target) and assigns the copy the filename of the source file. For example, the following command line

```
% cp memo /user_1
```

copies the file **memo** from the current working directory to the target directory **user_1**. Because **cp** assigns the copy the name of the source file, the new file has the pathname **/user_1/memo**.

By default, the system assigns the target file the default file protections of its parent directory (see Chapter 12). So, in the previous example, the system assigns the target file the default file protections of the directory **user_1**.

Moving or Renaming Files

When you move a file, you relocate the file in the naming tree. This may involve simply changing the file's name, or may result in the file being copied to another location and the original being deleted. The latter happens when moving files between physical disks.

To move a file or group of files from one location in the naming tree to another, use the **mv** (move) command in the format:

mv [*options*] *source target*

The *source* argument specifies the pathname of the file you want to move and *target* specifies the pathname of the file's new location in the naming tree. The rules for pathnames described in Chapter 1 apply to both arguments.

The following command moves the file **floorplan**:

```
% mv /designer/floorplan /builder/plans/cape
```

In this example, the target specifies the pathname for a nonexistent file named **cape**. The **mv** command moves the file **floorplan** from the directory **designer** to the directory **builder/plans** and names the file **cape**.

If the target pathname specifies a directory, **mv** moves the source file into the target directory. For example, the following command moves the file **floorplan** into the directory **builder**:

```
% mv /designer/floorplan /builder
```

In this example, since no target filename was specified, the file retains the name of the source file (**floorplan**).

Printing Files

The standard command for printing files is **lpr** (off-line printer), which takes arguments in the format

lpr [*options*] [*file...*]

The *file* argument indicates which file(s) are to be printed.

To send output to a specific printer, use the **-P** option followed immediately by the printer name. For example, to print the file **gyre** on the printer named **spin**, type the command

```
% lpr -Pspin gyre
```

Notice that there is no space between the **-P** flag and the name of the printer. If **-P** is not specified, **lpr** will use the default printer for your site. You can also set your own default printer by setting the **PRINTER** environment variable.

The **lpr** command copies files into the **/usr/spool** directory, where they remain until printed by a daemon process. To examine the status of printing jobs waiting in this queue, use the **lpq** (spool queue examination program) command in the following format:

lpq [*options*] [*users*]

The most commonly used option is **-P**, which is followed by the name of a printer (as with **lpr**). This option causes **lpq** to report only on jobs associated with the specified printer. If no options are given, **lpq** reports on jobs queued to the default printer.

You may also restrict **lpq** to reporting jobs owned by specific users. To do this, include the name(s) of the desired user(s) at the end of the command line. For instance, to inquire about the status of all jobs owned by users **barbara** and **pat** awaiting printing on **spin**, you would enter the command line

```
% lpq -Pspin barbara pat
```

For more information on **lpq** (and for a discussion of **lprm**, which removes jobs from the queue), see the entries in the *BSD Command Reference*.

Using the prf Command

An alternative to using **lpr** is the **prf** (print file) command. To print one or more files, use the **prf** command in the following format:

prf [*file...*] [*options*]

The *file* argument specifies the name of the file you want to print. The following command uses shell wildcards to print any file in the current working directory that begins with **file** and ends with a one-digit number:

```
% prf file_[0-9]
```

This command, for example, prints **file_2** and **file_8** but not **file_a** or **file_b**.

To indicate which printer to print the file(s) on, use the **-pr[inter]** option followed by the name of the printer. The example below prints the file **sales_plan** on the printer named **spin**:

```
% prf sales_plan -pr spin
```

After you enter such a command line, **prf** will normally respond by displaying a message like the following:

```
//node/owner/sales_plan queued for printing at site
//print_site.
```

If you normally use a printer connected to your node, your queue file is named **/sys/print/queue**. If a remote node controls the printers that you use by default, then **/sys/print** is a link that your system administrator creates to point to the **/sys/print** directory on the remote node. This link causes **prf** to queue files to the **/sys/print/queue** directory on the remote node by default. (Chapter 13 describes links in more detail.)

You can queue a file to the **/sys/print/queue** directory on another node by specifying the **-s[ite]** option along with the name of the node's entry directory. For example,

```
% prf sales_plan -s //boston
```

queues **sales_plan** to the **/sys/print/queue** directory on the remote node **boston**.

To find out the names of printers available to you, use the **-list_printers** option in the following manner:

```
% prf -list_printers
```

The line above lists the names of all printers located at your local print site (i.e., the one to which your **/sys/print** points). To determine the names of printers available at a different site, specify the following command line, where *site* is the name of a known print site:

```
% prf -list_printers //site
```

Options may also be specified in a configuration file. (Creating such a file saves you the work of entering the same options each time you use **prf**.) For information on creating a configuration file, or for general information on **prf** and its command options, refer to the *BSD Command Reference* entry for **prf**.

Using the Print Menu Interface

The **prf** command also has an option for displaying a special print menu interface. This menu allows you to specify print arguments and select various options without having to type them on the command line. The print menu interface is useful when you routinely specify several print options for each file you print. By using the menu interface, you can select all of the options once, and print several files without respecifying the options for each file you print.

To print files using the print menu interface, specify the **-dia** option:

```
% prf -dia
```

As shown in Figure 11–1, this command creates a special window pane at the top of the shell process window. An arrow cursor appears in the upper left corner of the menu.

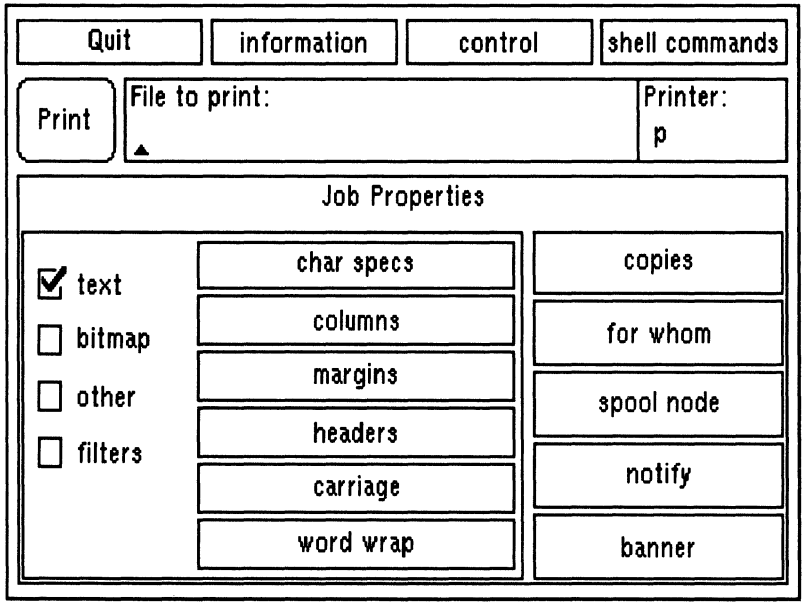


Figure 11–1. The Print Menu

The triangular cursor below the "File to print:" prompt indicates that characters typed at the keyboard will appear in this field. To print the file **sales_plan** in your working directory, first you must type its name followed by <RETURN>:

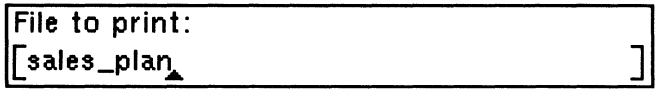


Figure 11–2. Specifying a Filename on the Print Menu

Note the square brackets which enclose the filename as you type. When present, these show that you've not yet pressed <RETURN>.

If you want to print a file from a directory other than the working directory, enter a full pathname like **//node/owner/jan_report**.

To select a different field, move the arrow cursor to the menu item you want to select and press <F1> or the left mouse button (M1). To display help information about an item, position the cursor over the item and press <HELP> or the right mouse button.

Most of the other items in the menu display submenus when you select them. These submenus allow you to select or specify additional print information. Table 11–1 describes the submenus for the **shell commands** menu item.

Table 11–1. Shell Commands Submenu Items

Item	Description
shell	Passes control to a temporary shell. When you select **shell**, a shell prompt appears in the process input window. To return control to the print menu, exit the shell.
set/inq working dir	Displays the name of the working directory. To change the working directory, enter a new directory pathname in the field.

Use submenus in the same way you use the main menu: either select an item, or type the requested information. When you finish with a submenu, move the arrow cursor out of the submenu box, and it will close. The value you entered will remain in effect until you change it again, or until you exit the program.

When you're satisfied with the selections you've made in the print menu, you can print the file by selecting **Print** with <F1> or the left mouse button. A message similar to the following appears on the transcript pad:

```
//node/owner/sales_plan queued for printing at site
//print_site.
```

After you print a file, the menu remains on the screen, enabling you to print additional files. To print another file using the same selections, specify a new filename and select **Print**; the print menu uses the submenu selections you already made. To exit the program, select **Quit**.

Most of the selections in the print menu perform the same print functions as options on the command line. For more information on a specific menu or submenu item, refer to the description of its related **prf** command option in the *BSD Command Reference*.

Displaying File Attributes

To display a file's attributes, such as its mode, number of links, owner, size in bytes, and time of last modification, use the **ls** (list directory) command in the following format:

ls -l *name*

The *name* argument specifies the pathname of the file or directory, and **-l** specifies that you want the attributes mentioned above displayed.

The following command line

% ls -l //node/user/memo

displays attribute information in long format for the file **memo**, as shown here:

```
-rwxrwxrwx 1 bob 660 Aug 3 16:58  memo ->//mad/doc
```

Note that the link resolution is also displayed for the file, showing that **memo** is linked to **//mad/doc**. For further details, see the *BSD Command Reference*.

Removing Files

To remove one or more files, use the **rm** (remove file) command in the following format:

rm [*options*] *file*

The *file* argument specifies the pathname of the file you want to delete. If you specify multiple pathnames to delete multiple files, separate each pathname with a space.

The following command deletes the files **my_plan** and **report** from the current working directory:

```
% rm my_plan report
```

You can also use pathname wildcards to delete related groups of files. For example:

```
% rm *.bak
```

The **.bak** expression causes **rm** to delete all of the files in the current working directory that end in **.bak**. For further details, see the *BSD Command Reference*.

Copying the Display to a File

You can copy the image of your current display to a file using the **cpscr** (copy screen) command in the following format:

cpscr [**-i**] *pathname*

The *pathname* argument specifies the pathname of the file to which you want to copy the display image. The **-i** option directs **cpscr** to store the file in reverse video (black on white or white on black depending on the current display setting).

To create a GPR bitmap file, use the **cpscr** command in the following format:

cpscr -b *pathname*

The **-b** option directs **cpscr** to create a gpr bitmap file (color screens are copied into a GPR bitmap file by default). To print a file that contains a screen image, use the **prf** command with the **-plot** option.

Comparing ASCII Files

To identify differences between ASCII text files, use the **diff** (show file differences) command in the following format:

diff [*options*] *file1 file2*

The *file1* and *file2* arguments specify the pathnames of the files to be compared; **diff** reports all differences in the files. If you specify a dash (-) in place of one of the file pathnames, **diff** compares the source with text from standard input.

