

*Using Your
Aegis
Environment*

011021-A00

apollo

Using Your Aegis Environment

Order No. 011021-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Confidential and Proprietary. Copyright © 1988 Apollo Computer, Inc., Chelmsford, Massachusetts. Unpublished -- rights reserved under the Copyright Laws of the United States. All Rights Reserved.

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.

Apollo and Domain are registered trademarks of Apollo Computer Inc.

ETHERNET is a registered trademark of Xerox Corporation.

Personal Computer AT and Personal Computer XT are registered trademarks of International Business Machines Corporation.

UNIX is a registered trademark of AT&T in the USA and other countries.

3DGMR, Aegis, D3M, DGR, Domain/Access, Domain/Ada, Domain/Bridge, Domain/C, Domain/ComController, Domain/CommonLISP, Domain/CORE, Domain/Debug, Domain/DFL, Domain/Dialogue, Domain/DQC, Domain/IX, Domain/Laser-26, Domain/LISP, Domain/PAK, Domain/PCC, Domain/PCI, Domain/SNA, Domain X.25, DPSS, DPSS/Mail, DSEE, FPX, GMR, GPR, GSR, NLS, Network Computing Kernel, Network Computing System, Network License Server, Open Dialogue, Open Network Toolkit, Open System Toolkit, Personal Supercomputer, Personal Super Workstation, Personal Workstation, Series 3000, Series 4000, Series 10000, and VCD-8 are trademarks of Apollo Computer Inc.

Apollo Computer Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should in all cases consult Apollo Computer Inc. to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF APOLLO COMPUTER INC. HARDWARE PRODUCTS AND THE LICENSING OF APOLLO COMPUTER INC. SOFTWARE PROGRAMS CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN APOLLO COMPUTER INC. AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY APOLLO COMPUTER INC. FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY BY APOLLO COMPUTER INC. WHATSOEVER.

IN NO EVENT SHALL APOLLO COMPUTER INC. BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATING TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF APOLLO COMPUTER INC. HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

THE SOFTWARE PROGRAMS DESCRIBED IN THIS DOCUMENT ARE CONFIDENTIAL INFORMATION AND PROPRIETARY PRODUCTS OF APOLLO COMPUTER INC. OR ITS LICENSORS.

Preface

Using Your Aegis Environment provides detailed information about the Aegis™ environment. This manual describes how to use the system to perform various computing tasks. You should already have read *Getting Started with Domain/OS* (002348), the beginner's guide to using Aegis software on an Apollo® node. If so, you know how to use the keyboard and display, read and edit text, create and execute programs, and request system services using interactive commands. A working knowledge of these tasks is necessary to understand the concepts presented in this more advanced user's guide.

The manual is organized as follows:

- | | |
|------------------|--|
| Chapter 1 | Provides an overview of Domain®/OS. Describes how the system organizes objects in the system naming tree, and how to use pathnames to identify these objects. |
| Chapter 2 | Describes 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. The chapter also describes procedures for changing your password and log-in home directory. |

- Chapter 3** Explains the functions of the default window manager, the Display Manager (DM). Describes how to use DM commands, and shows how to define keys to perform DM functions.
- Chapter 4** Describes how to use the DM to control your node's display. Each section describes a set of related display management tasks and the DM commands you use to perform these tasks.
- Chapter 5** 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 these tasks.
- Chapter 6** Describes the command shell environment that processes Aegis shell commands. The chapter includes information on shell commands, and on controlling command input and output. This chapter also describes the command line parser, and how to use pathname wildcards.
- Chapter 7** Describes how to use shell commands to manage files on the system.
- Chapter 8** Describes how to use shell commands to manage directories on the system.
- Chapter 9** Describes how to use shell commands to manage links on the system.
- Chapter 10** Describes Access Control Lists (ACLs) and how to use them to control access to files and directories.
- Chapter 11** Describes how to write shell scripts using Aegis shell commands, operators, and expressions.

Related Manuals

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. The manual presents user-oriented examples and includes a glossary of important terms.

The *Aegis Command Reference* (002547) contains detailed descriptions of all Aegis commands and utilities. The *Domain/OS Calls Reference* (007196) describes the system calls available to Aegis users. The *Domain Display Manager Command Reference* (011418) contains information about the use of the default Display Management software. Each description in these reference manuals is arranged for quick and easy access, and most provide examples.

For greater detail on using Aegis system calls to do programming tasks, refer to *Programming with Domain/OS System Calls* (005506).

The *Domain/OS Programming Environment Reference* (011010) describes the support tools and utilities available to Aegis 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.

For information on how to create the network environment, protect the network software, create servers, and maintain and troubleshoot the network, see *Managing Aegis System Software* (010852), *Planning Domain Networks and Internets* (009916), and *Managing Domain/OS and Domain Routing in an Internet* (005694).

As a aid to locating on-line information using Domain/Delphi, you should read *Retrieving Information with Domain/Delphi* (011270). To learn more about other Apollo manuals, consult the *Domain Documentation Master Index* (011242) and the *Domain Documentation Quick Reference* (002685).

The `/sys/help` file named `manuals` lists current revisions of all manuals for this software release. To access this file, see the “Getting Help” section on the following page.

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.

You can get more information about how to submit an APR by consulting the *Aegis Command Reference*. Refer to the `mkapr` (make apollo product report) shell command description. You can view the same description online by typing:

```
$ help mkapr
```

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 commands, system calls, and functions, press <HELP>. Then, at the prompt, type the *name* of the command, system call, or function, for which you need more information, as shown in the example below:

```
Help on: name
```

This reads the appropriate file in the `/sys/help` directory, displaying on-line versions of reference material from the *Aegis Command Reference*, the *Aegis Programmer's Reference*, and *Managing Aegis 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.

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.
<i>user-supplied values</i>	<i>Italic words or characters in formats and command descriptions represent values that you must supply.</i>
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.
	A vertical bar separates items in a list of choices.
< >	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 1 Introducing Domain/OS

Overview	1-2
The Naming Tree	1-4
Using Pathnames	1-6
The Working Directory	1-9
The Naming Directory	1-10
The Parent Directory	1-12
Pathname Summary	1-13

Chapter 2 Understanding Startup and Login

Understanding the System at Startup	2-2
Disked Node Startup	2-2
Diskless Node Startup	2-8
Understanding the System at Login	2-14
Logging In	2-19
Logging In to a Default Account	2-19
Changing Your Password	2-20
Changing Your Home Directory	2-21
Logging In to a Domain Server Processor (DSP) .	2-21

Chapter 3 Using the Display Manager

Using DM Commands	3-1
DM Command Conventions	3-3
Using DM Special Characters	3-4
Defining Points and Regions	3-5
Specifying Points on the Display	3-5
Using Keys to Perform DM Functions	3-9
Keyboard Types and Key Definitions	3-10
Key Naming Conventions	3-13
Defining Keys	3-15
Deleting Key Definitions	3-18
Displaying Key Definitions	3-19
Controlling Keys from Within a Program	3-19
Using DM Command Scripts	3-20

Chapter 4 Controlling the Display

Controlling Cursor Movement	4-2
Creating Processes	4-4
Creating a Process with Pads and Windows	4-5
Creating a Process without Pads and Windows ...	4-7
Creating a Server Process	4-7
Controlling a Process	4-8
Stopping a Program or Process	4-9
Suspending and Resuming a Process	4-9
Creating Pads and Windows	4-9
DM Rules for Defining Window Boundaries	4-10
Creating an Edit Pad and Window	4-12
Creating a Read-Only Pad and Window	4-13
Copying a Pad and Window	4-14
Closing Pads and Windows	4-15
Managing Windows	4-16
Changing Window Size	4-17
Moving a Window	4-19

Pushing and Popping Windows	4-20
Changing Process Window Modes	4-21
Defining Default Window Positions	4-24
Responding to DM Alarms	4-25
Moving Pads Under Windows	4-26
Moving to the Top or Bottom of a Pad	4-27
Scrolling a Pad Vertically	4-27
Scrolling a Pad Horizontally	4-29
Saving a Transcript Pad in a File	4-29
Using Window Groups and Window Icons	4-30
Creating and Adding to Window Groups	4-30
Removing Entries from Window Groups	4-31
Making Windows Invisible	4-32
Using Icons	4-32
Setting Icon Default Position and Offset	4-35
Displaying the Members of a Window Group	4-36

Chapter 5 Editing a Pad

Setting Edit Pad Modes	5-2
Setting Read/Write Mode	5-3
Setting Insert/Overstrike Mode	5-3
Inserting Characters	5-4
Inserting a Text String	5-5
Inserting a Newline Character	5-5
Inserting a New Line	5-5
Inserting an End-of-File Mark	5-6
Deleting Text	5-6
Deleting Characters	5-7
Deleting Words	5-7
Deleting Lines	5-8
Defining a Range of Text	5-8
Copying, Cutting, and Pasting Text	5-10
Using Paste Buffers	5-10
Copying Text	5-11
Copying a Display Image	5-13
Cutting Text	5-13

Pasting Text	5-14
Using Regular Expressions	5-15
ASCII Characters	5-16
Beginning of Line (%)	5-16
End of Line (\$)	5-16
Single Character Wildcard (?)	5-17
Expression Wildcard (*)	5-17
Strings and Character Classes	5-17
Escape (@)	5-19
Text Pattern Matching with {expr}	5-20
Searching for Text	5-20
Repeating a Search Operation	5-22
Canceling a Search Operation	5-23
Setting Case Comparison	5-23
Substituting Text	5-23
Substituting All Occurrences of a String	5-24
Substituting the First Occurrence of a String	5-25
Changing the Case of Letters	5-26
Undoing Previous Commands	5-26
Updating an Edit File	5-27

Chapter 6 Using the Aegis Shell

Shell Commands	6-1
Command Line Format	6-2
Standard Command Options	6-4
Command Search Rules	6-5
Special Characters	6-7
Creating and Invoking Shells	6-7
Setting Up the Initial Shell Environment	6-7
Controlling Input and Output	6-8
Reading Input from a File	6-10
Writing Output to a File	6-11
Appending Output to a File	6-11
Redirecting Output to Other Commands	6-12
The Command Line Parser	6-13
Using Query Options	6-14

Reading Data from Standard Input	6-15
Reading Pathnames from Standard Input	6-16
Using Pathname Wildcards	6-17
Running Programs in a Background Process	6-20

Chapter 7 Managing Files

Moving Around the Naming Tree	7-2
Setting the Working Directory	7-2
Setting the Naming Directory	7-3
Creating Files	7-5
Renaming Files	7-6
Copying Files	7-7
Moving Files	7-9
Appending Files	7-10
Printing Files	7-11
Using the prf Command	7-11
Using the Print Menu Interface	7-13
Displaying File Attributes	7-15
Deleting Files	7-16
Copying the Display to a File	7-17
Comparing ASCII Files	7-18

Chapter 8 Managing Directories

Creating Directories	8-2
Renaming Directories	8-2
Copying Directory Trees	8-3
Replacing Directory Trees	8-5
Merging Directory Trees	8-7
Comparing Directory Trees	8-8
Displaying Directory Information	8-9
Deleting Directory Trees	8-10

Chapter 9 Managing Links

Creating Links	9-2
Displaying Link Resolution Names	9-3
Redefining Links	9-3
Renaming Links	9-4
Copying Links	9-5
Deleting Links	9-6

Chapter 10 Controlling Access to Files and Directories

ACL Structure	10-2
The Subject Identifier (SID)	10-2
Access Rights	10-4
Searching Directories and Deleting Objects	10-6
Managing ACLs	10-7
Displaying ACLs	10-7
Editing ACLs	10-8
Rules to Specify ACL Entries	10-10
Adding ACL Entries	10-14
Changing Entry Rights	10-14
Adding Entry Rights	10-15
Deleting Entry Rights	10-16
Deleting ACL Entries	10-16
Setting Required Entries	10-17
Copying ACLs	10-17
Initial ACLs	10-18
Editing Initial ACLs	10-20
Copying Initial ACLs	10-21
Protected Subsystems	10-22
How Protected Subsystems Work	10-22
Creating a Protected Subsystem	10-24
Assigning Protected Subsystem Status	10-25

Chapter 11 Writing Shell Scripts

Creating Your Own Commands	11-1
Creating Scripts	11-2
Passing Arguments to Scripts	11-4
Using Quoted Strings	11-7
Using In-Line Data	11-8
Executing DM Commands from Shell Scripts	11-9
Debugging Shell Scripts	11-10
Using Expressions	11-11
Operands in Expressions	11-12
Mathematical Operators	11-14
String Operators	11-14
Comparison Operators	11-16
Logical Operators	11-16
Shell Variables	11-17
Defining Variables	11-17
Using Shell Variables	11-19
Variable Commands	11-20
Defining Variables Interactively	11-21
Using Active Functions	11-23
Controlling Script Execution	11-25
Using the If Statement	11-27
Using the While Statement	11-28
Using the For Statement	11-30
Using the Select Statement	11-32

Appendix A: Initial Directory and File Structure

Appendix B: Summary of Predefined Standard Key Definitions

Operating Considerations for Multinational Keyboards	B-8
Arrangement of Multinational Keyboard Keys	B-9
Key Interpretation During Service Mode	B-9