The **diff** command in Figure 11-3 compares the contents of the file **speech.bak** to the contents of **speech**. The lines from **file1** are marked with a less than sign (<), while the lines from **file2** are marked with a greater-than character (>). The **diff** command prints the line numbers, and letters identifying the types of changes necessary to make the files identical. If **file2** is a directory, it looks for a file with the same name as **file1** in that directory. For further details, see the *BSD Command Reference*.

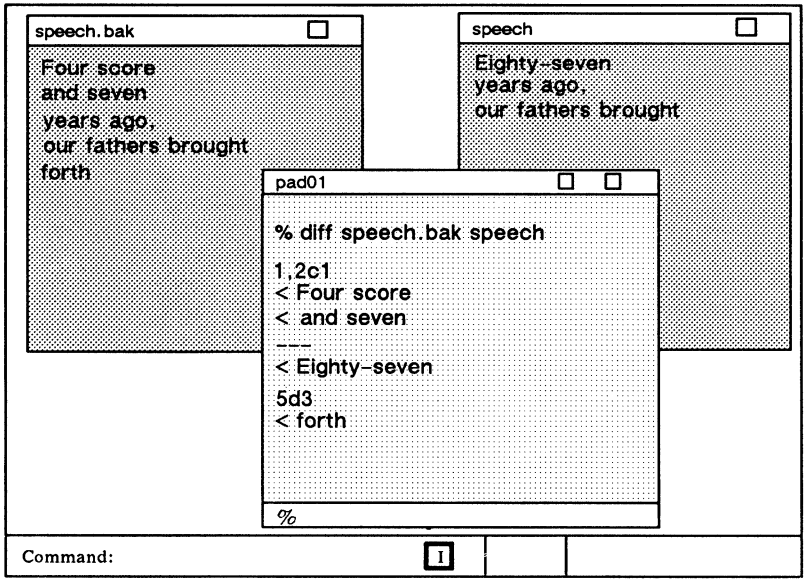


Figure 11–3. Comparing Two ASCII Files

Chapter 12

Managing Directories

Directories are the naming tree components that contain other objects. Table 12–1 summarizes the commands for managing directories.

Table 12–1. Commands for Managing Directories

Task	Shell Command
Create a directory	**mkdir** *dir*
Move or rename a directory	**mv** *source target*
Copy a directory tree	**cp –r** *source target*
Compare directory trees	**diff** *source target*
Display contents of a directory	**ls** [*dir*]
Delete a directory	**rmdir** *dir*
Delete a directory tree	**rm –r** *dir*

Creating Directories

Each directory that you create is actually a subdirectory of its parent directory (the directory above it in the naming tree). To create a directory, specify the **mkdir** (make directory) command in the following format:

% mkdir *dir*

The *dir* argument specifies the pathname of the directory you want to create. If you specify multiple pathnames to create multiple directories, separate each pathname with a space. The following command creates a directory named **reports**:

% mkdir /owner/reports

The **mkdir** command creates the directory **reports** as a subdirectory of the parent directory **/owner**. The new directory, **reports**, also receives an initial set of file permissions from those of the parent directory (**/owner**). You can change the file protection mode of **reports** with the **chmod** (change mode) or the **chacl** (change ACL) command. Chapter 14 explains file protection modes and describes how to use the **chmod** command.

Renaming Directories

To change the name of a directory, use the **mv** (move) command in the following format:

% mv [*options*] *source target*

The *source* argument specifies the pathname of the directory you want to rename, and *target* specifies the new name of the directory. For example, the following command line changes the name of the directory **reports** to **progress**::

% mv /owner/reports progress

changes the name of the directory **reports** to **progress**. Notice that the *target* argument applies to the rightmost component (**reports**) of the *source* argument. You cannot use **mv** to change the name of a directory embedded in a pathname. Also note that you cannot move directories across filesystems.

Copying Directory Trees

A directory and all of the objects it contains is called a **directory tree**. A directory tree represents the part of a naming tree that extends from a specific directory through all its files, subdirectories, and links as shown in Figure 12–1.

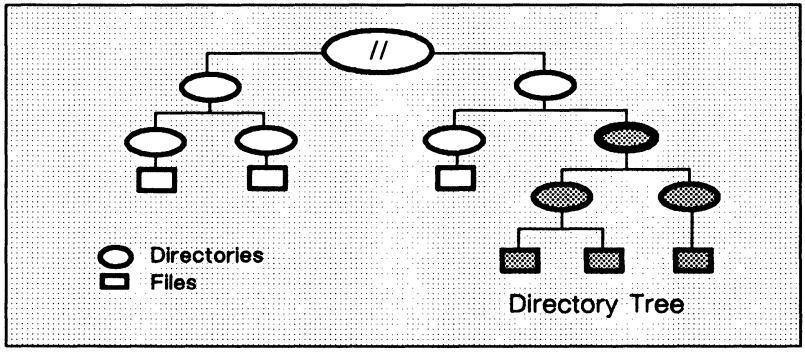


Figure 12–1. Sample Directory Tree

To copy a directory tree to another location, use the **cp** (copy) command with the **–r** option in the following format:

% **cp –r** *source target*

The *source* argument specifies the pathname of the directory you want to copy and *target* specifies the pathname of the naming tree location where you want the copy created. The rules for pathnames described in Chapter 1 apply to both command arguments.

Figure 12–2 illustrates how the **cp –r** command in the following example copies a directory tree.

% cp –r reports //boston/user_1/prog

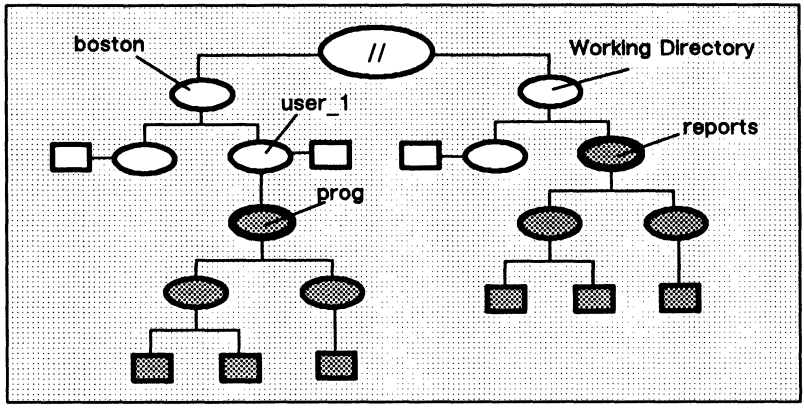


Figure 12–2. Copying a Directory Tree

The **cp –r** command copies the directory tree **reports** and names the copy **prog**. The copy is placed in the directory **user_1**. The **–r** option follows soft links, and will copy the destination of the link. If you are copying a directory that contains soft links, use the command **cp –rs**. The **–s** option says to copy the link text, not the link's destination. The **–p** option preserves file modes during the copy; **–P** preserves all the Access Control Lists (ACLs).

Comparing Directory Trees

To compare the contents of one directory to another, use the **diff** (differential file and directory comparator) command in the following format:

% **diff** [*options*] *source target*

The **diff** command compares all of the objects in the *source* directory against all the objects in the *target* directory, reporting on:

- Any objects that appear in both the source and target but whose contents are different.
- Any objects that appear in the source but not in the target. If the target contains objects that do not appear in the source, **diff** ignores the differences.

The **-r** option causes **diff** to operate recursively; if there are directories with the same name, they are compared in the same way.

Displaying Directory Information

To list the contents of a directory and give information about objects contained within the directory, use the **ls** (list directory) command as follows:

% **ls** [*options*] [*dir*]

The *dir* argument specifies the pathname of the directory, and *options* specifies the types of information you want **ls** to report about the objects it lists. If you omit the *dir* argument, **ls** lists the contents of the current working directory.

The folowing command line lists the contents of the directory **progress_reports**. The **-l** option directs **ls** to list in long format, i.e., to include the protection mode, number of links, owner name, size in bytes, and time of last modification for each file. If a file is a symbolic link, the pathname of the linked-to file is printed preceded by "->". Files with a plus sign (+) after their protection have extended ACL entries; see **lsacl** (list ACL) in the *BSD Command Reference*.

```
% ls -l /owner/progress_reports
total 18
-rwxrwxrwx+ 1   fred   1904 Aug 13 16:09  august.87
-rwxrwxrwx+ 1   fred   2947 Dec 19 15:24  december.86
-rwxrwxrwx  1   fred   1471 Feb 18 14:08  feb.87
lrwxrwxrwx  1   fred   1744 Aug 15 16:44  group -> //node/ann/gp
-rwxrwxrwx  1   fred   1908 Jul 10 13:22  july.87
-rwxrwxrwx  1   fred   2526 Jun 15 16:47  june.87
-rwxrwxrwx  1   fred   1777 May 12 13:28  may.87
lrwxrwxrwx  1   fred   2003 Sep 23 14:13  ytd -> //node/ann/yr
```

Removing Directory Trees

To remove directory trees containing files, use the **rm** (remove directory or file) command in the following format:

% **rm** **-r** *dir ...*

The **rm** **-r** command line attempts to remove the specified directory and all of the objects it contains. For example, the following command removes the directory tree shown in Figure 12–3:

% rm -r reports

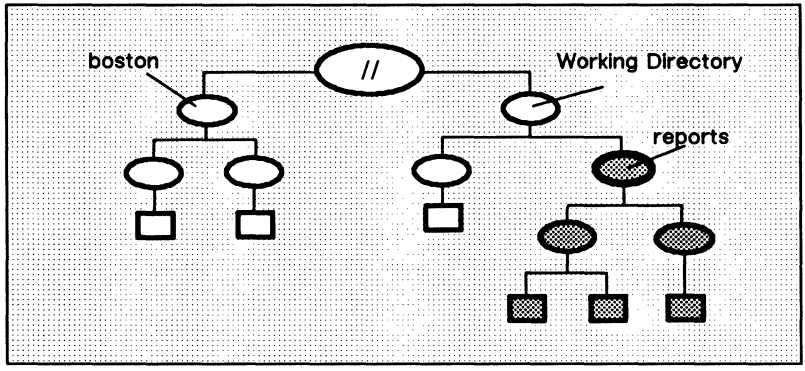


Figure 12–3. Removing a Directory Tree

The command in the previous example removes the directory tree starting at the directory **reports** in the current working directory.

NOTE: If an entry was the last link to the file, the file is destroyed. Removing a file requires write permission in its directory, but neither read nor write permission on the file itself.

To remove empty directory trees (i.e., those containing no files), use the **rmdir** (remove directory) command in the following format:

% **rmdir** *dir* ...

The *dir* argument specifies the pathname of the directory you want to remove.

Chapter 13

Managing Links

As you use the system, you may find that many of the files and directories that you access frequently have unusually long pathnames. You can eliminate the inconvenience of typing a lengthy pathname by creating a shorthand name for the object, called a **link**.

There are two kinds of links: hard links and soft links. A **soft link** is a special object that contains the name of another object. Thus, when you specify a soft link as a pathname or part of a pathname, the system substitutes the pathname that the link contains (the resolution name) for the name of the link. A **hard link** is another entry in the naming tree for the same file on disk. Table 13–1 summarizes the commands used to manage links.

Table 13–1. Commands for Managing Links

Task	Shell Command
Create a link	**ln** [**-s**] *source target*
Rename a link	**mv** *source target*
Copy a soft link	**cp -s** *source target*
Remove a link	**rm** *link_name*

Creating Links

To create a link, specify the **ln** (create link) command in the following format:

% **ln** [*option*] *source target*

The *source* argument specifies the pathname of the link, and *target* specifies the pathname of the object to which the link points. The rules for pathnames described in Chapter 1 apply to both arguments.

The **ln** command creates both hard and soft links to files. By default, **ln** makes hard links that are indistinguishable from the original directory entry. Any changes to a file are effective regardless of the name used to reference the file. Hard links do no span file systems and may not refer to directories.

You must use the **-s** option to **ln** create a soft (or symbolic) link, that is, a link containing the name of the file to which it is linked. Soft links may span file systems and may refer to directories.

The following command creates a soft link:

% **ln -s /owner/april/progress_reports reports**

The command in this example creates a link named **reports** in the process's current working directory. The link contains the pathname for the subdirectory **progress_reports**.

Renaming Links

To change the name of a link, use the **mv** (move or rename file) command in the following format:

% **mv** [*options*] *source target*

The *source* argument specifies the pathname of the link you want to rename, and *target* specifies the new name of the link. For example, the command:

```
% mv reports progress
```

changes the name of the link **reports**, in the current working directory, to **progress**.

Copying Soft Links

Copying links is basically the same as copying files; when you copy a link, you create a copy of the link in another location in the naming tree. To copy a soft link, use the **cp** (copy) command in this format:

```
% cp -s source target
```

The *source* argument specifies the pathname of the link you want to copy, and *target* specifies the naming tree location where you want the copy created. The rules for pathnames described in Chapter 1 apply to both command arguments.

The **cp -s** command always creates a copy of the source link at the location specified by the target. For example:

```
% cp -s reports /user_1/status
```

creates a copy of the source link **reports** in the directory **user_1**. And, since the target specifies the pathname of a link, **cp** assigns the copy the name specified by the target, **status**.

If the target specifies the pathname of a directory, **cp** creates a copy of the source link in the target directory (the current working directory if you omit the target) and assigns the copy the name of the source link. For example:

```
% cp -s reports /user_1
```

copies the link **reports** from the current directory to the target directory **user_1**. Because **cp** assigns the copy the name of the source link, the new link has the pathname **/user_1/reports**.

Removing Links

To remove one or more links, use the **rm** (remove file or directory) command in this format:

% **rm** *link_name*

The *link_name* argument specifies the pathname of the link you want to remove. If you specify multiple pathnames to remove multiple links, separate each pathname with a space.

The following command removes the link **reports** from the current working directory:

% rm reports

Chapter 14

Controlling Access to Files and Directories

Domain/OS lets you protect your files and directories from unauthorized use, as each file and directory in the system has a protection mode that defines:

- Who can use the object
- What operations these users can perform on the object

Protection modes can, for example, authorize some users to read the file, and permit others to change it. Domain/OS software provides the standard UNIX object protection scheme, as well as an important extension called an **Access Control List (ACL)**. ACLs provide more comprehensive protection (i.e., more types of permissions) than do standard UNIX system permissions. This chapter describes both forms of protection.

Using Standard UNIX Object Protections

Under standard UNIX protection mechanisms, an object's permissions should appear as some form of the set **rwxrwxrwx**. Each **r** represents read permission, each **w** write permission, and each **x** execute permission.

The first set of **rwx** describes the permissions granted to the owner of the object (usually, the user ID of the person who created the object). The next set shows group permission, allowing the creator of the file to restrict access to the object to a group. The final set shows permission of others (excluding object owner and group).

Listing File Permissions

The **ls** (list directory) command, executed with the **-l** option, shows the permissions set (modes) for any given object. (This option produces a "long listing" that also shows other attributes of the object, such as size in bytes and date of last modification. However, we will only concern ourselves with the display of object permissions here.)

For example, to determine the protection modes for a file named **report**, type the following line:

```
% ls -l report
```

Assuming that the object were a file with completely open permissions (i.e., owner, group, and others had read, write, and execute rights), the output might be as follows:

```
-rwxrwxrwx 1 john  eng  941 Aug 2 14:48  reports
```

If the file were protected with an Access Control List that added permissions for additional users, groups, and organizations, the output would contain a plus sign after the standard permissions, as follows:

```
-rwxrwxrwx+ 1 john  eng  941 Aug 2 14:48  reports
```

You can use the **lsacl** (list ACL) command to list the ACL entries for an object. ACLs are discussed in detail later in this chapter.

If **reports** were a directory, the output would be similar, except for the "d" before the listing of access rights:

```
drwxrwxrwx  1 john  eng  5941 Aug 2 14:48  reports
```

If **reports** were a link, you would see an "l" at the beginning of the access list:

```
lrwxrwxrwx 1 john eng 941 Aug 2 14:48 reports -> //no/reports
```

For more information on listing permissions, see **ls** in the *BSD Command Reference*.

Changing Access Rights

To change permissions on an object that you own, use either the **chmod** (change mode) command or the **chacl** (change ACL) command. You may change permissions according to absolute mode (using an octal number) or symbolic mode (using one or more alphabetic characters).

As a brief example of how these commands are used, suppose that you want to change the permissions of a file that is currently readable, writable, and executable by all users on your network. You want to ensure that only you, the file owner, have complete access permissions. You also want to grant only read rights to your project group and all others. The following command line, where **reports** is the file whose permissions are being changed, shows how to make the change using the **chmod** command in symbolic mode:

```
% chmod u=rwx,go=r reports
```

You may use **chacl** in the following manner to perform the same task (the additional permission **p** indicates that you may change permissions):

```
% chacl u=prwx,go=r reports
```

This command line shows the same change done using **chmod** in absolute mode:

```
% chmod 744 myfile
```

For more information about specific modes to use for changing file permissions, see **chmod** and **chacl** in the *BSD Command Reference*.

Using Access Control Lists (ACLs)

The ACL for each file and directory contains one or more ACL entries. An entry describes the operations a user or set of users can perform on the object. For a file, the ACL can also contain an indicator that it belongs to a **protected subsystem**.

Each ACL entry consists of two elements: a **subject identifier (SID)** specification and a set of **access rights**. Figure 14–1 shows the elements that make up an ACL entry.

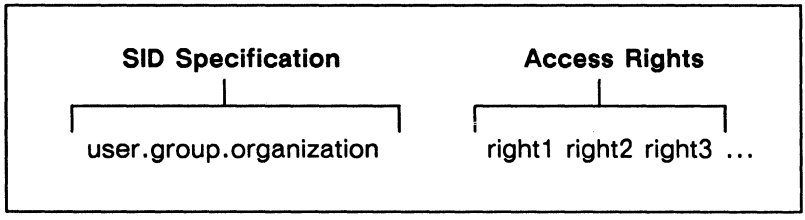


Figure 14–1. Structure of an ACL Entry

The SID specification identifies a specific user or group of users. The access rights define what operations that user or group can perform on the object. Let's take a closer look at these two elements to see how the system uses them to control access to an object.

The Subject Identifier (SID)

As described earlier, the system associates each user process with a SID that identifies the owner of the process. Like the SID specification in an ACL entry, the SID assigned to a user process has the following format:

user.group.organization

The SID consists of three fields: *user*, *group*, and *organization* (abbreviated *ugo*). When you log in, the system gathers SID information for your account. Then, each time you create a process, the system assigns the same SID to it to identify you as the owner.

When a user requests access to a file or directory, the system checks the object's ACL. Specifically, the system searches for an ACL entry whose SID matches the SID of the user's process. If the system doesn't find a match, it denies the user access to the object. If the system does find a match, it grants the user the set of rights specified by the ACL entry. (The "Access Rights" section describes the meaning of the various access rights.)

Figure 14–2 shows a set of two ACL entries for a file.

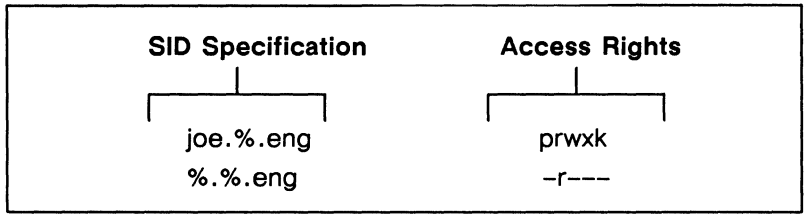


Figure 14–2. Sample ACL Entries

The percent signs (%) that appear in the different fields of the SID specification are wildcards. Wildcards match any name in the network within a specific SID field. For example, the SID for the second ACL entry in Figure 14–2 (**%.%.eng**) contains wildcards in the *user* and *group* fields. These wildcards match any name in the corresponding fields of a user's process SID. As a result, the ACL entry for **%.%.eng** matches any process SID with the organization name **eng**.

When a user process requests access to an object, the system starts its search for a matching SID at the most specific SID specification and continues searching toward the most general. As soon as the system finds a specification that matches the process's SID, it stops the search and grants the rights listed in that ACL entry.

For example, the SID specification for the first ACL entry in Figure 14–2 (**joe.%.eng**) is more specific than that of the second entry (**joe** is a specific user in the organization **eng**). Suppose a process with the SID **joe.bridge.eng** tries to access the object. In this case, the SIDs for both ACL entries match the process SID. However, since the system matched the more specific SID (**joe.%.eng**) first, it grants the process the associated rights (**prwxk**).

When you create a process, the system assigns an SID consisting of a username (owner), a group and an organization. The ACL entries for the owner, group and organization are called the **required ACL entries**. Each object in the system has a set of required ACL entries specifying rights for an owner, a group, an organization, and all others (world). The required ACL entries provide the standard UNIX protection modes, and add the **p** and **k** permissions to the standard **rwx** set. The **p** (protect) specifies rights to change an object's permissions; **k** (keep), prevents the file from being removed or having its name changed. For directories, **x** (execute/search) allows a directory to be searched for subordinate objects. These protection rights are indistinguishable from standard UNIX protections.

You can also create **extended ACL entries** for an object. An extended ACL entry allows you to extend access rights to other users, groups, and organizations. Figure 14–3 shows an object with required and extended ACLs.

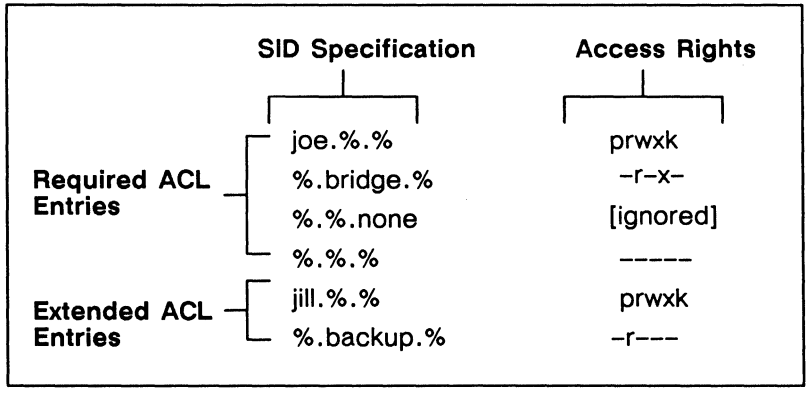


Figure 14–3. Sample Extended ACL Entries

Access Rights

Access rights specify what operations, such as read, write, and execute, a user process can perform on a particular file or directory. Table 14–1 lists the access rights for files and directories.