Appendix C: Sample Shell Scripts

Script 1: Prompting For and Checking a Target Node Name	C-2
Script 2: Generic Routine Prompting for a Yes or No Answer	C-3
Script 3: Disk Cleanup Utility	C-4
Script 4: Printing a Directory's Most Recent Backup Activity	C-6
Script 5: Resolving Links to Find an Ultimate Pathname	C-9

Appendix D: Composing European Characters

The Compose Function	D-1
Composing European Characters	D-1
Typing European Characters	D-3
Printing Latin-1 Characters	D-3
Restrictions on Using Latin-1 Characters	D-4
Character Compose Sequences	D-4

Figures

1-1	A Simple Domain/OS Network	1-2
1-2	A Sample Naming Tree	1-4
1-3	A Sample Path through the Naming Tree ...	1-7
1-4	A Sample Path Beginning at the Node Entry Directory	1-8
1-5	A Sample Path Beginning at the Current Working Directory	1-10
1-6	A Sample Path Beginning at the Current Naming Directory	1-11
1-7	A Sample Path Beginning at the Parent Directory	1-12
2-1	The Start-Up Sequence for Disked Nodes ...	2-3
2-2	A Sample Start-Up Script	2-7
2-3	The Start-Up Sequence for a Diskless Node .	2-9
2-4	The Start-Up Script Search Sequence	2-14
2-5	The Log-In Sequence	2-15
2-6	A Sample DM Log-In Start-Up Script	2-17
2-7	A Sample DM Start-Up Script	2-19
3-1	Invoking a DM Command Interactively	3-3
3-2	Defining a Display Region	3-8
3-3	Key Names for the Low-Profile Keyboards ..	3-11
4-1	A Process Running the Aegis Shell	4-6
4-2	Creating an Edit Pad and Window	4-13
4-3	Copying a Pad and Window	4-14
4-4	Growing a Window Using Rubberbanding ...	4-18
4-5	Pushing and Popping Windows	4-20
4-6	Process Window Legend	4-22
4-7	Location of Pad Scroll Keys	4-28
4-8	Default Icon for Shell Process Windows	4-33

5-1	The Edit Pad Window Legend	5-2
5-2	Defining a Range of Text with <MARK>	5-9
5-3	Copying Text with the xc -r Command	5-12
6-1	The Aegis Shell Process	6-2
6-2	Shell Command Line Components	6-3
6-3	Sample Shell Start-Up Script	6-8
7-1	The Print Menu	7-13
7-2	Specifying a Filename on the Print Menu ...	7-14
7-3	Comparing Two ASCII Files	7-18
8-1	Sample Directory Tree	8-4
8-2	Copying a Directory Tree	8-5
8-3	Replacing a Directory Tree	8-6
8-4	Two Sample Directories	8-8
8-5	Comparing Directory Trees	8-9
8-6	Deleting a Directory Tree	8-11
10-1	Structure of an ACL Entry	10-2
10-2	Sample ACL Entries	10-3
10-3	Sample Extended ACL Entries	10-4
10-4	Sample ACL Display	10-8
10-5	Initial ACLs for Files and Directories	10-19
10-6	Controlling Access to Protected Subsystem Files	10-23
10-7	Sample of a Protected Subsystem Transcript .	10-27
11-1	Flow of Execution in a Simple Script	11-25
11-2	Flow of Execution with a Conditional Statement	11-26
A-1	The Node Entry Directory (/) and Subdirectories	A-2
A-2	The System Software Directory (/sys)	A-3
A-3	The Display Manager Directory (/sys/dm) ...	A-4
A-4	The Network Management Directory (/sys/net)	A-5
B-1	Multinational Keyboard Numeric Keypad	B-9

Tables

1-1	Pathname Symbols	1-9
2-1	Node DM Start-up Script Files	2-7
2-2	Node Log-In Start-Up Script Files	2-17
3-1	Ranges for Coordinate Values	3-7
3-2	Default Mouse Key Functions	3-10
3-3	Key Definition Filenames	3-12
3-4	Key Naming Conventions	3-14
4-1	Cursor Control Commands	4-3
4-2	Commands for Creating Processes	4-5
4-3	Commands for Controlling a Process	4-8
4-4	Commands for Creating Pads and Windows	4-10
4-5	Commands for Closing Pads and Windows	4-15
4-6	Commands for Managing Windows	4-17
4-7	Process Window Modes	4-22
4-8	Commands for Moving Pads	4-26
4-9	Commands for Controlling Window Groups and Icons	4-30
4-10	Window Paste Buffers	4-37
5-1	Commands for Setting Edit Modes	5-2
5-2	Commands for Inserting Characters	5-4
5-3	Commands for Deleting Text	5-7
5-4	Commands for Copying, Cutting, and Pasting Text	5-10
5-5	Commands for Searching for Text	5-21
5-6	Commands for Substituting Text	5-24
6-1	Standard Aegis Shell Command Options	6-4
6-2	I/O Control Characters	6-10
6-3	Command Line Parser Options	6-14
6-4	Command Query Responses	6-15
6-5	Summary of Pathname Wildcards	6-17

7-1	Commands for Setting the Working and Naming Directory	7-2
7-2	Commands for Managing Files	7-4
7-3	Shell Commands Submenu Items	7-14
8-1	Commands for Managing Directories	8-1
9-1	Commands for Managing Links	9-1
10-1	Access Rights for Files and Directories	10-6
10-2	Summary of Commands for Editing ACLs ...	10-10
10-3	Class Names for Commonly Assigned Rights .	10-13
10-4	Summary of Commands for Editing and Copying Initial ACLs	10-19
10-5	Options for Copying Initial ACLs	10-21
11-1	Shell Parsing Operators	11-3
11-2	Script Verification Options	11-10
11-3	Summary of Expression Operators	11-13
11-4	Rules for Assigning Variable Types	11-18
11-5	Variable Commands	11-21
B-1	Controlling the Cursor	B-2
B-2	Creating Processes	B-3
B-3	Controlling Processes	B-3
B-4	Creating Pads and Windows	B-3
B-5	Closing Pads and Windows	B-4
B-6	Managing Windows	B-4
B-7	Moving Pads	B-5
B-8	Controlling Window Groups and Icons	B-6
B-9	Setting Edit Pad Modes	B-6
B-10	Inserting Characters	B-6
B-11	Deleting Text	B-7
B-12	Copying, Cutting, and Pasting Text	B-7
B-13	Commands for Searching for Text	B-8
B-14	Commands for Substituting Text	B-8
D-1	Compose Sequences for Latin-1 Characters .	D-4

Chapter 1

Introducing Domain/OS

Domain/OS is an operating system that 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, 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 runs on a physical network in which member nodes can load data from the network into memory just as they would from their own disks. 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.

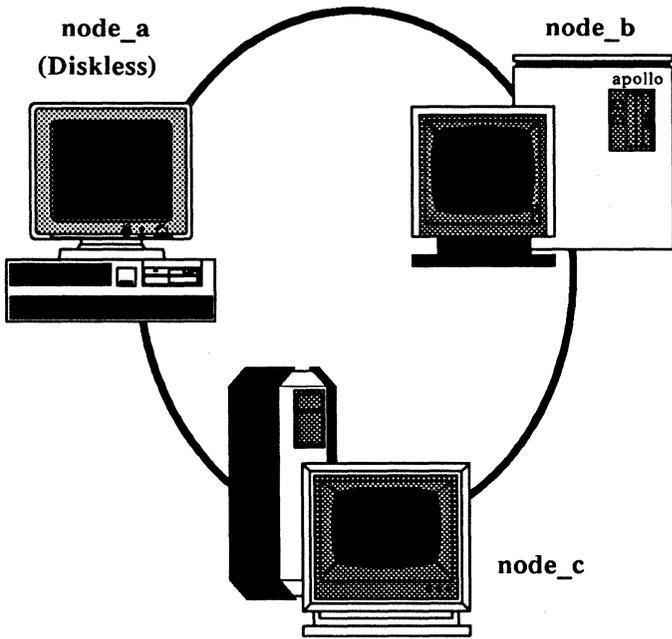


Figure 1-1. A Simple Domain/OS Network

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. The Aegis environment uses the Aegis shell, which 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 the shell program 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.

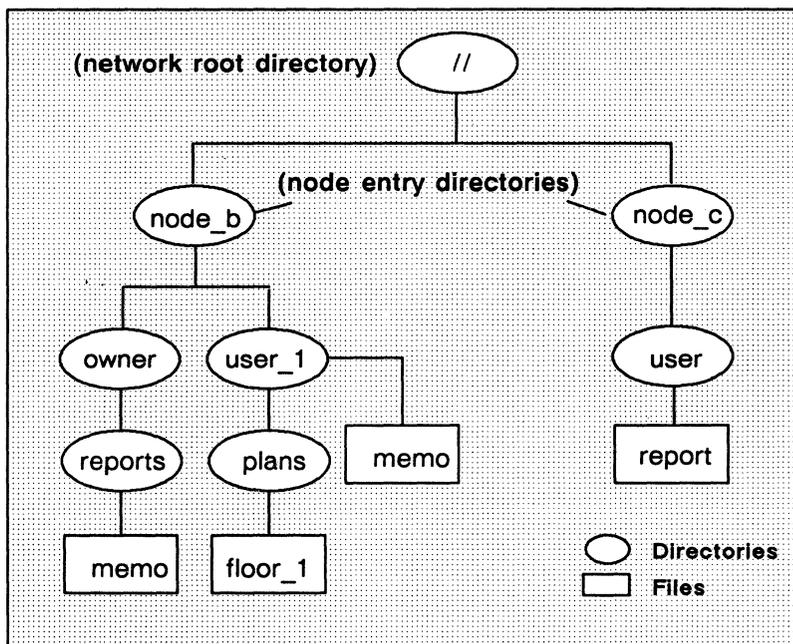


Figure 1-2. A Sample Naming Tree

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

Entry directories contain one or more upper-level directories. An **upper-level directory** is one level below the entry directory and normally serves as the main directory for a branch of related objects. For example, the **/sys** directory that we supply is an upper-level directory that contains many of the system objects that make up the operating system. (Appendix A illustrates how the system organizes the software we supply with your node.) An upper-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 upper-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 upper-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**.

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 upper-level directories. An upper-level directory serves as the main directory for a group of related 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're added to the network. To catalog new nodes, use the shell command **ctnode** (catalog node) described in the *Aegis Command Reference*.

Some network sites use **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 Aegis System Software* describes **ns_helper** and explains how to catalog nodes to update the network root directory.

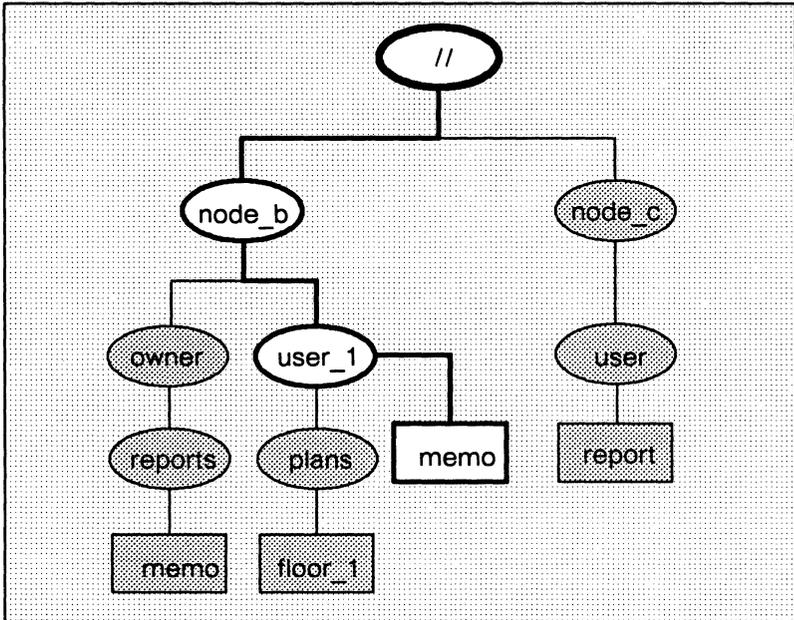


Figure 1-3. A Sample Path through the Naming Tree

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:

```
$ dlf /user_1/memo
```

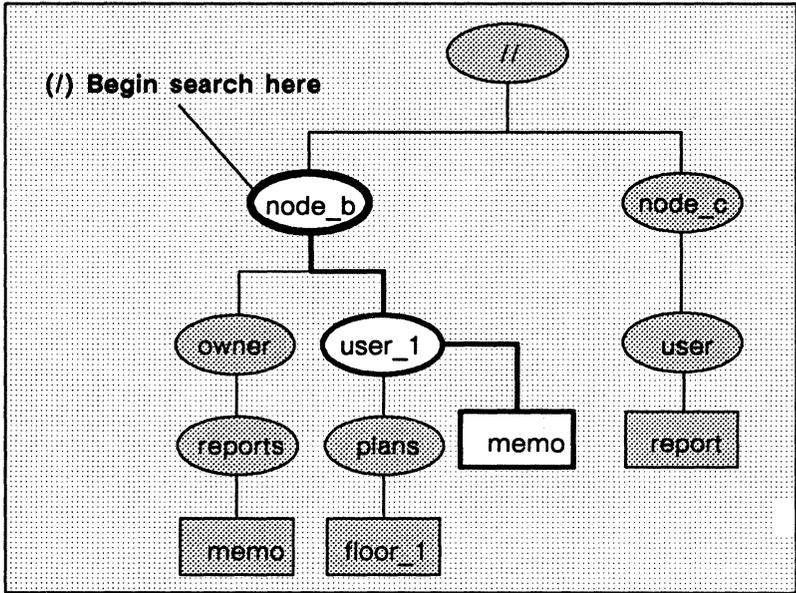


Figure 1-4. A Sample Path Beginning at the Node Entry Directory

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 upper-level directory, **user_1**.
3. Stop at the file named **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	Working directory
~	Naming directory
..	Parent directory

The Working Directory

If you specify a pathname without a symbol preceding it, 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. 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. (Chapter 2 describes your home directory and how to change it.) The system uses this directory as your working directory unless you change it to another directory. (Chapter 7 describes how to change your working directory.)