For example, the following ACL entry for a file grants the specified set of access rights to all users:

```
%.%.%                    -rwx-
```

In this example, the **rwx** specification indicates that the file has read (**r**), write (**w**), and execute (**x**) rights. Notice the hyphens that surround **rwx** rights.

When you list the ACL entries for an object (see the "Displaying ACLs" section) the system displays the hyphens to represent access rights that are not valid (denied) for the entry. In the previous example, the entry denies **p** and **k** rights (represented by hyphens) and grants **r**, **w**, and **x** rights.

As you'll see later in this chapter, you can also deny certain users any access to an object. For example:

```
%.bridge.eng          prwxk
%.%.eng               -----
```

This ACL denies every user in the **eng** group access to the file, except those working on the **bridge** project.

As shown in Table 14–1, the types of access for directories are different than those for files.

Table 14–1. Access Rights for Files and Directories

Access Right	Abbrev.	Meaning for Directories	Meaning for Files
Protect	p	Change the object's ACLs.	
Keep	k	Prevent deletion or changing of name.	
Read	r	List entries.	Read file contents.
Write	w	Add, change, or delete entries.	Write to the file.
Execute/ Search	x	Allow directory to be searched for subordinate objects.	Execute object file.

NOTE: To remove a directory tree, you need write rights to any object being removed. If objects are protected with keep rights, you must have protect rights to the object as well.

Searching Directories and Removing Objects

To access an object, you must have appropriate rights to both the object and its parent directory. To access an object, its parent must grant you search (x) rights. To remove an object, its parent must grant you write (w) rights. Consider the following example:

```
% ls /owner/reports
```

In order to list the contents of reports, you must have search rights to its parent directory **/owner**, as well as read rights to **reports**. Similarly, to remove the subdirectory **reports**, you need write rights to both **/owner** and **reports**.

If **reports** contains additional objects, you need write rights to **reports** to remove them. Therefore, to remove a directory tree, you must have write rights to the parent directory and all of its subdirectories except the subdirectories at the very bottom of the tree.

Managing ACLs

By default, the system assigns an ACL to every file or directory that you create. (The "Initial ACLs" section describes how the system assigns ACLs to objects.) You can list, change, and copy an object's ACL using the following shell commands:

- **lsacl** (list ACLs)
- **cpacl** (copy ACLs)
- **chacl** (change ACLs)
- **dbacl** (dialog–based ACL editor)

Displaying ACLs

To display an object's ACL, use the **lsacl** (list ACL) command in the following format:

lsacl *name* ...

The *name* argument specifies the pathname of the object whose ACL you want to list. For example:

% lsacl /owner/report

This command lists the ACL entries for the file **report**. Figure 14–4 shows a sample display produced by this command.

```
jill.%.%                prwx-
%.bridge.%              -rwx-
%.%.none                [Ignore]
%.%.%                   -----
```

Figure 14–4. Sample ACL Display

Changing ACLs

You can change an object's ACL using either the **chacl** (change ACL) command or the **dbacl** (dialog-based ACL editor) command. These commands let you add, change, and remove ACL entries. You can also use them to change a directory's initial ACLs. (The "Changing Initial ACLs" section describes how to use these commands in this manner).

You can change the ACLs of several objects either by specifying multiple pathnames (separating each pathname with a space) or by using pathname wildcards.

This simplest form of the **chacl** command is as follows:

chacl *SID<operator><rights> name ...*

The *SID* argument is the user, group, and organization to which the rights apply; *operator* is one of + (add to existing rights), - (remove from existing rights), or = (assign absolute rights); *rights* are one or more letters identifying types of access for directories and files; and *name* is the pathname of a file or directory. The **chacl** command also has options that allow you to change initial directory and file ACLs, and inheritence rights.

This section describes how to use the **chacl** command to change ACLs. For a complete description of **chacl**, see the *BSD Command Reference*. Table 14-2 summarizes the commands used to change ACLs.

Table 14–2. Summary of Commands for Changing ACLs

Task	Command
Display an object's ACL	**lsacl** *name*
Set an ACL entry*	**chacl** *SID=rights name*
Add rights to an ACL entry*	**chacl** *SID+rights name*
Remove rights from an ACL entry*	**chacl** *SID–rights name*
Remove an ACL entry	**chacl -D** *ugo name*
Set the required entry for owner (**u**)	**chacl u=***rights name*
Set the required entry for group (**g**)	**chacl g=***rights name*
Set the required entry for organization (**z**)	**chacl z=***rights name*
Set the required entry for world (**o**)	**chacl o=***rights name*
Ignore all required entries	**chacl a=I**
* If entry doesn't exist, it will be added.	

NOTE: Each object must have the required entries of user, group, and organization. However, it is sometimes useful to have these rights specified, but not used for rights checking. This may be done by using the special case mode, **I** (ignore). The ignore mode is only valid for the required entries, owner (**u**), group (**g**), and organization (**z**).

Rules to Specify ACL Entries

Most of the **chacl** commands described in this section require you to specify SID and access right information. For example, to add an ACL entry, you must specify an SID and a set of access rights. Before you attempt to use **chacl** commands, you should understand the rules for specifying SIDs and access rights.

When you specify an SID, you can use the percent sign (%) wildcard character in each field to match any name in the corresponding field of a process SID. For example, the following SID matches any process SID in the system with the username **joe**:

```
joe.%.%
```

When you specify a SID that uses % wildcards, you may omit trailing % wildcards and the periods that separate them. For example, the following SID specifications are the same:

```
joe.%.%
joe.%
joe
```

Table 14–1 lists the access rights that you can specify for files and directories. To specify access rights individually, use the one–letter abbreviations listed therein. For example:

```
% chacl joe+rw report
          ┗┛
          ┃
     access rights
```

The command in this example specifies the read (**r**) and write (**w**) access rights for the file **report.**

To deny rights (grant no rights) for an entry, specify an equal sign (=) without any access rights as follows:

```
% chacl joe=  report
          ┗┛
          ┃
    no access rights
```

You can also use any of the special class names in Table 14–3 to specify a set of required rights. For example:

```
% chacl ugz=prx report
```

required rights abbreviations

The **ugz** abbreviation in this example specifies the required *user*, *group*, and *organization* rights.

> **NOTE:** The **chacl** command will not allow you to perform an operation that restricts everyone from changing an ACL. At least one user must have the protect (**p**) rights to change the ACL.

Table 14–3. Abbreviations for Required Rights

Name	Meaning
u	Required user entry
g	Required group entry
z	Required organization entry
o	Required other (world) entry
a	All of the required entries

Setting ACL Entries

To set an entry (SID and rights) in an ACL, use the = operator in the following format:

chacl *SID=rights name*

The *SID* argument specifies the SID for which to set *rights*. The = operator directs **chacl** to add the specified SID and access rights to the ACL if it didn't exist. For example:

```
% chacl %.%.man=prwxk report
```

The command in this example adds a new ACL entry to the ACL for the file **report**, if necessary. The new entry grants full access (**prwxk**) to anyone in the organization named **man**.

Changing Entry Rights

To change the access rights for an SID, use the + or - operator in the following format:

chacl *SID+rights name*
or
chacl *SID-rights name*

The *SID* argument specifies the SID for which you want to add or subtract *rights*.

For example, suppose the file **report** has the following ACL entry granting full rights:

```
%.%.man                 prwxk
```

The following command changes the access rights for **%.%.man** to read (**r**) access:

```
% chacl %.%.man-pwxk report
```

As a result, the new ACL entry now looks like this:

```
%.%.man   -r---
```

Adding Entry Rights

To add access rights to an existing ACL entry, use the + operator in the following format:

chacl *SID+rights name*

The *SID* argument specifies the SID for which you want to add *rights*, creating an entry if necessary.

For example, suppose the file **report** has the following ACL entry granting full rights:

```
%.%.man                    -r---
```

The following command adds write (w) and execute (x) rights to the current access rights for **%.%.man**.

```
% chacl %.%.man+wx report
```

As a result, the ACL entry now looks like this:

```
%.%.man                    -rwx-
```

Removing ACL Entries

To remove an entry (SID and rights) from an ACL, use the **-D** option in the following format:

chacl -D *SID name*

The *SID* argument specifies the SID for the entry you want to remove. For example:

```
% chacl -D %.%.man.% /owner/report
```

This command removes the entry **%.%.man** from the ACL for the file **/owner/report**.

Copying ACLs

To copy an ACL from one object to another, use the **cpacl** (copy access control list) command in the following format:

cpacl *source destination ...*

The *source* argument specifies the pathname of the object whose ACL you want to copy. The *destination* argument specifies the pathname of the object to which you want the ACL copied.

The following command copies the ACLs from the directory **/owner** to the directories **/user_1** and **/user_2**:

```
% cpacl /owner  /user_1  /user_2
```

Initial ACLs

Whenever you create a new file or directory, the system assigns it a default ACL by copying a special ACL, called an **initial ACL**, from the parent directory. Each directory, in addition to its own ACL, has two initial ACLs: an **initial file ACL** for new files, and an **initial directory ACL** for new directories.

For example, if you create a file named **report** in the directory **owner**, the system assigns **report** the initial file ACL of the directory **owner**. If you create a subdirectory in **owner**, the system assigns the new subdirectory **owner**'s initial directory ACL. New subdirectories also receive a set of initial ACLs that match the parent directory's initial ACLs. In this example, the new subdirectory also receives **owner**'s initial ACLs. Figure 14–5 shows how the system assigns initial ACLs to files and directories.

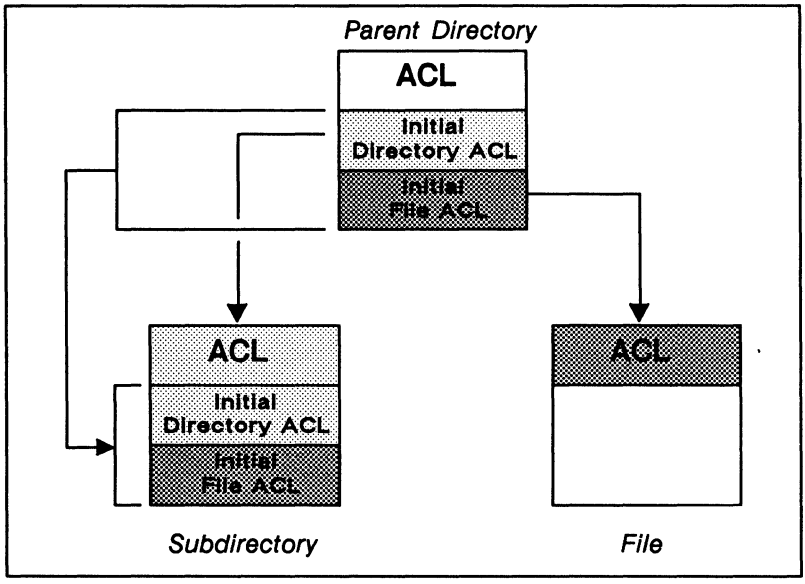


Figure 14–5. Initial ACLs for Files and Directories

Table 14–4 summarizes the commands used to change and copy initial ACLs.