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

```
$ dlf 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 deletes the file **memo**.

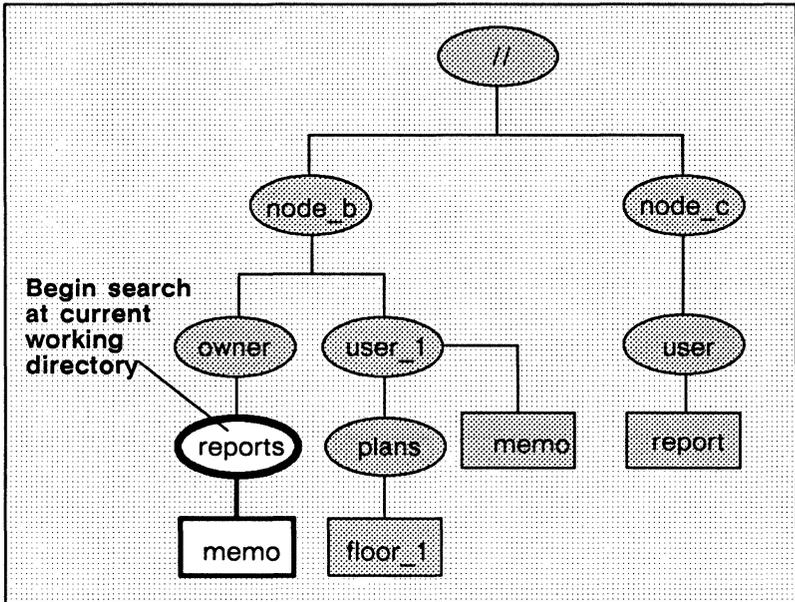


Figure 1-5. A Sample Path Beginning at the Current Working Directory

You'll notice in Figure 1-5 that 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 delete this file instead. A pathname that starts at the working directory functions differently depending on the directory currently being used as the working directory.

The Naming Directory

If you precede a pathname with the tilde and slash symbols in combination (`~/`), the system starts its search at a location in the naming tree called the naming directory. Like the working directory, each process has a naming 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 the **naming directory** of that process to the home directory name designated in your user account. The system uses this directory as your naming directory unless you change it to another directory.

The following command deletes the file **memo** in the directory **reports**, in the current naming directory:

```
$ dlf ~/reports/memo
```

Let's assume that the current naming directory is the upper-level directory **owner**. As shown in Figure 1-6, the pathname directs the system to:

1. Start at your node's naming directory, **owner**.
2. Follow the path through the directory **reports**.
3. Stop at the file, **memo**.

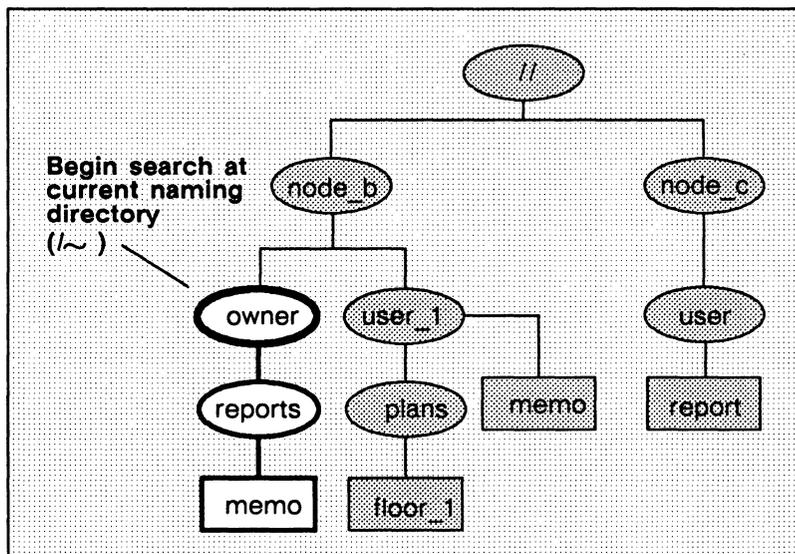


Figure 1-6. A Sample Path Beginning at the Current Naming Directory

Like pathnames that use the current working directory, pathnames starting at the naming directory work differently depending on the directory currently being used as the naming 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 delete the file `memo` in the directory `user_1`:

```
$ dlf ../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 deletes the file `memo`.

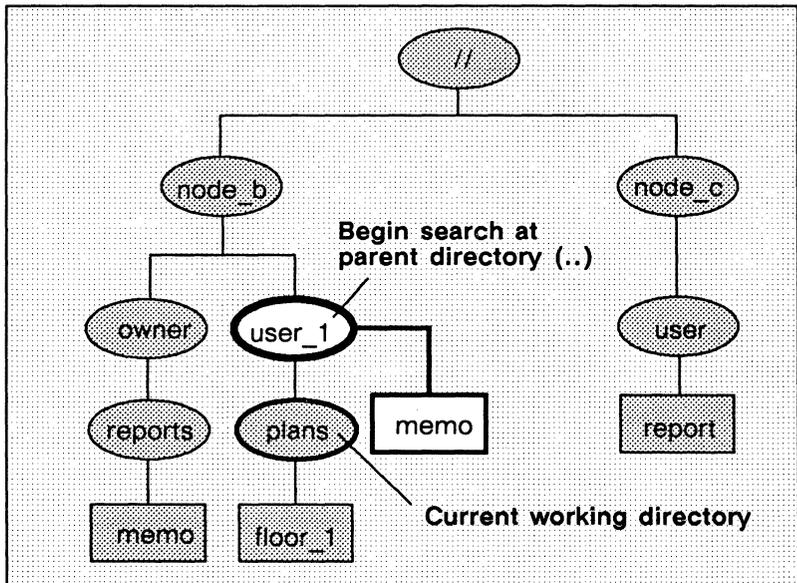


Figure 1-7. A Sample Path Beginning at the Parent Directory

Pathname Summary

You now know how to use pathnames to point to objects in the system naming tree. The examples in this section 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'll find that many of the objects that make up the operating system are referred to by their pathnames. For example, Chapter 2 describes many of the objects the system uses at startup and login. Appendix A illustrates how the system organizes the system software supplied with your node; system objects are referenced by their pathnames. By understanding which objects the system uses and where they're located, you'll better understand how these objects work together to make up a functioning system.



Chapter 2

Understanding Startup and Login

Each time you start up a node and log in to it, the system executes several 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 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 and log-in 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 2-1 shows the start-up sequence on a disked node.

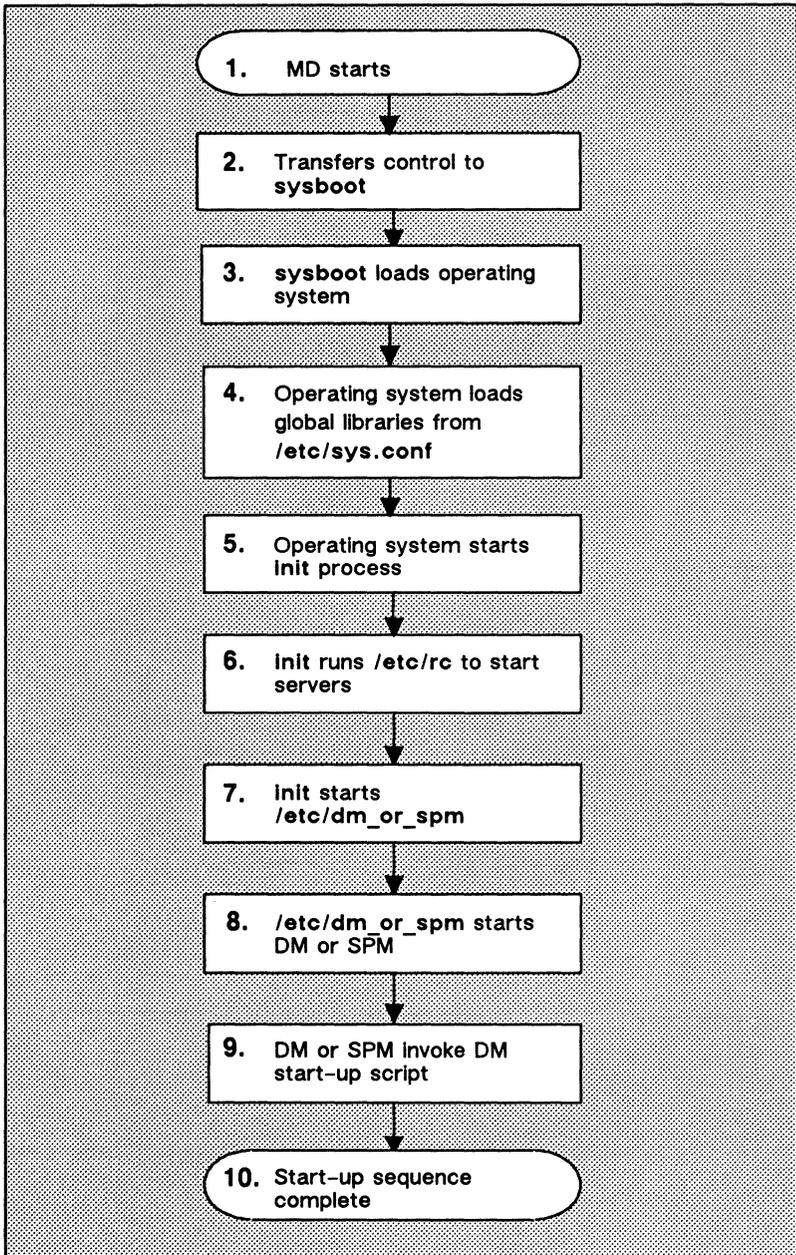


Figure 2-1. The Start-Up Sequence for Disked Nodes

The descriptions that follow explain each step in the start-up sequence shown in Figure 2-1.

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 (e.g., **Aegis**), and one for specifying the **SYSTYPE** variable (**bsd4.3**, **sys5.3**). If **SYSTYPE** is not set, the **Aegis** environment is assumed, and the the default log-in shell is **/com/sh** (**Aegis** shell).

6. The **init** process runs the **/etc/rc** script to start the server programs. 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 server programs that must be invoked by the system administrator. 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, **rc.local** and **rc.user** (run as "user"). The **rc** scripts contain commands that start various server programs. 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, and the **print server**

(**prsvr**) program runs peripheral printers. For a description of these and all of the Domain/OS server programs, see *Managing Aegis System Software*.

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

- To start Aegis servers such as **netman**, **prsvr**, or **mbx_helper**, you can edit the `/etc/rc.user` file and remove the pound sign (#) from the command line that invokes the server.
- To start up other server programs, such as 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 a `/etc/daemons/llbd` file (it doesn't matter what's in the file, as `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 you shut down and restart your node. (See your node's operating guide for node startup and shutdown procedures.)

6. The `/etc/init` program reads the file `/etc/tty`s (which is normally a link to the file `'node_data/etc/tty`s) and starts the `/etc/dm_or_spm` program associated with the display and listed in the file. Any programs started by `/etc/tty` inherit the SYSTYPE value specified in the `/etc/envron` file. (If no SYSTYPE is specified, Aegis is assumed to be the correct environment.) Other lines in the `etc/tty`s file contain directives that start **getty** on the tty lines for the node; see the `/etc/tty`s file for further information.
7. The `/etc/dm_or_spm` program starts either:
 - **Display Manager (DM)** on nodes with displays.
 - **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 (see *Managing Aegis System Software*.)

8. The DM or the SPM executes a start-up file that sets up the initial operating environment on your node. Table 2-1 lists the different files used at startup. As shown in Table 2-1, the system chooses which file to execute according to the type of node.

All of the DM start-up script files listed in Table 2-1 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, `'node_data` points to the `/sys/node_data.node_id` directory 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 2-1. 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.191
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 2-2 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 2-2. A Sample 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.

9. 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.

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 slightly different.

Figure 2-3 presents a flowchart showing the start-up sequence for a diskless node. The descriptions that follow explain each step in the diskless node start-up sequence shown in Figure 2-3. 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.