Table 14–4. Summary of Commands for Changing and Copying Initial ACLs

Task	Command
Change initial directory ACL	**chacl -d** *command*
Change initial file ACL	**chacl -f** *command*
Copy both initial ACLs	**cpacl -df** *source destination*
Copy initial directory ACL	**cpacl -d** *source destination*
Copy initial file ACL	**cpacl -f** *source destination*
Display initial directory ACL	**lsacl -d** *name*
Display initial file ACL	**lsacl -f** *name*

Displaying Initial ACLs

You can display a directory's or file's initial ACLs with the **lsacl** command.

To display the initial directory ACL, use **lsacl** with the **-d** option in the following format:

lsacl -d *dir*

The **-d** option directs **lsacl** to display the initial directory ACLs, and *dir* specifies the pathname of the directory whose initial ACLs you wish to display. For example:

```
% lsacl -d /owner
```

The command in this example displays the initial directory ACL for the directory **/owner**.

To display the initial file ACL, use **lsacl** with the **-f** option in the following format:

lsacl -f *dir*

The **-f** option directs **lsacl** to display the initial file ACLs, and *dir* specifies the pathname of the file whose initial ACLs you wish to display. The following example displays the initial file ACL for the file **report**:

```
% lsacl -f report
```

Changing Initial ACLs

You can change a directory's initial ACLs with the **chacl** command.

To change the initial directory ACL, use **chacl** with the **-d** option in the following format:

chacl -d *command dir*

The **-d** option directs **chacl** to change initial directory ACLs, and *command* specifies one of the ACL changing commands described in the "Changing ACLs" section discussed earlier.

To set the rights for an SID in the initial directory ACL for **/owner**, use the = operator as follows:

```
% chacl -d %.%.eng=rwx /owner
```

The following example uses the **-d** option to take away write (**w**) rights from the entry in the previous example:

```
% chacl -d %.%.eng-w /owner
```

To change the initial file ACL, use the **chacl** command with the **-f** option in the following format:

chacl -f *command dir*

The **-f** option directs **chacl** to change initial file ACLs, and *command* specifies one of the ACL changing commands described in the "Changing ACLs" section discussed earlier.

Copying Initial ACLs

You can copy a directory's initial ACLs using the **cpacl** command in the following format:

cpacl *option source destination*

The *option* argument specifies one of the options listed in Table 14–5. The *source* argument specifies the pathname of the object whose initial ACL you want to copy. The *destination* argument specifies the pathname of the object to which you want the initial ACL copied.

Table 14–5. Options for Copying Initial ACLs

Option	Description
-df	Copies both the initial file and initial directory ACLs from the source to the destination.
-d	Copies the initial directory ACL from the source to the destination.
-f	Copies the initial file ACL from the source to the destination.
-t	Copies the ACLs from the source to the inital file and initial directory ACLs on the destination.
-i	Copies the initial file or initial directory ACL to the ACL on the destination.

The command in the following example uses the **-df** options to copy the inital file and directory ACLs from the directory **/owner** to the directory **/user_1**.

```
% cpacl -df /owner /user_1
```

To copy only the initial file ACL, use the **-f** option as shown in the following example:

```
% cpacl -f /owner /user_1
```

For a complete description of how to use the **cpacl** command to copy initial ACLs, see the *BSD Command Reference*.

Appendix A

Initial Directory and File Structure

The following illustrations show how the system organizes the software that we supply with your node:

- Figure A–1 shows the contents of the node entry directory (/)
- Figure A–2 shows the files and directories in the system software directory (**/sys**)
- Figure A–3 shows the files and directories in the Display Manager directory (**/sys/dm**)
- Figure A–4 shows the network management directory (**/sys/net**)

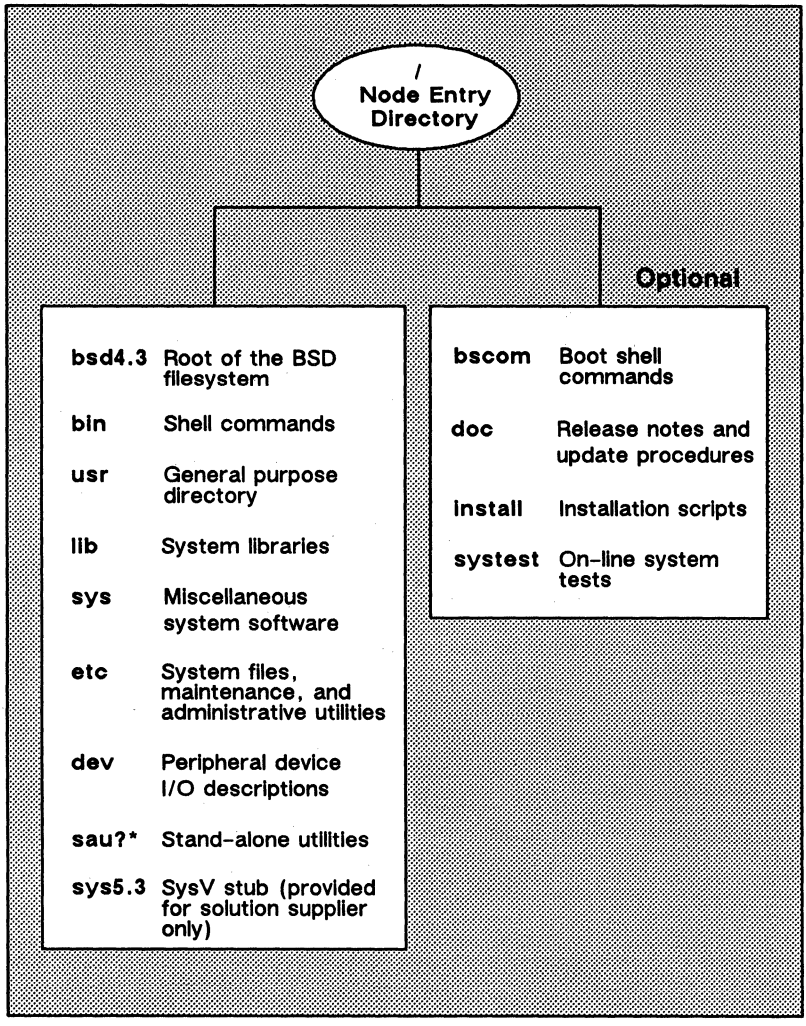


Figure A–1. The Node Entry Directory (/) and Subdirectories

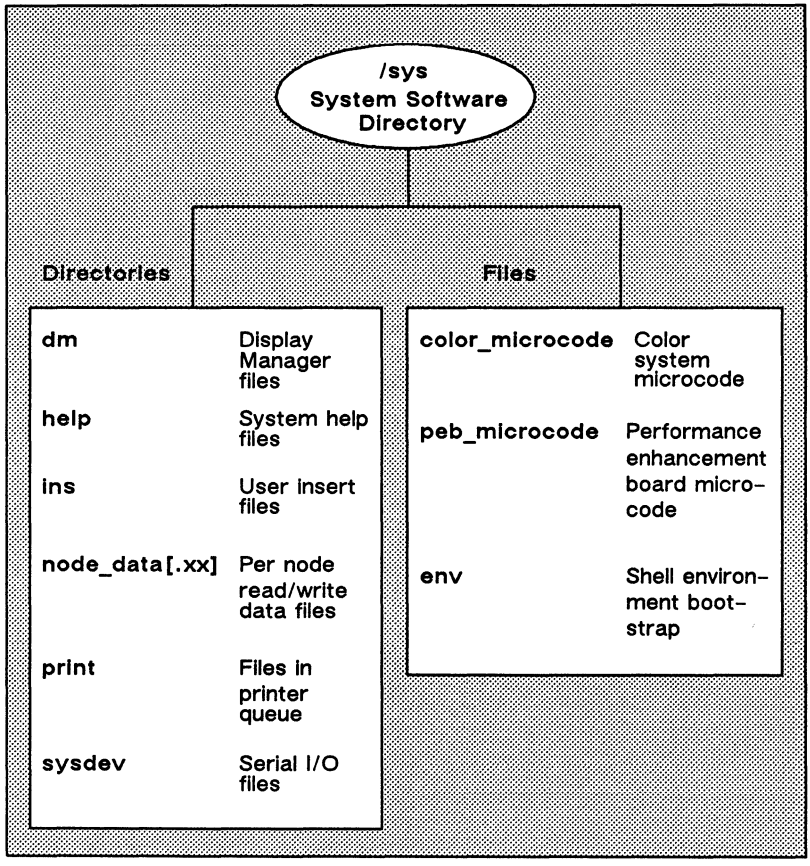


Figure A–2. The System Software Directory (/sys)

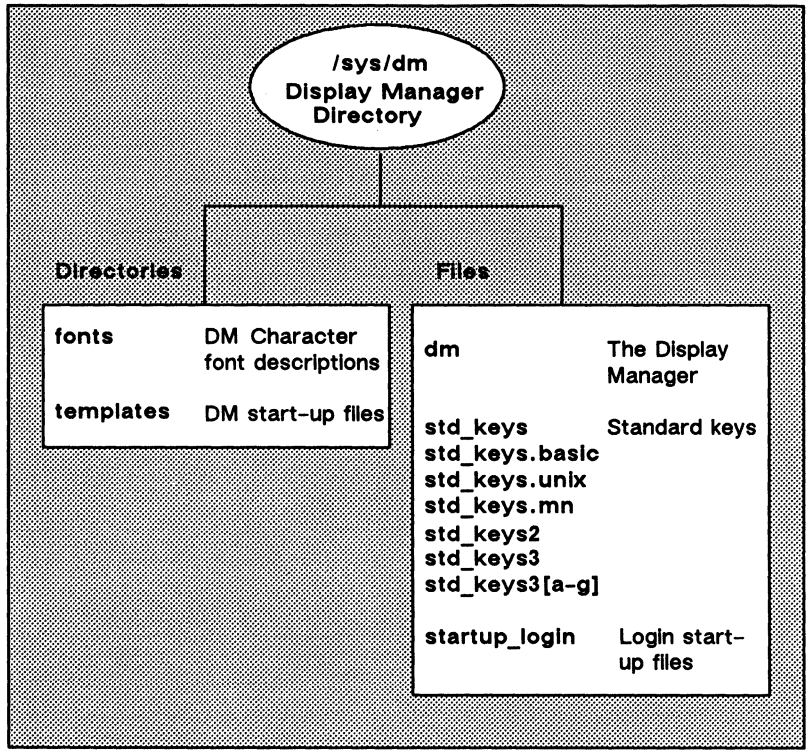


Figure A–3. The Display Manager Directory (/sys/dm)

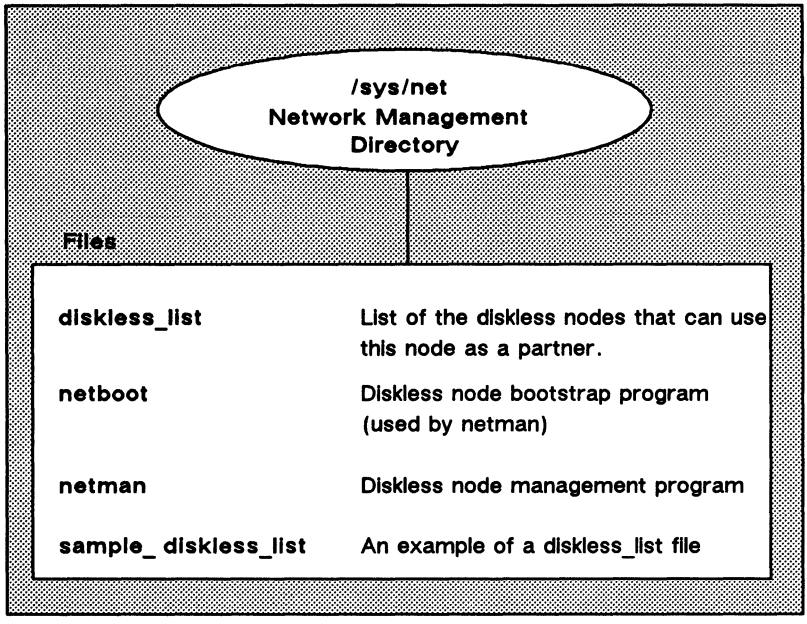


Figure A–4. The Network Management Directory (/sys/net)

Appendix B

Summary of Predefined Standard and UNIX Key Definitions

This appendix summarizes the predefined standard and UNIX key definitions read during DM startup. These are found in the files **/sys/dm/std_keys.basic** and **/sys/dm/std_keys.unix**. This appendix also includes special operating considerations for the Domain Multinational keyboards. Figure B–1 shows the key names for the Multinational keyboard keypad.