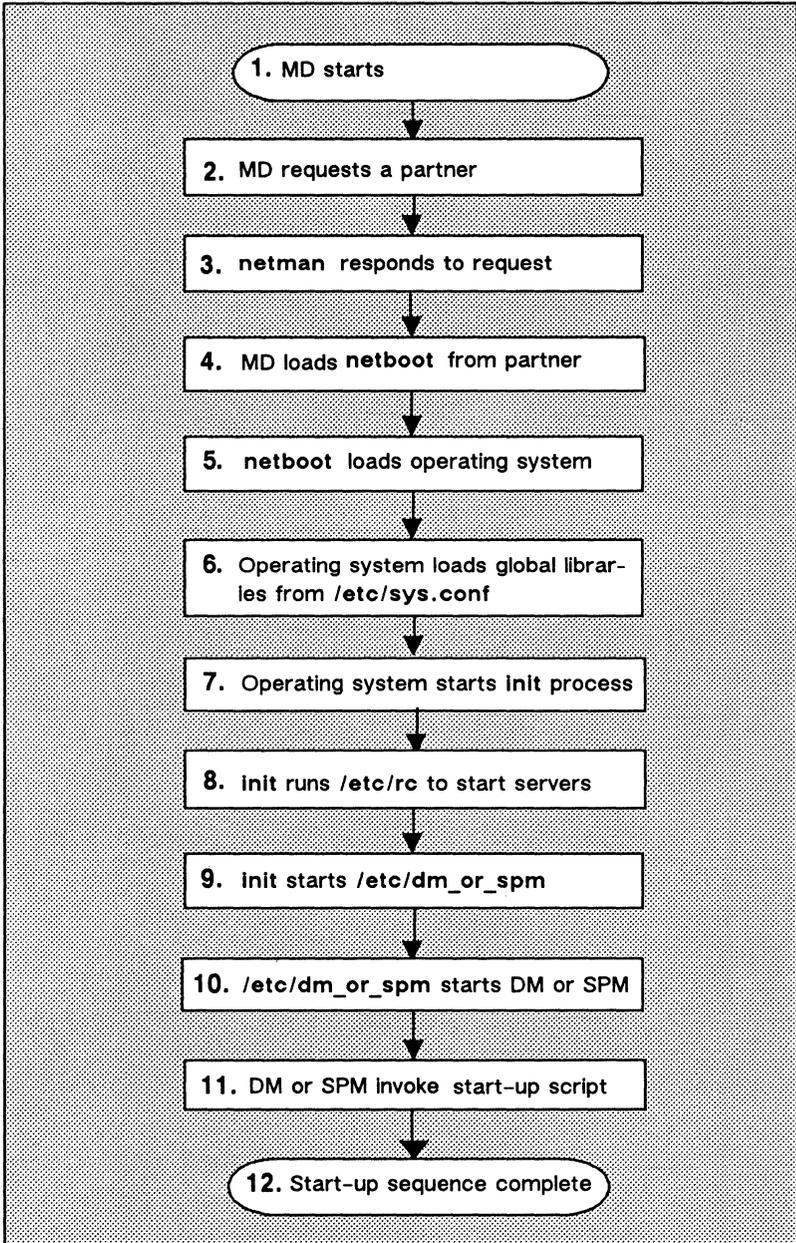


Figure 2-3. The Start-Up Sequence for a Diskless Node

1. When you power on your node in normal mode (by following the instructions in your 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 Aegis System Software*.

4. Once the diskless node finds a partner, the MD copies the **netboot** program from `/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 (Aegis, SysV, BSD), and one for specifying the `SYSTYPE` variable (note that only `bsd4.3` and `sys5.3` are valid here; if no `SYSTYPE` is set, the system assumes that you are an Aegis user). 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 server programs that must be invoked by the system administrator. 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` (run as “user”) and `/etc/rc.local`. The `rc` scripts contain commands that start various server programs. 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 Aegis System Software*.

If you want your node to automatically start any server programs (daemons), two methods are available. The method you use depends on the types of servers you wish to run.

- To start Aegis servers such as `netman`, `prsvr`, or `mbx_helper`, you can edit the `/etc/rc.user` file and remove the pound sign (`#`) from the command line that invokes the server.
- To start up other server programs, such as 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 a `/etc/daemons/llbd` file

(it doesn't matter what's in the file, as `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` (which 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 Aegis System Software*.)
11. The DM or the SPM executes a start-up file that sets up the initial operating environment on your node. Table 2-1 lists the different files used at startup. As shown in Table 2-1, 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 the diskless node points to a directory by the name of `/sys/node_data.node_id` 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 2-2 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 2-4 summarizes this search.

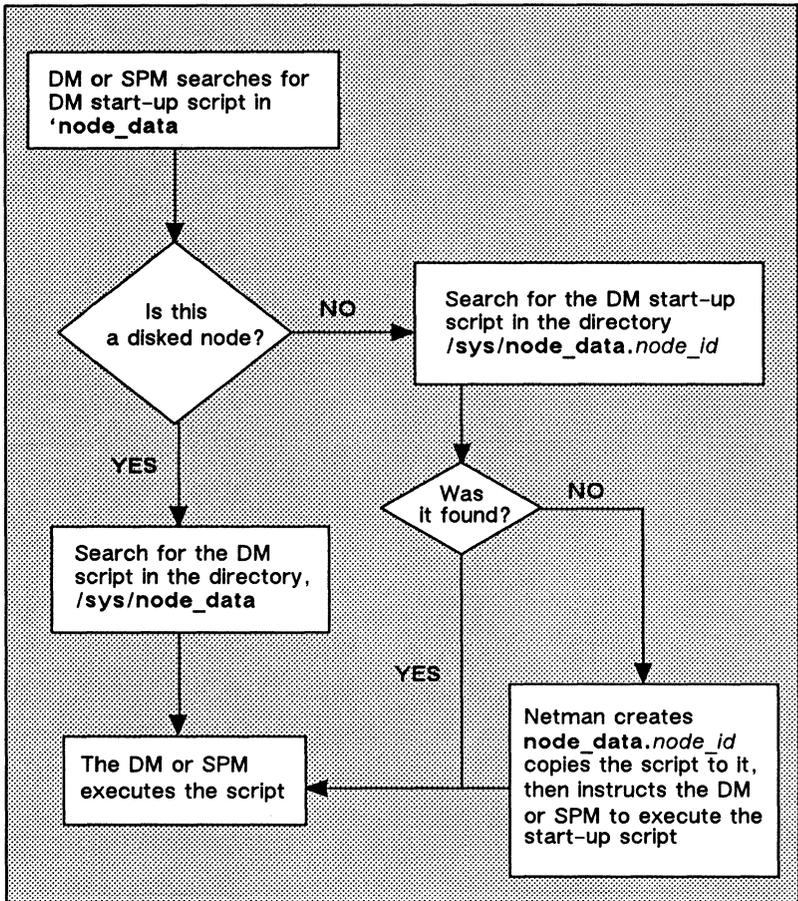


Figure 2-4. The Start-Up Script Search Sequence

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 2-5 shows the log-in sequence for a node.

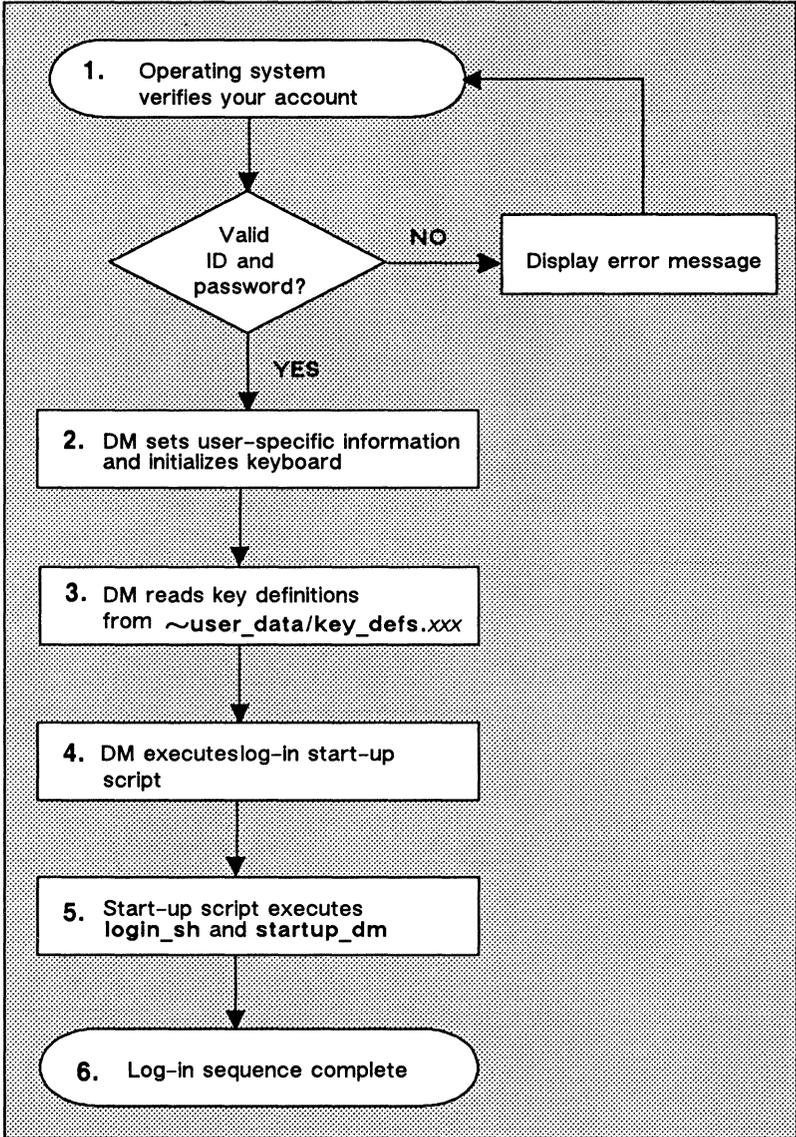


Figure 2-5. The Log-In Sequence

The descriptions that follow explain each step in the log-in sequence shown in Figure 2-5.

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 Aegis 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 (if there is no SYSTYPE specified, the Aegis environment is assumed); 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. If your environment is Aegis, the DM loads both `std_keys.basic` and `std_keys`.
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. Chapter 3 describes key definition files further.
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 2-2. The system chooses the log-in start-up file according to the type of node you are using. 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 2-2. 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

- As shown in Figure 2-6, 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 adding 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 2-6. 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 Aegis shell looks for a script named `~/user_data/sh/login`.

- 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 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 `~/user_data/sh/login` file containing the commands. Note that this file is only executed upon login (by a log-in shell). If you have commands that you wish to execute every time you start a new Aegis shell, create a file named `~/user_data/sh/startup`. For more information about shell start-up files, see Chapter 6.

The last line in the sample script shown in Figure 2-6 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 2-7 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 unindenting:
#
kd F4 t1;s/%      // ke
kd F5 t1;s/%/    / ke
# Set tab every 5 spaces:
#
ts 5 -r
# Set window default location
(0,770)dr;(600,110) wdf1
# Build an Aegis shell window
#
(0,500)dr;(799,955) cp /com/sh

```

Figure 2-7. 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, in Figure 2-6, the **cmdf** command 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 and log-in home directory, and log in to a Domain Server Processor (DSP).

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** as shown on the following command line:

```
login: user
```

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 by using the **chpass** (change password) or the **edrgy** (edit registry) commands. Use the **chpass** command as follows:

```
$ chpass new_password
```

After you specify your new password and press <RETURN>, the system prompts you to verify your new password (to ensure that you entered it correctly). At the prompt, type the new password again and press <RETURN>.

Use the **edrgy** command as follows:

```
$ /etc/edrgy  
edrgy => change username -p new_password  
edrgy => quit
```

When you invoke **edrgy**, it enters interactive mode. For more information about the interactive command options for **edrgy**, see the *Aegis Command Reference*.

Use the new password the next time you log in. If you want to maintain a secure account, avoid using obvious passwords such as your username or your initials. If security is not a high priority, you can use a blank password. (Note, however, that blank passwords violate system security.) To change your password to a blank, specify a space in quotation marks. To enter a blank password when you log in, just press <RETURN>.

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 and naming directories to your home directory. You can change your home directory by using the **chhdir** (change home directory) or the **edrgy** commands. Use the **chhdir** command as follows:

```
$ chhdir new_pathname
```

Use the **edrgy** 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 file **account** in your site registry directory. This file contains information about your account, such as your username, password, and home directory. By updating the **account** file, the system stores your new home directory for logging in later.

See *Managing Aegis System Software* for more information about the **account** file and system registries; see the *Aegis Command Reference* for more information about the **chhdir** and **edrgy** commands.

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 while you sit at a user node in the network.

For a complete description of the procedure for logging into a DSP, see the operating guide for your particular node model.



Chapter 3

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 4 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 specify a position 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 they appear in the script. 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 3-1 illustrates the interactive procedure for invoking the `wc` command to delete a window.

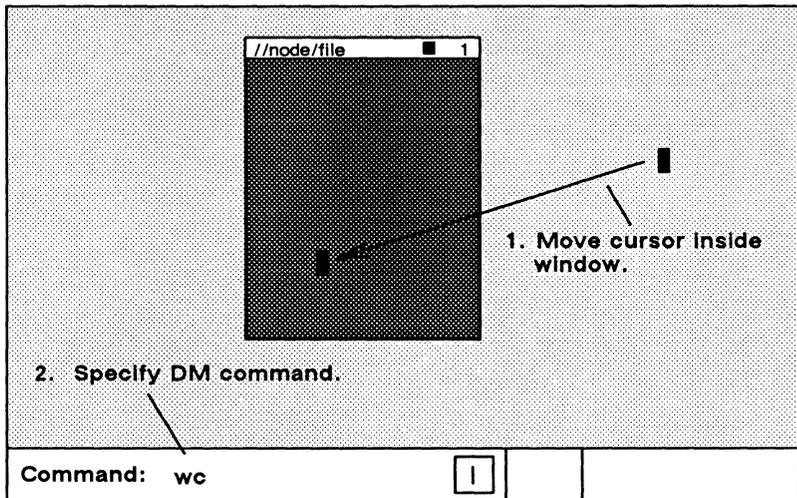


Figure 3-1. Invoking a DM Command Interactively

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 at-sign 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 precede the character with an @. 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 4), 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** 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 4 for more information on how the DM defines window boundaries.)

Specifying Points on the Display

If you choose not to indicate a 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 5.

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 outer pair of brackets are part of the format. The inner pairs of brackets indicate line and column numbers are optional (i.e., either one can be omitted). 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 5 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 3-1 shows the ranges for the coordinate values.

Table 3-1. 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. Note that 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 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. Thus, to move the cursor to line 75, column 5 in an edit pad, specify this 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 3-2.

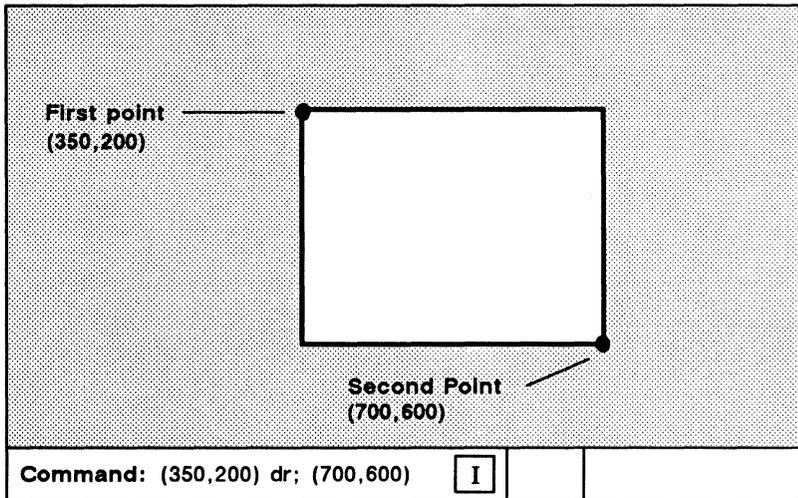


Figure 3-2. Defining a Display Region

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 indicate a point with the cursor. Press `<MARK>` to invoke the `dr` command, which marks the first point. To define a region using the cursor, perform the following tasks:

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.

Command: `(350,200) dr; (700,600) cv report_file`

|
|
└──────────────────┘
└──┘
region
command

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 "short-hand" 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 earlier 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 3-2 describes the default mouse key functions.

Table 3-2. 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 keyboards

Low-Profile type keyboards (shown in Figure 3-3) 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.

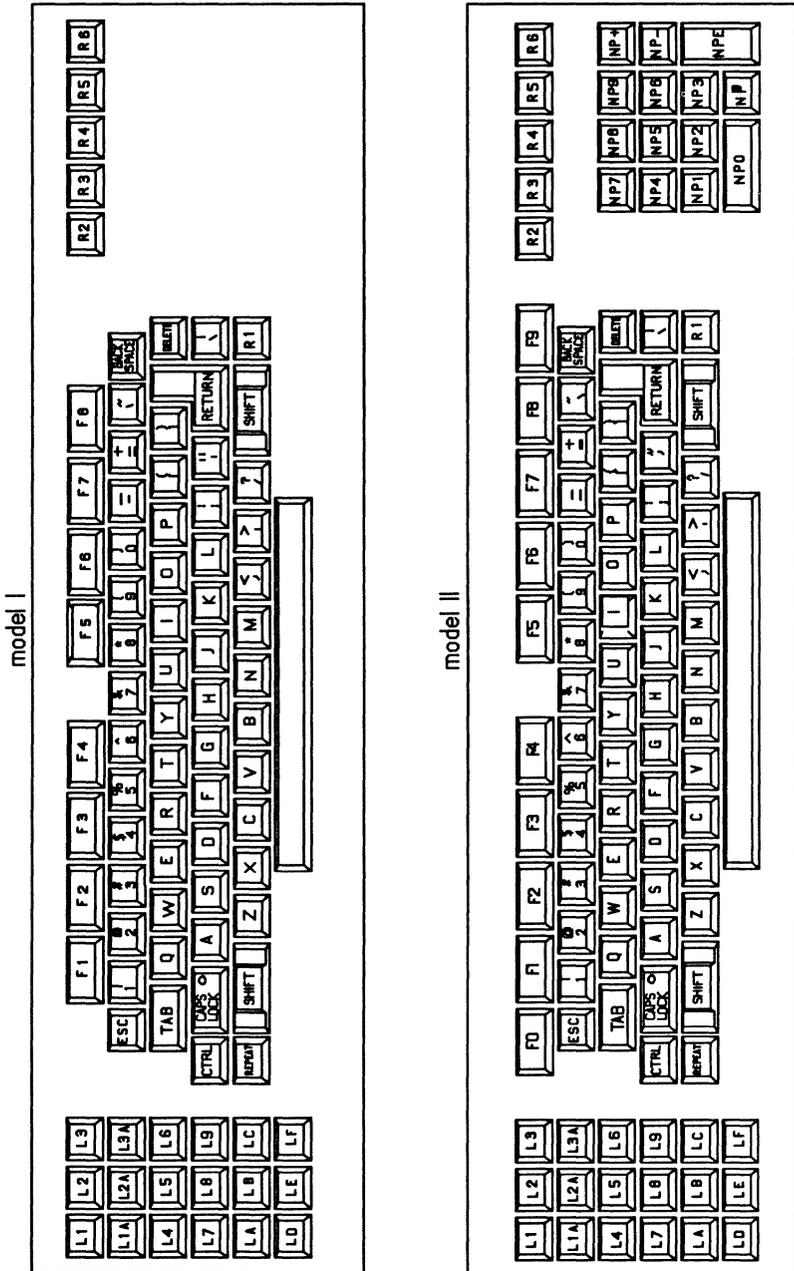


Figure 3-3. Key Names for the Low-Profile Keyboards

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 appear 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 D for information about the compose function.

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

Table 3-3. Key Definition Filenames

Keyboard	Key Definition File
Low-Profile Model I	<code>/sys/dm/std_keys2</code>
Low-Profile Model II	<code>/sys/dm/std_keys3</code>
Multinational Keyboard	<code>/sys/dm/std_keys3x</code> (<i>x</i> 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_keys3x`, where *x* represents 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 3-3 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`).

After you log in, if you find that the predefined keys do not work as described in this manual, you can execute the appropriate `std_keysn` file to set up the proper default key definitions for your keyboard. For example, to set up the predefined key definitions for the Model II keyboard, specify the following in the DM input pad:

Command: `cmdf /sys/dm/std_keys3`

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 next two 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 3-3 shows the names and locations of the keys on both the Low-Profile type keyboards.

NOTE: The Multinational numeric keypad keys do not have the same names as the standard Low-Profile Model II keypad keys. Figure B-1 in Appendix B shows the names and locations of the of the numeric keypad keys on the Multinational keyboard.

For example, the CUT/COPY function key (whose special name is 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, referred to as 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 the key performs when released. The name L1AC, referred to as the key's control key sequence name, identifies the function when pressed along with <CTRL>. Table 3-4 describes the key naming conventions you should use when defining keys.

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 3-4. 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 Shifted 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 3-4. 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.
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.

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 2); 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_keysn** (see Table 3-3).

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. The "Controlling Keys from Within a Program" section 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 3-3 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 following command defines the program function key, F1, to move the cursor to the end of the previous line in a window:

```
kd F1 au;tr ke
```

command *key_name* *definition*

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:

```
kd F3 kd ^X es 'This is a test' ke@; pv ke
```

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** command to scroll the current pad one line. (Chapter 4 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'

The *prompt* argument 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, the READ key (R3) has the following default key definition:

kd R3 cv &'read file: ' ke

Whenever you press <READ>, the DM displays the prompt “read file: ” in the DM input pad and moves the cursor next to it. When you respond to the prompt by typing the name of a file and pressing <RETURN>, the DM replaces &'read file: from the key definition with your response. In this way, the *cv* command opens the file you specify. (Chapter 4 describes the *cv* command.)

NOTE: When you define keys in scripts, you must precede the input request character (&) with the escape character (@).

When you enter a response to a prompt, the DM remembers the response you typed. So, the next time you press the key, the DM automatically displays the previous response next to the prompt. (This is why <READ> and <EDIT> offer the names of 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 @

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 you use **edfont**, 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 4).

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 pathname of the script file with the DM command **cmdf** (command file) as follows:

cmdf *pathname*

The start-up scripts discussed in Chapter 2 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. See the *Aegis Command Reference* for further information about the **cmdf** command.



Chapter 4

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 by entering 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.

Chapter 3 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 indicate the location where you want a DM command to operate, or you can move the cursor into the DM input window and then type the name of a command.

This section summarizes the DM commands and control key sequences used to control cursor movement. Table 4-1 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 3 for more information on key names and defining keys. This note applies to all command summary tables in this chapter.

Table 4-1. 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	<TAB>
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/L

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 10 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. You can also 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 **server programs** that provide you, or a program, with access to some service, such as the use of a peripheral device. Server programs 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 4-2 summarizes the commands used to create processes.

Table 4-2. Commands for Creating Processes

Task	DM Command	Predefined Key
Create new process, pads, and windows	<code>cp pathname</code>	None
Create new process without pads and windows	<code>cpo pathname</code>	None
Create a server process	<code>cps pathname</code>	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] pathname [arguments]
```

The *region* argument specifies the coordinates of the process window and *pathname* 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.bin`. The program prompts for program input and displays its output to the process's transcript pad.

```
cp -n counter /horace/progs/counter.bin
```

The `-n` option assigns the process the name `counter`. When `counter.bin` 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 the 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 an Aegis shell, type the following:

Command: `cp /com/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 4-1 shows a process running the Aegis shell.

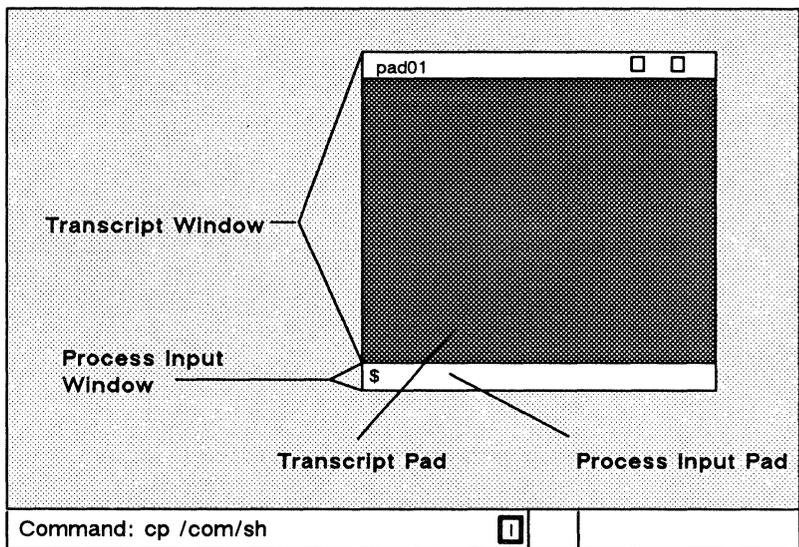


Figure 4-1. A Process Running the Aegis Shell

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

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 pathname [options]
```

The *pathname* 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 off, the process stops. *Managing Aegis 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 **cps** from the DM input window.

Creating a Server Process

You can create a 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 pathname [options]
```

The *pathname* 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 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**) during start-up. (Chapter 2 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 either stop it, suspend it, or restart it. Table 4-3 summarizes the DM commands used to control processes.

Table 4-3. Commands for Controlling a Process

Task	DM Command	Predefined Key
Quit, stop, or blast a process	dq [-b -s -c n]	CTRL/Q
Suspend execution of a process	ds	None
Resume execution of a suspended process	dc	None

Stopping a Program or Process

To stop a program or an entire process, use the **dq** (debug quit) command in the following format:

```
dq [options]
```

To stop a program, position the cursor inside the window of the process and either press CTRL/Q or specify the **dq** command without any options. Either operation will generate a normal quit fault, which interrupts the execution of the current program and returns the process to the calling program (usually the shell).

To stop an entire process, position the cursor inside the window of the process. Then, specify the following DM command:

```
dq -s
```

This command stops the current process and closes any open streams, files, and pads. To delete the remaining window, move the cursor inside the window and press <EXIT>.

Suspending and Resuming a Process

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

To interrupt 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.

Creating Pads and Windows

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

Table 4-4. Commands for Creating Pads and Windows

Task	DM Command	Predefined Key
Create an edit pad and window	<i>ce pathname</i>	<EDIT> (R4)
Create a read-only window	<i>cv pathname</i>	<READ> (R3)
Create a copy of an existing pad and window	<i>cc</i>	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 3.)

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 for more information). The following list 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 closely as possible to the given cursor position.
- Move** Select the 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 pathname
```

The *pathname* argument specifies the pathname 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 5 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, and press `<RETURN>` as shown in Figure 4-2.

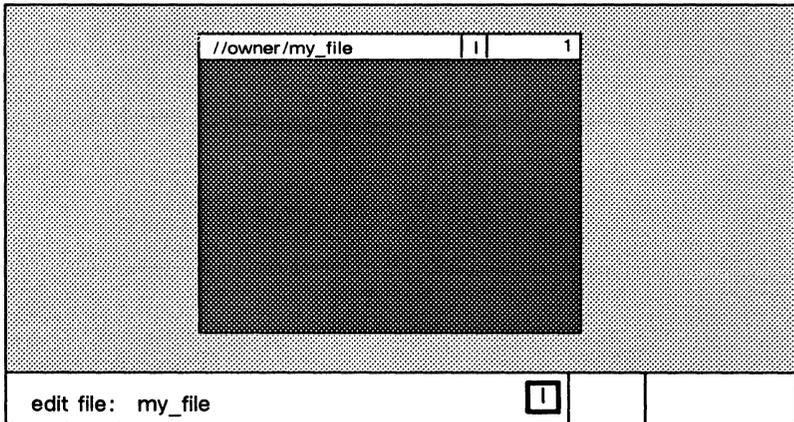


Figure 4-2. Creating an Edit Pad and Window

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 pathname
```

The *pathname* argument specifies the pathname 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 either press CTRL/M or specify the DM command `ro` (set read/write mode) to change the read-only pad to an edit pad. Chapter 5 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>`, which works in a manner similar to `<EDIT>`.

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 4-3 illustrates how to use the `cc` command to copy a pad and window.

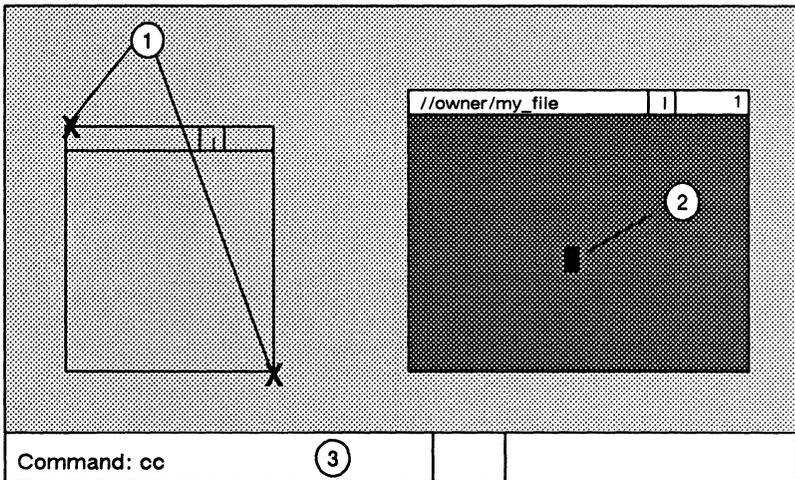


Figure 4-3. Copying a Pad and Window

The numbers in Figure 4-3 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 3 describes how to use `dr` and `<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 4-5.

Table 4-5. Commands for Closing Pads and Windows

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

To delete (quit) a read-only or edit pad and associated windows, position the cursor inside the window and either press <ABORT> (on low-profile keyboards only), press CTRL/N, 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 with "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> (for low-profile keyboards only), press CTRL/Y, 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 4-6 summarizes the window control commands.

Table 4-6. Commands for Managing Windows

Task	DM Command	Predefined Key
Change window size	wg	CTRL/G
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]	CTRL/S
Set autohold mode	wa [-on -off]	None
Scroll and autohold mode	wa; ws	CTRL/A
Set hold mode	wh [-on -off]	<HOLD> (R6)
Define position of default window <i>n</i>	wdf [<i>n</i>]	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 4-4, the **wge** command displays a flexible border, or rubberband, that changes as you move the cursor to enlarge or shrink the window. The position of the rubberband shows you the size and shape the window will become when you complete the operation.

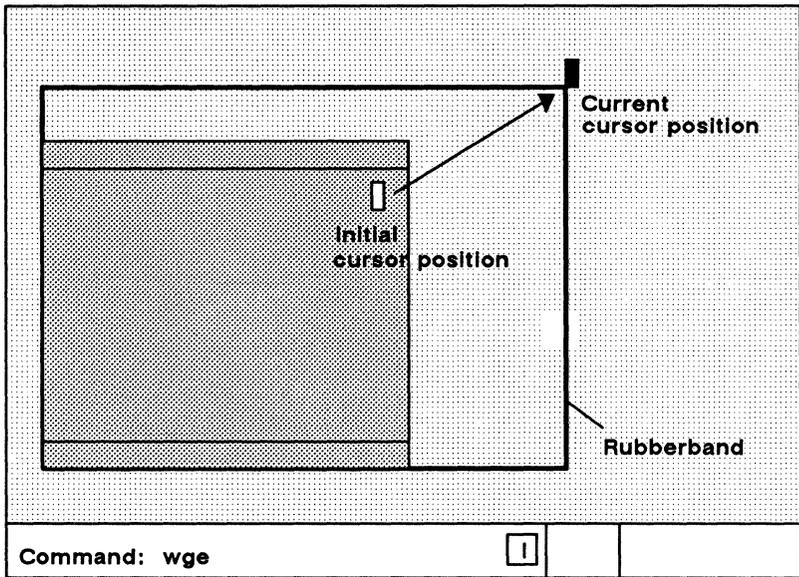


Figure 4-4. Growing a Window Using Rubberbanding

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 enter 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 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 4-5 shows how to push and pop windows.

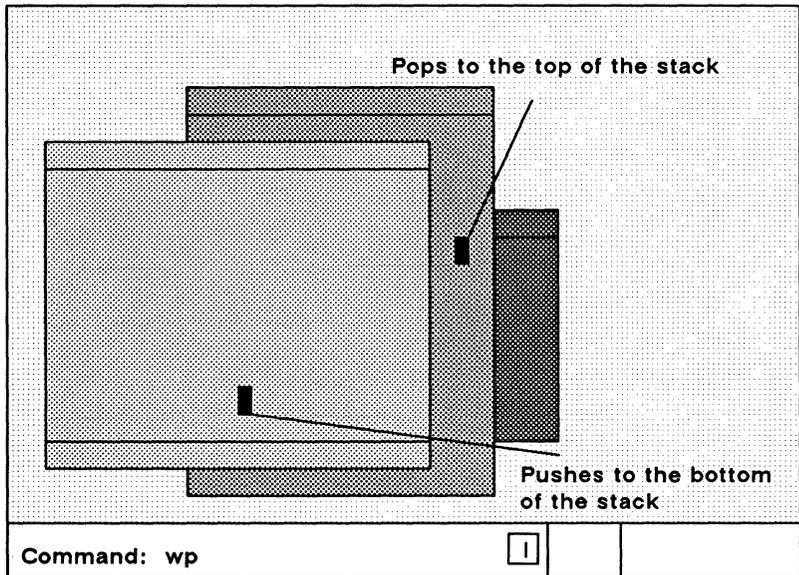


Figure 4-5. Pushing and Popping Windows

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), typing CTRL/P, or 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 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 4-7 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 4-7. 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 4-6 shows the mode indicators and other components that make up the process window legend.

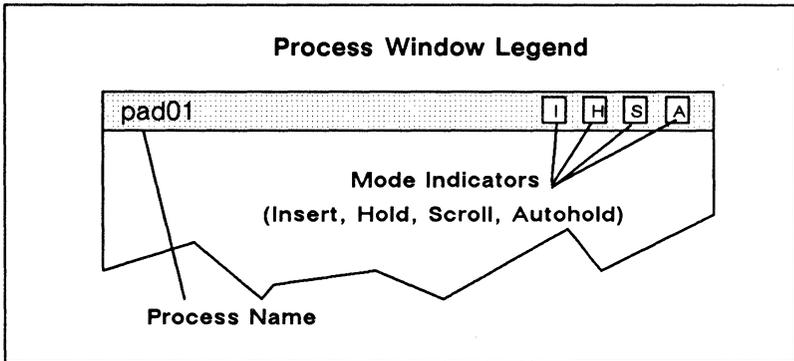


Figure 4-6. Process Window Legend

By default, the window legend displays the letter “I” indicating 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. You can also toggle scroll mode on or off by pressing **CTRL/S**.

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>.

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:

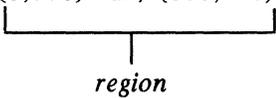
```
[region] wdf [n]
```

The *region* argument specifies the position that the window will occupy on the screen (see the “Specifying Points on the Display” section in Chapter 3), and *n* specifies the identification 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.

```
(0,770) dr; (600,110) wdf 4
```



The diagram shows a horizontal line with vertical end caps at each end, positioned under the coordinates "(0,770) dr; (600,110)". A vertical line extends downwards from the center of this horizontal line to the word "region".

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" section in Chapter 2.

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. If you wish to respond to an alarm, enter either the **aa** or **ap** command.

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 4-8 summarizes the pad control commands.

Table 4-8. 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 <i>n</i> pages	pp [-] <i>n</i>	<div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 2px; text-align: center;">↑</div> <div style="border: 1px solid black; padding: 2px; text-align: center;">↓</div> </div> (LD, LF)
Move pad <i>n</i> lines	pv [-] <i>n</i>	SHIFT/ ↑ (L8S) SHIFT/ ↓ (LES)
Move pad <i>n</i> characters	ph [-] <i>n</i>	<div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 2px; text-align: center;">←</div> <div style="border: 1px solid black; padding: 2px; text-align: center;">→</div> </div> (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 to scroll. A positive *n* scrolls the pad up *n* pages; a negative *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 4-7 shows the location of these keys.

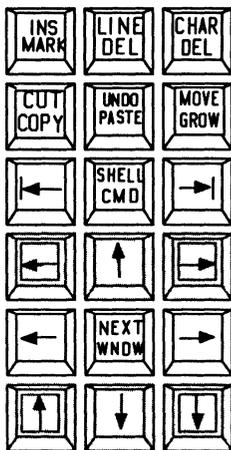


Figure 4-7. Location of Pad Scroll Keys

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 to scroll. A positive n scrolls the pad up n lines; a negative n scrolls the pad down n lines.

You can also use the two predefined function keys shown in Figure 4-7 to scroll a pad either up or down one line at a time. To scroll one line at a time, press `<SHIFT>` and the up-arrow or down-arrow 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 to scroll. A positive *n* scrolls the pad to the left *n* characters; a negative *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 4-7 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 *pathname*

The *pathname* argument specifies the pathname of the file where the DM saves the contents of the pad. You must specify a pathname cataloged on your node; you can not use a pathname 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 4-9 summarizes the commands used to control window groups and icons.

Table 4-9. *Commands for Controlling Window Groups and Icons*

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

mand. If you omit the *entry_name* argument, `wgrr` uses the name 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 name 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 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 in this chapter), 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 4–8 shows the default icon for shell process windows.