Table B–1. Controlling the Cursor

Task	DM Command	Predefined Key
Move left one character	**al**	← (LA)
Move right one character	**ar**	→ (LC)
Move up one line	**au**	↑ (L8)
Move down one line	**ad**	↓ (LE)
Set arrow key scale factors	**as** *x y*	None
Move to the beginning of line	**tl**	|← (L4)
Move to end of line	**tr**	→| (L6)
Move to top line in window	**tt**	SHIFT/ ↑ (LDS)
Move to bottom line in window	**tb**	SHIFT/ ↓ (LFS)
Move to window borders	**twb [l, r, t, b]**	None
Move to the beginning of next line	**ad; tl**	CTRL/K
Tab left	**thl**	CTRL/<TAB>
Tab right	**th**	SHIFT/<TAB>
Set tabs	**ts** [*n1, n2...*]	None
Move to DM input pad	**tdm**	<CMD>(L5)
Move to next window on screen	**tn**	<NEXTWNDW> (LB)
Move to previous window	**tlw**	CTRL/~
Move to next window in which input is enabled	**ti**	None

Table B–2. Creating Processes

Task	DM Command	Predefined Key
Create new process, pads, and windows	**cp** *command*	None
Create new process without pads or windows	**cpo** *command*	None
Create a server process	**cps** *command*	None

Table B–3. Controlling Processes

Task	DM Command	Predefined Key
Quit a process	**dq -c 9010003**	CTRL/\
Interrupt a running process	**dq -i**	CTRL/C
Stop or blast a process	**dq [-s|-b]**	None
Suspend execution of a process	**dq -c 120028**	CTRL/Z
Resume execution of a suspended process	**dc**	None

Table B–4. Creating Pads and Windows

Task	DM Command	Predefined Key
Create an edit pad and window	**ce** *file*	<EDIT> (R4)
Create a read-only window	**cv** *file*	<READ> (R3)
Create a copy of an existing pad and window	**cc**	None

Table B–5. Closing Pads and Windows

Task	DM Command	Predefined Key
Close window and pad; update file	**pw; wc -q**	<EXIT> (R5)
Close window and pad; no update	**wc -q**	<ABORT> (R5S)
Close (delete) a window	**wc [-q|-f]**	None

Table B–6. Managing Windows

Task	DM Command	Predefined Key
Change window size	**wg**	None
Change window size with rubberbanding	**wge**	<GROW> (LA3)
Move a window	**wm**	None
Move a window with rubberbanding	**wme**	<MOVE> (LA3S)
Set scroll mode	**ws [-on|-off]**	None
Set autohold mode	**wa [-on|-off]**	None
Scroll and autoholf mode	**wa; ws**	CTRL/A
Set hold mode	**wh**	<HOLD> (R6)
	wh -on	CTRL/S
	wh -off	CTRL/Q
Define position of default window *n*	**wdf [*n*]**	None
Acknowledge alarm	**aa**	None
Acknowledge alarm and pop window	**ap**	None

Table B–7. Moving Pads

Task	DM Command	Predefined Key
Move top of pad into window	**pt**	None
Move cursor to first character in pad	**pt; tt; tl**	CTRL/T
Move bottom of pad into window	**pb**	None
Move cursor to last character in pad	**pb; tb; tr**	CTRL/B
Move pad *n* pages	**pp** [-]*n*	 (LD, LF)
Move pad *n* lines	**pv** [-]*n*	SHIFT/ ↑ (L8S) SHIFT/ ↓ (LES)
Move pad *n* characters	**ph** [-]*n*	 (L7, L9)
Save transcript pad in a file	**pn**	None

Table B–8. Controlling Window Groups and Icons

Task	DM Command	Predefined Key
Create or add to a window group	**wgra** *grp_name* [*entry_name*]	None
Remove a window from window group	**wgrr** *grp_name* [*entry_name*]	None
Make windows invisible	**wi** *entry_name*	None
Change windows to icons	**icon** [*entry_name*] [*options*]	SHIFT/<POP>
Set icon positioning and offset	**idf**	None
Display list of windows in group	**cpb** *group_name*	None

Table B–9. Setting Edit Modes

Task	DM Command	Predefined Key
Set read/write mode	**ro** [-on|-off]	SHIFT/<AGAIN>
Set insert/overstrike mode	**ei** [-on|-off]	<INS> (L1S)

Table B–10. Inserting Characters

Task	DM Command	Predefined Key
Insert string at cursor	**es** '*string*'	Default DM operation
Insert newline character	**en**	<RETURN>
Insert tab character	None	<TAB>
Insert raw (noecho character)	**er** *nn*	None
Insert a new line after current line	**tr; en; tl**	<F1>
Insert end-of-file mark	**eef**	CTRL/D

Table B–11. Deleting Text

Task	DM Command	Predefined Key
Delete character at cursor	**ed**	<CHAR DEL> (L3)
Delete character before cursor	**ee**	<BACK SPACE>
Delete word	**dr;/[~ a-z0-9 ! $_]/xd**	<F6>
Delete previous word	**dr;\[~ @@t@@n]\;\[@@t@@n]\; ar;xd**	CTRL/W
Delete from cursor to end of line	**es '';ee;dr;tr xd;tl;tr**	<F7> (L3A)
Delete from cursor to beginning of line	**dr;tl;xd**	CTRL/U
Delete entire line	**cms;tl;xd**	<LINE DEL> (L2)

Table B–12. Copying, Cutting, and Pasting Text

Task	DM Command	Predefined Key
Copy text to a paste buffer or file	**xc** [*name* | **-f** *file*] [**-r**]	<COPY> (L1A)
Cut (delete) text and write it to a paste buffer or file	**xd** [*name* | **-f** *file*] [**-r**]	<CUT> (L1AS)
Paste (write) text from a paste buffer or file into a pad	**xp** [*name* | **-f** *file*] [**-r**]	<PASTE> (L2A)

Table B–13. Commands for Searching for Text

Task	DM Command	Predefined Key
Search forward for string	/*string*/	None
Search backward for string	*string*\\	None
Repeat last forward search	//	CTRL/N
Repeat last backward search	\\	CTRL/P
Cancel search or any action involving the **echo** command	**abrt**	CTRL/X
Set case comparison for search	**sc** [**-on**] [**-off**]	None

Table B-14. Commands for Substituting Text

Task	DM Command	Predefined Key
Substitute *string2* for all occurrences of *string1* in a defined range	**s**/*string1*/*string2*/	None
Substitute *string2* for the first occurrence of *string1* in each line of a defined range	**so**/*string1*/*string2*/	None
Change case of each letter in a defined range	**case** [-s] [-u] [-l]	None

Operating Considerations for Multinational Keyboards

The Domain Multinational keyboard is a Low-Profile Model II keyboard adapted to international standards. Because of the differences between the North American keyboard and the International keyboard, there are certain operating considerations that you should note. These operating considerations are described in the following sections.

Arrangement of Multinational Keyboard Keys

The Multinational keyboard has seven additional keys that impose a slightly different overall arrangement, as well as some different key labels.

The Multinational keyboard keypad has more keys than the Low-Profile Model II keypad. Figure B-1 shows the names and locations of the numeric keypad keys on the Multinational keyboard.

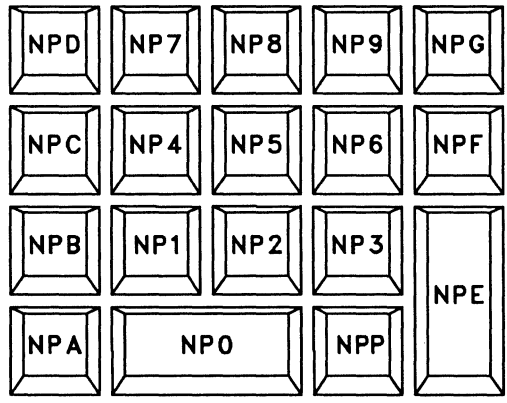


Figure B–1. Multinational Keyboard Numeric Keypad

Key Interpretation During Service Mode

The Mnemonic Debugger (MD) begins executing as soon as you power on your node. The MD reads the program responsible for booting your node, loads it, and transfers control to it. Ordinarily, this is the only function of the MD (aside from performing the automatic fix from a power failure). However, if a node needs to be serviced, the MD is used instead of the normal operating mode.

System administrators must be aware that the MD expects the standard Domain keyboard shown in Figure 4–3. National characters, therefore, may not be valid. This discrepancy currently affects the French keyboard where the Q key and the A key positions are transposed in comparison to the Domain North American keyboard. Because of the difference in these key positions, typing **ex domain_os** on a French keyboard in Service Mode (which calls the Mnemonic Debugger) sends **ex domqin_os** to the system. To correct this, you must press the Q key instead of the A key when starting the OS on such a node.

Appendix C

Summary of Bourne Shell Grammar

The following is a summary of Bourne shell grammar described in Chapter 9 of this manual.

item: word
input-output
name = value

simple-command: item
simple-command item

command: simple-command
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word ...* **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part ...* **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: command
pipeline | *command*

andor: pipeline
andor *&& pipeline*
andor || *pipeline*

command-list: andor
command-list ;
command-list &
command-list ; andor
command-list & andor

input-output: > *file*
< file
>> word
<< word

file: word
& digit
& -

case-part: pattern) *command-list* ;;

pattern: word
pattern | *word*

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits, or underscores starting with a letter

digit: 0 1 2 3 4 5 6 7 8 9

Appendix D

Summary of Bourne Shell Metacharacters and Reserved Words

The following is a summary of Bourne shell metacharacters and reserved words as described in Chapter 9 of this manual.