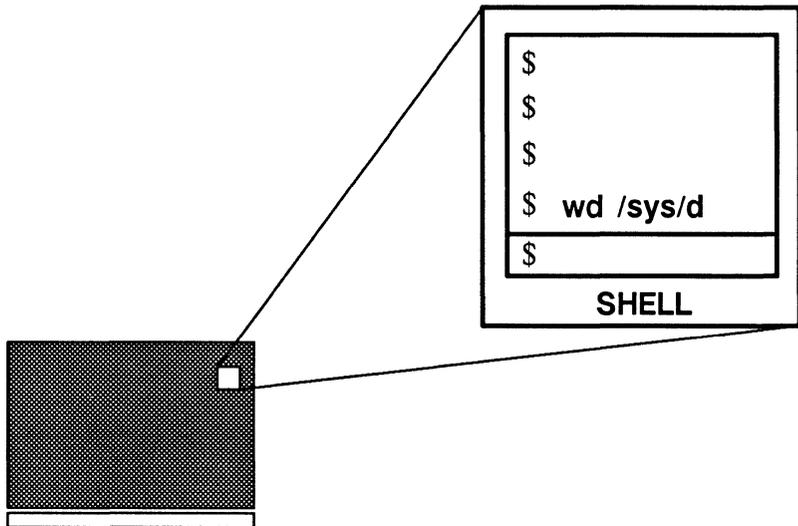


Figure 4–8. Default Icon for Shell Process Windows

To either change a window or window group into an icon, or to change an icon into the window or group it represents, you can use the predefined keys `SHIFT/⟨POP⟩` or specify the `icon` command in the following format:

```
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 for specifying a particular icon. 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 4-8. 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*. (Note that 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*. You can also 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. The `-c` option directs the DM to use the icon 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 horizontal line across the top of portrait displays, and in a vertical line along the right side of landscape displays. The default offset (horizontally for portrait displays, vertically for landscape displays) is the width of one icon (60 pixels).

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 the “Defining Points and Regions” section in Chapter 3 for more 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 group1

This command displays the names of all the windows in the window group **group1**. A paste buffer named **group1** contains these names.

The DM automatically creates three special paste buffers to help you manage your windows and icons. Table 4-10 describes these paste buffers. To list the contents of one of these special paste buffers, enter 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 4-10. 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 5

Editing a Pad

Chapter 4 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 3 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 the current DM mode, which determines whether you can make changes to the material in the pad, and whether the DM either inserts or overstrikes characters that you type. Table 5-1 summarizes the DM commands used to change edit pad modes.

Table 5-1. Commands for Setting Edit Modes

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

Figure 5-1 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**, 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.

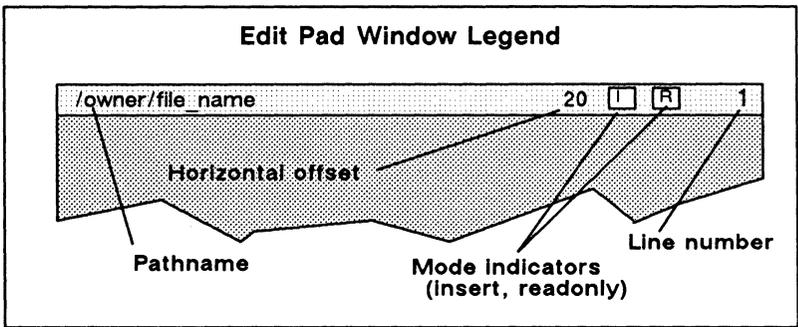


Figure 5-1. The Edit Pad Window Legend

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 (see Figure 5-1). The “R” disappears in write mode. To turn read-only mode either on or off, specify the **ro** 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 (that is, it turns read-only mode off). If you do not specify an option, the **ro** command toggles the current mode setting.

You can also toggle the current setting by typing CTRL/M. The CTRL/M sequence invokes the **ro** command without options.

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). Use the **pw** command, described in the “Updating an Edit File” section, 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 for entering 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 5-1. 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** 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 (that is, 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 5-2 perform special insertion functions.

Table 5-2. Commands for Inserting Characters

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

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** 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 3 describes how to define keys to perform DM functions.

Inserting a Newline Character

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

By default, pressing <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/Z or specify the `eef` 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 CTRL/N. The “Closing Pads and Windows” section in Chapter 4 describes the `wc -q` command line and CTRL/N. See the `wc` command description in the *Domain Display Manager Command Reference* for more information about auto-close mode.

Deleting Text

The commands listed in Table 5-3 delete characters, words, or lines of text. To delete a larger block of text, refer to the “Cutting Text” section.

Table 5-3. Commands for Deleting Text

Task	DM Command	Predefined Key
Delete character at cursor	<code>ed</code>	<CHAR DEL> (L3)
Delete character before cursor	<code>ee</code>	<BACK SPACE> (BS)
Delete "word" of text	<code>dr;/[~a-z0-9\$_]/xd</code>	<F6>
Delete from cursor to end of line	<code>es ";ee;dr;tr;xd;tl;tr</code>	<F7>
Delete entire line	<code>cms;tl;xd</code>	<LINE DEL> (L2)

Deleting Characters

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

To delete the character to the left of the cursor, either press <BACK SPACE> or specify the `ee` command. If the pad is in overstrike mode, the `ee` command replaces the character with a blank.

Both <CHAR DEL> and <BACK SPACE> 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 pressing <PASTE> or specifying the **xp** command. For more about **paste buffers** and the **xp** command, see the “Copying, Cutting, and Pasting Text” section.

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 the “Copying, Cutting, and Pasting Text” section.

Defining a Range of Text

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

To use the **dr** command 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 point with the cursor and use **<MARK>**, which invokes the **dr** and **echo** commands that in turn mark the first point and begin highlighting the text. Figure 5-2 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>**, do the following:

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 (the 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.

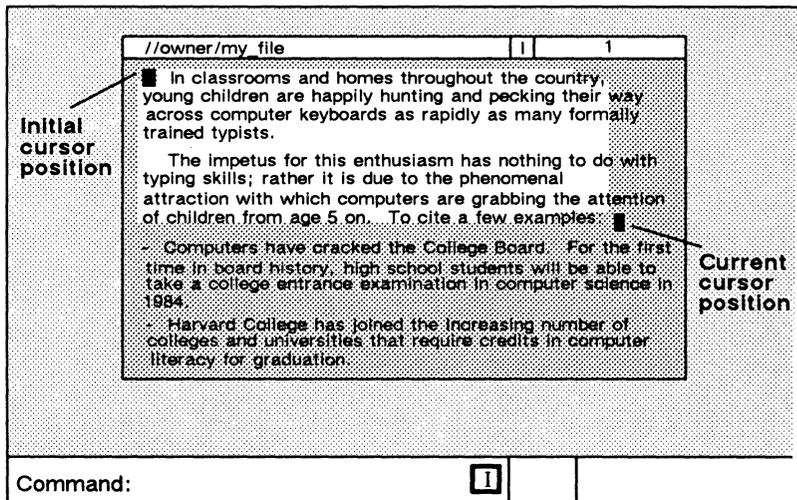


Figure 5-2. Defining a Range of Text with **<MARK>**

Copying, Cutting, and Pasting Text

The commands listed in Table 5-4 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` 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 5-4. Commands for Copying, Cutting, and Pasting Text

Task	DM Command	Predefined Key
Copy text to a paste buffer or file	<code>xc [name -f file] [-r]</code>	<code><COPY></code> (L1A)
Cut (delete) text and write it to a paste buffer or file	<code>xd [name -f file] [-r]</code>	<code><CUT></code> (L1AS)
Paste (write) text from a paste buffer or file into a pad	<code>xp [name -f file] [-r]</code>	<code><PASTE></code> (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 that you have put in the buffer.

Copying Text

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

```
xc [name | -f pathname] [-r]
```

The *name* argument specifies the name of a paste buffer that the DM creates to hold the copied text. The **-f** *pathname* option specifies the name of a permanent file for the text. For example,

```
xc copy_text
```

copies a defined range of text into a paste buffer named **copy_text**.

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 to the paste buffer all characters displayed to the right of the column.

Figure 5-3 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.)

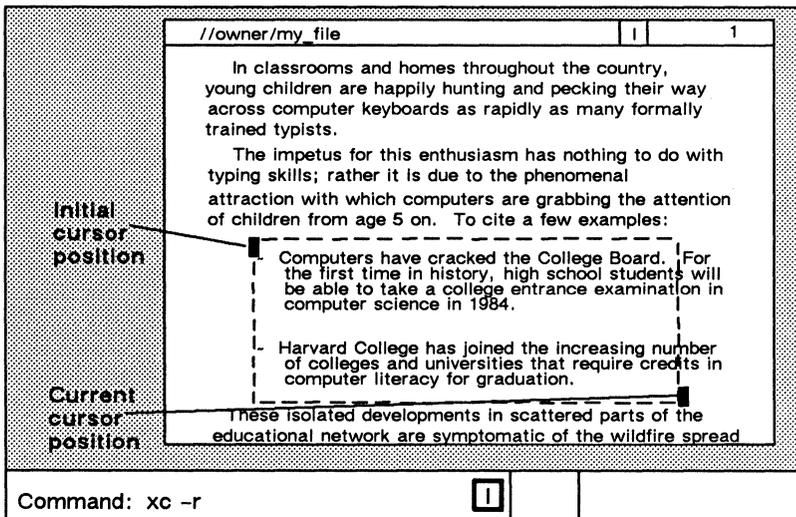


Figure 5-3. Copying Text with the `xc -r` Command

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 pathname` 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 elsewhere in the same pad, or in another pad (see the “Pasting Text” section).

Copying a Display Image

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

```
xi [-f pathname]
```

The `-f pathname` option specifies the name of the file where you want to store the display image. If you omit the `-f` option, the image is written 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. (Note that, on a color node, the `xi` command only copies the text plane, not the full color image.) If you want to copy the whole screen, use the shell command `cpscr` (copy screen). Chapter 7 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` command in the following format:

```
xd [name | -f pathname] [-r]
```

The *name* argument specifies the name of a paste buffer that the DM creates to hold the deleted text. The *-f pathname* 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* 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 pathname* 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* command in the following format:

```
xp [name | -f pathname] [-r]
```

The *name* argument specifies the name of an existing paste buffer that contains the text you want to insert. The *-f pathname* 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` 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, pressing <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 pathname` 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 can also use regular expressions with the shell commands `ed` (edit), `edstr` (edit stream), `fpat` (find pattern), `fpatb` (find pattern block), and `chpat` (change pattern). See the *Aegis 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 list of the special characters used to construct regular expressions, and a brief description of their functions. Although the discussion below only applies to characters used in 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. 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, e.g.,

`%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, e.g.,

The expression `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 ? appears inside a character class (see the “Strings and Character Classes” section); then, 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 “Strings and Character Classes” below), 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**.

`[A-Z][A-Z][A-Z]*` matches any string containing at least two (and possibly more) uppercase characters (see the “Strings and Character Classes” section). Strings like **Mary** would not match, since **Mary** does not begin with two uppercase characters.

Strings and Character Classes

A string of characters enclosed in square brackets [*string*] is called 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 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. Note, however, that the TAB key does not insert a tab character. It simply moves the cursor to the display’s next tab stop. In a regular expression, **@t** matches only tab characters 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, for example,

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

The `s` command is the DM command for substituting strings of text (see the “Substituting All Occurrences of a String” section). This example of the `s` command moves a 3-character sequence from the beginning of a line to the end of the line. The characters `???` match 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 command for substituting strings of text, but it only substitutes the first occurrence of the first pattern on a line (see the “Substituting the First Occurrence of a String” section). 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 5-5 locate strings of characters in a pad. You describe the string pattern using regular expressions (see the previous section).

Table 5-5. Commands for Searching for Text

Task	DM Command	Predefined Key
Search forward for string	<i>/string/</i>	None
Search backward for string	<i>\string\</i>	None
Repeat last forward search	//	CTRL/R
Repeat last backward search	\\	CTRL/U
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 (see the *Aegis Command Reference*).

Actually, a search is not syntactically a command; it is 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` or `sq` command (see the “Cancelling a Search Operation” section).
- 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/R` sequence.

To repeat the last search backward, specify the `\\` command or type the `CTRL/U` sequence.

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

Canceling a Search Operation

To cancel the current search operation, type CTRL/X. The CTRL/X sequence invokes the **abrt** 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 the “Defining Keys” section in Chapter 5).

The DM command **sq** 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, uppercase 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 5-6 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** 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 substitu-

tion occurs from the current cursor position to the end of the line. All substitutions are case-sensitive. You cannot make a substitution case-insensitive.

Table 5-6. Commands for Substituting Text

Task	DM Command	Predefined Key
Substitute <i>string2</i> for all occurrences of <i>string1</i> in a defined range	<i>s/string1/string2</i>	None
Substitute <i>string2</i> for the first occurrence of <i>string1</i> in each line of a defined range	<i>so/string1/string2</i>	None
Change case of each letter in a defined range	<i>case [-s] [-u] [-l]</i>	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.
- You can 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, specify the **undo** command. You can also undo the previous command by pressing <UNDO>.

NOTE: The **undo** command only applies to DM operations. You cannot undo 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 information about the **wc** command, refer to the “Closing Pads and Windows” section in Chapter 4.)

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

1. The **pw** command leaves the edit pad open so that you can continue editing the file.
2. 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>` or the `CTRL/Y` sequence (see the “Closing Pads and Windows” section in Chapter 4).



Chapter 6

Using the Aegis Shell

Chapter 3 describes the Display Manager (DM), the operating system program that you use to create processes and control your node's display. We supply another operating system program, called the **command shell**, that lets you perform more traditional computing operations. The shell lets you enter commands to perform such operations as copying files and directories, compiling and running programs, and monitoring system activity.

This chapter describes the command shell environment that processes shell commands. The chapter includes information on shell commands, controlling command input and output, the command line parser, and using pathname wildcards.

Shell Commands

The command shell runs in a process called a **shell process**. As shown in Figure 6-1, you enter shell commands in the shell's process input pad, referred to as the **shell input pad**. To specify a shell command, type the name of the command next to the dollar sign (\$) prompt and press <RETURN>.

Most shell commands that you specify in the shell input pad are actually the names of command files that the shell looks for and executes. For example, when you specify the command **date**, the shell looks for a command file named **date** and executes it. The shell looks for command files according to a set of command search rules that indicate which directories the shell should search. You'll learn more about command search rules later in this chapter.

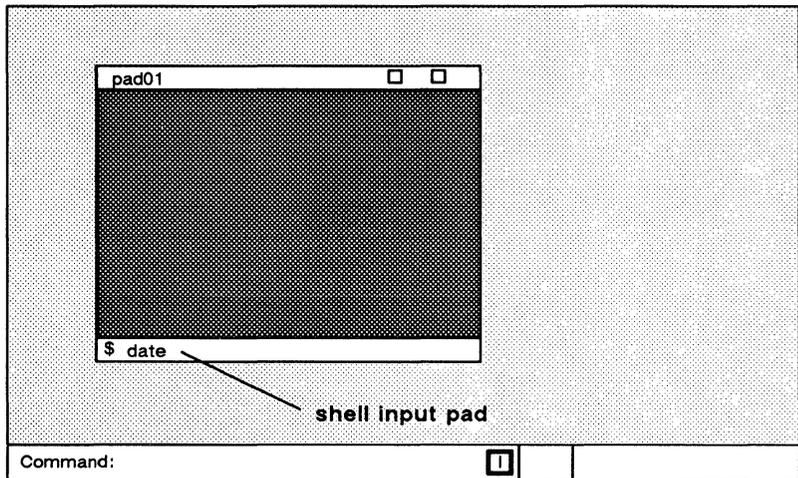


Figure 6-1. The Aegis Shell Process

As part of the Aegis environment, we supply a set of shell commands for your use. The *Aegis Command Reference* provides detailed descriptions of all the commands that we supply. You can also create your own shell command programs, called **shell scripts**, and execute them in the same way you execute the shell commands that we supply. Chapter 11 describes how to write shell scripts.

Command Line Format

Aegis shell command lines have the following format:

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

The value of *command* is the name of either a shell command or shell script.

The *arguments* indicate which objects you want the command to operate on. An argument is either the pathname of an object in the naming tree, or a literal string that you want the command to manipulate. Always separate an argument from a command and any additional arguments or options with at least one blank space.

The *options* that you specify direct the command to perform a special action. As shown in the command line format, you can specify options either before or after arguments on the command line. Many options require secondary arguments of their own. Thus, to properly delimit options, always precede each option with a hyphen (-).

Figure 6-2 shows a sample command line and its components.

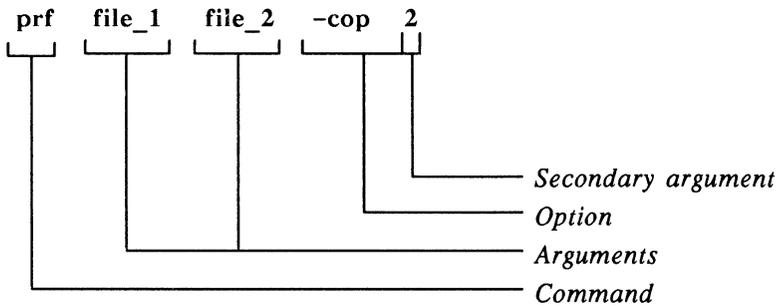


Figure 6-2. Shell Command Line Components

The command line in Figure 6-2 prints two files: `file_1` and `file_2`. The `-cop 2` option directs `prf` to print two copies of each file.

Normally, you specify each command as a separate line. You can also place multiple commands together on the same line by separating them with semicolons (;). For example,