Syntactic

|	Pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	Command separator
;;	Case delimiter
&	Background commands
( )	Command grouping
<	Input redirection
<<	Input from a here document
>	Output creation
>>	Output append

Patterns

*	Match any character(s) including none
?	Match any single character
[...]	Match any of the enclosed characters

Substitution

${...}	Substitute shell variable
`...`	Substitute command output

Quoting

\	Quote the next character
'...'	Quote the enclosed characters except for '
"..."	Quote the enclosed characters except for $ ' \ "

Reserved Words

- **if**
- **then**
- **else**
- **elif**
- **fi**
- **case**
- **in**
- **esac**
- **for**
- **while**
- **until**
- **do**
- **done**
- { }

Appendix E

Summary of C Shell Metacharacters

The following is a summary of C Shell metacharacters as described in Chapter 8 of this manual. Many of these characters also have special meaning in expressions. See the information on **csh** (C shell) in the *BSD Command Reference* for a complete list.

Syntactic

;	Separates commands to be executed sequentially
|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed in background

Filename

/	Separates components of a file's pathname
.	Separates root parts of a filename from extensions
?	Expansion character matching any single character except a leading dot (.)
*	Expansion character matching any sequence of characters except a leading dot (.)
[ ]	Expansion sequence matching any single character from a set
~	Used at the beginning of a filename to indicate home directories
{ }	Specifies groups of arguments with common parts

Quotation

\	Prevents metameaning of following single character
'	Prevents metameaning of a group of characters
"	Like a single quote ('), but allows variable and command expansion

Input/Output

<	Indicates redirected input
>	Indicates redirected output
<<	Reads shell input up to a specified line
>>	Places output at the end of an existing file
>&	Routes both diagnostic output and the standard output into a specified file
>>&	Places diagnostic and standard output at the end of an existing file
>!	Overrides the **noclobber** variable

Expansion/Substitution

$	Indicates variable substitution
!	Indicates history substitution
:	Precedes substitution modifiers
↑	Used in special forms of history substitution
‘	Indicates command substitution

Miscellaneous

#	Begins shell comment
%	Prefixes job name specifications

Appendix F

Composing European Characters

This appendix describes how to create and display European characters that don't ordinarily appear on Apollo keyboards, and how to switch between European and ASCII characters on Multinational keyboards.

The Compose Function

The default Apollo character set is the ISO 8-bit character set (International Standards Organization 8859/1), commonly known as Latin-1. This set includes the characters necessary to support Western European languages.

European characters do not appear on the standard North American keyboard, and only a subset appear on the various models of the Multinational keyboards. However, you can use the compose function to enter and display any character in the Latin-1 set that does not appear on your keyboard.

To enable the compose function, you must either run the **kbm** command or edit your workstation's start-up file. (See Chapter 3 to determine which start-up file goes with your node type.) If you decide to edit your start-up file, you'll see the following line in that file:

#cps /usr/apollo/bin/kbm -c f5

To turn on the compose function, simply delete the comment character (#) from the line. By default, the command sets <F5> to be the compose key, but you can substitute another keyname on the line if you prefer.

To compose, press <F5> (or your user-defined compose key), followed by the two characters that make up your chosen character. For example, if you want to create an e with a circumflex accent, type the following:

```
<F5> e ^
```

The character appears after you have pressed all three keys. See the section "Character Compose Sequences" for a list of the sequences you must type to compose each Latin-1 character.

The compose function only works if you have a Latin-1 based font loaded on your node. We supply each node with a large group of fonts that are based on Latin-1. Those fonts include:

- Courier family
- **din_f7x11**
- **f5x9**
- **f7x13, f7x13.b**
- Helvetica family
- Legend family
- Std family
- Times family

You should be aware, however, that many software packages use their own fonts, and those fonts may or may not include the European Latin-1 characters.

If you enter a valid key sequence, but the font currently loaded doesn't include the Latin–1 character, the system displays a blank. If you later load a font that *does* have the Latin–1 character and open the file again, the correct character appears.

European Characters and the Multinational Keyboard

You can always use the <F5> method to compose national characters on Multinational keyboards. However, those keyboards also include keys that have both national characters *and* regular ASCII characters engraved on them. By default, the ASCII characters appear on the screen when you press keys with double engravings. You can use ALT mode, however, to tell the system that you want the national characters to appear.

If you hold <ALT> while pressing any key which is marked with both ASCII and national characters, you will toggle that individual key between the ASCII and national character. For example, if you are typing ASCII and then press <ALT> and a double–engraved key, the keyboard will produce the national character on that key.

The SHIFT/<ALT> key combination toggles the entire keyboard between producing ASCII characters and national characters. That is, if you are typing ASCII and then press SHIFT/<ALT>, the keyboard will only produce national characters until you press SHIFT/<ALT> again.

Printing Latin–1 Characters

The print server can process Latin–1 characters correctly, so in most cases, what you see on the screen will match the output from your printer. However, if you have a printer that uses a daisy wheel or other mechanical impact device, you might have to replace the current printer font with one that incluces the Latin–1 characters. Similarly, you might have to load a language–specific PROM for a dot–matrix printer in order to generate the same characters on paper that you see on–screen.

If your printer font does not include a Latin–1 character that is in your file, the system simply prints a space.

Restrictions on Using Latin-1 Characters

You may use Latin-1 characters in any edit pad or DM input window. Likewise, they are acceptable in SysV, Bourne, and, Korn shells, and in the Aegis **/com** shell. However, Latin-1 characters are not legal in all parts of the system. For example, the BSD shells do not support them for input or output.

For more details on the conditions under which you can and cannot use Latin-1 characters, see the release notes for the current system software release that you are using.

Character Compose Sequences

The following chart shows what two characters you must type to compose each individual Latin-1 character.

Table F-1. Compose Sequences for Latin-1 Characters

Keystrokes	Character	Name
<sp><sp>		No break space (NBSP)
!!	¡	Inverted exclamation mark
c/	¢	Cent sign
L-	£	Pound sign
XO	¤	Currency sign
Y-	¥	Yen sign
| |	¦	Broken bar
SO	§	Section sign
""	¨	Diaeresis
co	©	Copyright sign
a_	ª	Feminine ordinal indicator
<<	«	Left angle quotation mark
-,	¬	NOT sign
--		Soft hyphen
RO	®	Registered trade mark sign
-^	¯	Macron
0^	°	Ring above, degree sign
+-	±	Plus-minus sign

2^	²	Superscript two
3^	³	Superscript three
''	´	Acute accent
/u	µ	Micro sign
P!	¶	Paragraph sign, pilgrow sign
.^	·	Middle dot
,,	¸	Cedilla
1^	¹	Superscript one
o_	º	Masculine ordinal indicator
>>	»	Right angle quotation mark
14	¼	Vulgar fraction one quarter
12	½	Vulgar fraction one half
34	¾	Vulgar fraction three quarters
??	¿	Inverted question mark
A`	À	Capital letter A with grave accent
A'	Á	Capital letter A with acute accent
A^	Â	Capital letter A with circumflex
A~	Ã	Capital letter A with tilde
A"	Ä	Capital letter A with diaeresis
A*	Å	Capital letter A with a ring above
AE	Æ	Capital diphthong AE
C,	Ç	Capital letter C with cedilla
E`	È	Capital letter E with grave accent
E'	É	Capital letter E with acute accent
E^	Ê	Capital letter E with circumflex
E"	Ë	Capital letter E with diaeresis
I`	Ì	Capital letter I with grave accent
I'	Í	Capital letter I with acute accent
I^	Î	Capital letter I with circumflex accent
I"	Ï	Capital letter I with diaeresis
D-	Ð	Capital icelandic letter ETH
N~	Ñ	Capital letter N with tilde
O`	Ò	Capitol letter O with grave accent
O'	Ó	Capital letter O with acute accent
O^	Ô	Capital letter O with circumflex
O~	Õ	Capital letter O with tilde
O"	Ö	Capital letter O with diaeresis
xx	×	Multiplication sign

O/	Ø	Capital letter O with oblique stroke
U‘	Ù	Capital letter U wigh grave accent
U’	Ú	Capital letter U with acute accent
U^	Û	Capital letter U with circumflex
U”	Ü	Capital letter U with diaeresis
Y’	Ý	Capital letter Y with acute accent
TH	Þ	Capital icelandic letter THORN
ss	ß	Small German letter sharp s
a‘	à	Small letter A with grave accent
a’	á	Small letter A with acute accent
a^	â	Small letter A with circumflex accent
a~	ã	Small letter A with tilde
a”	ä	Small letter A with diaeresis
a*	å	Small letter A with a ring above
ae	æ	Small diphthong AE
c,	ç	Small letter C with cedilla
e‘	è	Small letter E with a grave accent
e’	é	Small letter E with acute accent
e^	ê	Small letter E with circumflex accent
e”	ë	Small letter E with diaeresis
i‘	ì	Small letter I with grave accent
i’	í	Small letter I with acute accent
i^	î	Small letter I with circumflex accent
i”	ï	Small letter I with diaeresis
d–	ð	Small icelandic letter ETH
n~	ñ	Small letter N with tilde
o‘	ò	Small letter O with grave accent
o’	ó	Small letter O with acute accent
o^	ô	Small letter O with circumflex accent
o~	õ	Small letter O with tilde
o”	ö	Small letter O witth diaeresis
–:	÷	Division sign
o/	ø	Small letter O with oblique stroke
u‘	ù	Small letter U with grave accent
u’	ú	Small letter U with acute accent
u^	û	Small letter U with circumflex accent
u”	ü	Small letter U with diaeresis
y’	ý	Small letter Y with acute accent

th	þ	Small icelandic letter THORN
y"	ÿ	Small letter Y with diaeresis

When creating symbols composed of two alphabetic characters, you can type those characters in uppercase *or* lowercase, but not both. For example, either of the following:

<F5>	<F5>
x	X
o	O

generates the currency symbol (¤), but if you mix the case of the letters, the symbol does not appear.

You must press <F5> (or your user-defined compose key) every time you want to create a character that does not appear on your keyboard.

Glossary

Access rights

These rights list the users who have access to objects in the network, and specify permissions (i.e., read, write, and execute) that each individual user has for accessing specific objects.

Alarm window

The Display Manager alarm window appears near the bottom of your screen. It displays a small pair of bells when a process displays a message in an output window hidden by an overlapping window.

Argument

See Command argument.

Background process

A noninteractive process that runs immune to quit and interrupt signals issued from your node. In this mode, a shell doesn't wait for a command to terminate before it prompts you for another command. This lets you start a task and then go on to another task while the system continues with the initial one. (*See also* Process.)

C language

A general purpose programming language used to generate programs and operating systems.

Command

An instruction that you give a program; the name of an executable file that is a compiled program.

Command argument

A command option or the name of the object upon which the command acts. Command arguments follow commands on the same line, although not all commands require an argument. (*See also* Command option.)

Command list

A sequence of one or more simple commands separated or terminated by a newline or a semicolon.

Command option

Information you provide on a command line to indicate the type of action you want the command to take. (*See also* Default.)

Control character

A special invisible character that controls some portion of the input and output of the programs run on a node. (*See also* Control key sequence.)

Control key sequence

A keystroke combination (<CTRL> followed by another key) used as a shorthand way of specifying commands. To enter a control key sequence, hold <CTRL> down while pressing another key.

Current directory (.)

The location, within the hierarchical naming tree, of the directory that you are working in at a given time. Entering the UNIX command **pwd** (print working directory) prints the name of your current directory. (*See also* Working directory.)

Cursor

The small, blinking box initially displayed in the screen's lower left corner. The cursor marks your current typing position on the screen and indicates which pad receives your input.

Default

Most programs give you a choice of one or more options. If you don't specify an option, the program automatically assigns one. This automatic option is called the default. (*See also* Command option.)

Directory

A special type of object that contains information about the objects beneath it in the naming tree. Basically, it is a file that stores names and links to files. (*See also* File.)

Disk

A thin, record-shaped magnetic plate, or a collection of such plates, used for storing data. The system uses heads (similar to heads in tape recorders) to read and write data on concentric disk tracks. The disk spins rapidly, and the heads can read or write data on any disk track during one disk revolution.

Diskless node

A node that has no disk for storage, and therefore uses the disk of another node. (*See also* Node and Disk.)

Display Manager (DM)

The program that executes commands that start and stop processes, and commands that open, close, move, or modify windows and pads.

DM alarm window

See Alarm window.

DM environment variables

Values set by either the system or the user to determine how the Display Manager handles processes started at login or during command execution.

DM function keys

Single keys that invoke DM commands.

DM input window

The window where you type DM commands (contains the "Command: " prompt).

DM output window

The window that displays output messages from DM commands.

Domain/OS

The operating system that resides on a high-speed communications network connecting two or more Apollo nodes. Each node can use the data, programs, and devices of other network nodes. Each node contains main memory, and may have its own disk, or share one with another node.

EOF

The End-Of-File character is used to terminate a shell and close the pad in which the shell was running. It is generated by pressing CTRL/D.

File

The basic named unit of data stored on disk. A file can contain a memo, manual, program, or picture. (*See also* Directory.)

Filter

A command that reads its input, performs a user-specified task, and prints the result as output.

Foreground

A mode of program execution when a shell waits for a command to terminate before prompting for another.

Full pathname

The pathname of a specific file starting from the network root directory. (*See also* Network root directory and Pathname.)

Function keys

See DM function keys.

Globbing

The expansion of metacharacters to pathnames. This practice is useful as an abbreviated method of finding and selecting file names according to a specified pattern.

Group Identification Number (GID)

A unique number assigned to one or more logins that is used to identify groups of related users.

Hard link

A link that points directly to an object (file).

Here document

A command procedure of the form *command << eofstring* which causes a shell to read subsequent lines as standard input to the command until a line is read consisting of only the *eofstring*. Any arbitrary string can be used for the *eofstring*.

Home directory

Your initial working directory. Your user account specifies the name of your home directory.

Initial working directory

The working directory of the first user process created after you log in.

Input pad

A pad that accepts commands typed at your keyboard.

Input window

The window that displays a program's prompt and any commands typed.

Insert mode

This mode lets you change text displayed in windows by repositioning the cursor and inserting characters. The rest of the line moves right as you insert additional characters.

Kernel

The resident operating system that controls your node's resources and assigns them to active processes.

Keyword parameter

An argument to a command procedure which has the form *name=value command arg1 arg2. . .* and lets shell variables be assigned values when a shell script is called. (*See also* Shell script.)

Link

A special type of object that points from one place in the naming tree to another. (*See also* Hard link and Soft link.)

Link text

The name of the object contained in a symbolic link to show what is being linked. When you use a link name as a pathname or as part of a pathname, UNIX shells substitute the link text for the link name. (*See also* Soft link.)

Logging in

Initially signing on to the system so that you may begin to use it. This creates your first user process.

Main memory

The node's primary storage area. It stores the instruction that the node is executing, as well as the data it is manipulating.

Memory

Any device that can store information.

Metacharacter

See Shell metacharacter.

Mode

The UNIX protection for an object. An absolute mode is an octal number used in conjunction with the UNIX **chmod** (change mode) command to change permissions of files.

Name

A character string associated with a file, directory, or link. A name can include various alphanumeric characters, but never a slash (/) or null character. Remember that certain characters may have special meaning to a shell and must be escaped if they are used.

Naming tree

A hierarchical tree structure that organizes network objects.

Network

Two or more nodes sharing information.

Network root directory

The top directory in the network. Each node has a copy of the network root directory.

Node

A network computer. Each node in the SysV environment can use the data, programs, and devices of other network nodes. Each node contains main memory, and has its own disk, or shares one with another node. (*See also* Diskless node.) We frequently use "terminal" interchangeably with node (or, usually, "the node's keyboard").

Node entry directory

A subdirectory of the network root directory. The top directory on each node. Diskless nodes share the node entry directory of their disked partner node. (*See also* Network root directory.)

Object

Any file, directory, or link in the network.

Operating system

A program that supervises the execution of other programs on your node.

Option

See Command option.

Output window

The window that displays a process's response to your command.

Pad

A temporary, unnamed file that holds the information displayed in a window. A window can display an entire pad, or show only part of the pad. (*See also* Window.)

Parent directory (..)

The directory one level above your current working directory.

Partial pathname

The pathname between the current working directory and a specific file. (*See also* Pathname.)

Partner node

A node that shares its disk with a diskless node. (*See also* Diskless node.)

Password

The string you enter at the "Password:" prompt upon logging in. As you type your password, the system displays dots (. . .) instead of the letters in your password. (*See also* User account.)

Pathname

A series of names separated by slashes that describe the path of the operating system in getting from some starting point in the network to a destination object. Pathnames begin with the starting point's name, and include every directory name between the starting point and the destination object. A pathname ends with the destination object's name. (*See also* Full pathname and Partial pathname.)

Pipe

A simple way to connect the output of one program to the input of another program, so that each program runs as a sequence of processes.

Pipeline

A series of filters separated by a pipe (|) character. The output of each filter becomes the input of the next filter in the line. The last filter in the line writes to its standard input. (*See also* Filter.)

Print Server

A process that oversees the printing of files submitted to the print queue. It need only run from the node connected to the print device(s).

Process

A program that is in some state of execution; the execution of a computing environment including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and other such data.

Program

Software that can be executed by a user.

Process input window

Window in which you type commands after being prompted.

Process output window

The large window immediately above the process input window. This window displays commands, along with a shell's response to them.

Prompt

A message or symbol displayed by the system to let you know that it is ready for your input.

Regular expression

A string specifier that can help you find occurrences of variables, expressions, or terms in programs and documents. Shell regular expressions are specified by allowing certain characters special meaning to a shell.

Root directory

See Network root directory.

Screen

See Transcript pad.

Search path

The route that a shell takes in searching through various directories for command files. A default search path exists for the BSD shells. You may add other directories of executable files which a shell then looks through on its way to finding a particular command name.

Secondary prompt

A notification to the user that the command typed in response to the primary prompt is incomplete.

Shell

A command-line interpreter program used to invoke utility programs.

Shell command

An instruction you give the system to execute a utility program. (*See also* Shell script.)

Shell metacharacter

Any character that has special meaning to a shell. Asterisks, question marks, and ampersands are a few examples.

Shell script

A file that you create that contains one or more shell commands. A script lets you execute a sequence of commands by entering a single command (the script name). (*See also* Shell command.)

Soft link

A link that points to link text or the pathname of an object (file). (*See also* Link.)

Software

Programs, such as shells and the DM, that allow you to perform various tasks.

Standard input

The standard input of a command is sent to an open file which is normally connected to the keyboard. An argument to a shell of the form **< file** opens the specified file as the standard input, thus redirecting input to come from the file named instead of the keyboard. (*See also* Pipe.)

Standard output

Output produced by most commands is sent to an open file which is normally connected to the printer or screen. This output may be redirected by an argument to a shell of the form **> file** to open the specified file as the standard output. (*See also* Pipe.)

Start-up script

A file that sets up the initial operating environment on your node. This file is also known as a "boot script".

Symbolic link

See Soft link.

System administrator

The person responsible for system maintenance at your site. The account named **root** is the administrative account.

SYSTYPE

A DM environment variable that shows the UNIX system version currently in use. Valid SYSTYPES are **sys5.3** and **bsd4.3**. (*See also* DM environment variable.)

Super-user

See System administrator.

Terminal

See Node.

Transcript pad

A transcript pad contains a record of your interaction with a process. The process output window provides a view of its transcript pad. The term "screen" found in some of our documentation also refers to the transcript pad of the window in which a shell is running.

User account

The system administrator defines a user account for every person authorized to use the system. Each user account contains the name the computer uses to identify the person (user ID), and the person's password. User accounts also contain project and organization names, helping the system determine who can use the system, and what resources they can use. (*See also* User ID and Password.)

User ID

The name the computer uses to identify you. Your system administrator assigns you your user ID. Enter your user ID during the log-in procedure when the system displays the log-in prompt. (*See also* User account.)

Utilities

Programs provided with the operating system to perform frequently required tasks, such as printing a file or displaying the contents of a directory. (*See also* Command.)

Variable

A name that represents a string value. Variables normally set only on a command line are called parameters. Other variables are simply names to which the user or a shell may assign string values.

Wildcards

Special characters that you may use to represent one or more pathnames. (*See also* Shell metacharacter.)

Window

Openings on the screen for viewing information stored in the system. Display management software lets you create several different windows on the screen. Each window is a separate computing environment in which you may execute programs, edit text, or read text. Move the windows on your screen, change their size and shape, and overlap or shuffle them as you might papers on your desk. (*See also* Pads.)

Window legend

The area of a window that displays window status information. For example, the window legend of an edit window contains such information as the pathname of the file you're editing, the letter **I** if the window is in insert mode, and the number of the line at the top of the window. (*See also* Insert mode.)

Working directory

The default directory in which a process creates or searches for objects. (*See also* Current directory.)

Index

Symbols are listed at the beginning of the index. Entries in color indicate task-oriented information.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Using Your BSD Environment*
Order No.: 011020-A00
Date of Publication: July, 1988

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator ___ Student
___ Manager/Professional ___ Novice
___ Technical Professional ___ Other

How often do you use the Apollo system?_____

What additional information would you like the manual to include?_____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc._____
_____
_____

_____
Your Name

Date

_____
Organization

_____
Street Address

_____
City State

Zip

No postage necessary if mailed in the U.S.

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Using Your BSD Environment*
Order No.: 011020-A00
Date of Publication: July, 1988

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator
___ Manager/Professional
___ Technical Professional
___ Student
___ Novice
___ Other

How often do you use the Apollo system?_____

What additional information would you like the manual to include?_____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc._____
_____
_____

_____
Your Name

Date

_____
Organization

_____
Street Address

_____
City State

Zip

No postage necessary if mailed in the U.S.

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Using Your BSD Environment*
Order No.: 011020-A00
Date of Publication: July, 1988

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator
___ Manager/Professional
___ Technical Professional
___ Student
___ Novice
___ Other

How often do you use the Apollo system?_____

What additional information would you like the manual to include?_____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc._____

_____

_____

_____

Your Name

Date

_____

Organization

_____

Street Address

_____

City State

Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Using Your BSD Environment*
Order No.: 011020-A00
Date of Publication: July, 1988

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator ___ Student
___ Manager/Professional ___ Novice
___ Technical Professional ___ Other

How often do you use the Apollo system?_____

What additional information would you like the manual to include?_____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc._____
_____
_____

_____
Your Name

Date

_____
Organization

_____
Street Address

_____
City State

Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

apollo

Using Your BSD Environment

011020-A00

OPERATING SYSTEM

011020-A00