```
$ wd //my_dir/sub_1; ld
```

executes two commands: `wd` sets the working directory to `//my_dir/sub`, and `ld` lists the contents of that directory.

You may also specify multiple commands on a single line when you use pipes and filters. The “Redirecting Output to Other Commands” section describes how to use pipes and filters.

The DM limits each shell command line to 1024 characters. You can, however, continue a command over several lines by typing an at sign (@) character and then pressing <RETURN> at the end of each line you want to continue, as follows:

```
$ wd //node@
$_ /directory
```

The @ character is a special character (see the “Special Characters” section) called an **escape character**. As shown in the example above, the @ character “escapes” the normal execution of <RETURN>, letting you continue the command line at the continuation prompt (\$_).

Standard Command Options

All Aegis shell commands that we supply allow you to specify the standard command options listed in Table 6-1. These options allow you to display useful information about a command.

Table 6-1. Standard Aegis Shell Command Options

Option	Description
-help	Displays detailed information on how to use the command
-usage	Displays a brief summary on how to use the command
-version	Displays the command’s software version number

Command Search Rules

As mentioned earlier, most commands are the names of files. Since you usually specify command names, rather than the full pathnames of the files they represent, the shell searches different locations in the system naming tree for the file that matches the command name you specify. When you specify a command, the shell determines which directories to search according to a set of command search rules.

Some commands, such as **inlib** (used to initialize a library) are not files. They invoke internal shell functions and do not follow command search rules. The *Aegis Command Reference* identifies which commands are internal shell commands.

The default command search rules direct the shell to search the following directories in the order shown:

1. Your working directory (.)
2. Your personal command directory (**~/com**)
3. The system command directory (**/com**)
4. The system command extensions directory (**/usr/apollo/bin**).

When you specify a command, the shell searches the directories in the order specified by the command search rules. As soon as the shell finds a file with the name that matches the command you specified, it attempts to execute the file. For example, when you specify the shell command

```
$ ld
```

the shell first looks for a file named **ld** in your current working directory. If the shell does not find the file in the current working directory, it checks the directory **~/com**, which is a subdirectory of your naming directory. The **~/com** directory is your personal command directory where you store your own frequently used shell scripts. (You don't have to create a personal command directory; if the shell doesn't find one, it continues its search and no error occurs.)

The shell then searches the node's main command directory `/com`, which contains all of the standard commands that we supply. Since `ld` is a system command, the shell finds it in `/com` and executes it, displaying the contents of the working directory. (The final directory that would have been searched, had the command not been found in `/com`, is the `/usr/apollo/bin` directory, which contains all of the command extensions that we provide.)

This example assumes that you have not created an executable file or shell script named `ld` in your working directory or personal command directory. Had you done so, the shell would have executed your version of `ld` before the system's version.

You can set or show a shell's command search rules using the `csr` (command search rules) command. For example, the `csr` command in this example displays the shell's command search rules:

```
$ csr
. ~/com /com /usr/apollo/bin
```

You can use the `-a` option with the `csr` command to append additional directories to the current list. For example, to add two additional directories (`~/prog` and `/prog`) to the current set, specify:

```
$ csr -a ~/prog /prog
$ csr
~/com /com /usr/apollo/bin ~/prog /prog
```

To completely change a set of command search rules, specify the `csr` command along with a new set of rules. For example:

```
$ csr /com /usr/apollo/bin ~/com
$ csr
/com /usr/apollo/bin ~/com
```

If you change a shell's command search rules, any shell script that you invoke inherits the new command search rules. If you create a new process running the shell, the new shell uses the original default command search rules, not the new rules. Also changing the search rules in a shell script will usually change them in the parent shell.

Special Characters

The shell recognizes a variety of special characters that allow you to direct the action of commands. These characters are divided into three basic categories:

- Input and output (I/O) control characters
- Pathname wildcards
- Parsing operators

The following sections explain how to use I/O control characters and pathname wildcards. Since you will use shell command parsing operators most frequently in shell scripts, we describe parsing operators in detail in Chapter 11.

Creating and Invoking Shells

The shell is a command line interpreter that reads command lines that you type and interprets them as requests to execute other programs. When you press <SHELL>, you create a process running the shell program. Each new shell process that you create provides a separate environment in which the shell runs.

You can invoke additional Aegis shells from within a shell process using the shell command **sh**. When you specify **sh**, you generate a separate subordinate shell, in which you can carry on separate operations and execute programs and scripts. Each subordinate shell may inherit environment variables from its parent shell.

Setting Up the Initial Shell Environment

When you log in using the DM, the `/sys/dm/login_sh` command executes a log-in shell. This shell looks in the directory `~/user_data/sh` for a file named **startup** and a file named **login**. If these files exist, the shell executes them to set up the initial environment for the shell (it executes **startup** first, then **login**). Since no default **startup** or **login** files exist, you must create them if you want to use them.

The **login** file is only executed when you start a shell with the **login_sh** command. You can use this file for commands that you wish to execute once, when you log in. For example, your **login** file may contain commands to start a clock program or a mail program.

The **startup** file is executed whenever you create a new process to run the shell. The **startup** script helps when you want to define a standard set of variables for each shell process, or set up certain shell characteristics, such as variable evaluation on (EON), or new command search rules. Figure 6-3 shows a sample shell start-up script.

NOTE: The **siologin** command (which invokes a log-in sequence on an sio line) executes its own shell start-up file. For more information about the **siologin** command, see the *Aegis Command Reference*.

```
# Sample shell start-up script ~/user_data/sh/startup
#
# Set up standard variables
#
A := 3
B := 4
# Turn on variable evaluation
eon
# Add additional directory to command search rules
csr -a ~/progs
```

Figure 6-3. Sample Shell Start-Up Script

Controlling Input and Output

Processes pass data to and from programs, such as the shell, through open system channels called **streams**. Every process that you create has the following streams open for program input and output:

- Standard input
- Standard output
- Error input
- Error output

Standard input and **standard output** are the streams that channel normal input and output between a program and a process. By default, standard input passes program input that you type in the process input pad; standard output passes program output to the process transcript pad.

Error input and **Error output** are two streams used for additional program input and output. Like the standard streams, they use the process input pad and process transcript pad by default.

The error input stream has nothing to do with errors; it is simply another input stream for passing data to a program. For example, when a command queries you to verify wildcard names, error input passes your response to the command. (The “Using Query Options” section describes how commands query you to verify wildcard names.) Error output is the stream that passes program error messages to the process transcript pad. (However, if the object type is **coff**, you can’t redirect error input.)

Shell commands use input and output streams when processing command line data. When you specify a command in the shell input pad, standard input passes data from the command line to the command program. Standard output passes data from the program to the transcript pad.

In certain instances, you may want the shell to read input from and write output to locations other than the input and transcript pads. For example, you may want to save the output from a command in a file. Using **I/O control characters**, you can redirect input and output streams to pass data to and from other locations, usually files. Table 6–2 lists the I/O control characters and a brief summary of their functions.

Table 6-2. I/O Control Characters

Character	Function
<	Redirect standard input
< ?	Redirect error input
>	Redirect standard output
> ?	Redirect error output
>>	Append standard output
>>?	Append error output
	Pipe standard output
()	Group commands for I/O redirection
<<	Redirect standard input to read data from scripts.
<< ?	Redirect error input to read data from scripts.

The following sections show how to use I/O control characters. Chapter 11 describes the characters used to redirect standard input to read data from scripts.

Reading Input from a File

To redirect standard input to read data from a file rather than the input pad, use a left angle bracket (<). For example, the following command reads data from a file named `file_1`:

```
$ tlc a-z A-z < file_1
```

The `tlc` command, used to transliterate characters, normally substitutes or deletes characters from text that you type in the shell input pad. The `tlc` command in this example redirects standard input to read data from a file (`file_1`) instead of the input pad. The command changes all lowercase characters in `file_1` to uppercase characters, and writes the converted text to the transcript pad.

Writing Output to a File

To redirect standard output to write output to a file rather than to the transcript pad, use a right angle bracket (>). For example, the following command writes output to a file named **file_1.fmt**:

```
$ fmt file_1 > file_1.fmt
```

The **fmt** command formats **file_1** and writes the output (the formatted file) to **file_1.fmt** instead of to the transcript pad.

Shell commands use the error output stream to report any errors found in the input file. By default, error output writes output to the transcript pad. To redirect error output to write output to a file instead of the transcript pad, use a right angle bracket/question mark combination (>?). For example, the following command redirects both standard output and error output:

```
$ fmt file_1 > file_1.fmt >? file_1.err
```

The **fmt** command writes the formatted file to **file_1.fmt**, and if it discovers any errors, writes error messages to the file **file_1.err**.

Appending Output to a File

To redirect standard output to append output to the end of a file, use double right angle brackets (>>). For example, the following command appends output to a file named **book**:

```
$ catf ch4 ch5 ch6 >> book
```

The **catf** command, used to catenate files, normally reads input files in order and writes them to the transcript pad. The **catf** command in this example reads the files **ch4**, **ch5**, and **ch6** in that order and appends them to the existing file **book**. If the specified output file didn't exist, **catf** would create a new file named **book** and write output to it.

To redirect error output to append error output to the end of a file, use a double right angle bracket/question mark (>>?) combination. For example, suppose you wanted to keep a record of all **fmt** er-

rors. Each time you used the **fmt** command to format a file, you could direct the command to append error output to a file as follows:

```
$ fmt file_1 >>? error_log
```

The command in this example formats **file_1**. If it encounters any errors, it appends any error output to the file **error_log**.

Redirecting Output to Other Commands

If you place two commands on one line, and separate them with a vertical bar (**|**), the shell connects the standard output stream of the first command to the standard input stream of the next command. For example, the **srf** command here sorts the contents of **file_1** and passes the output to the shell command **dldupl**, which deletes duplicate lines:

```
$ srf file_1 | dldupl
```

The vertical bar between the commands is called a **pipe**. Commands such as **srf** and **dldupl** that copy standard input to standard output (making some changes along the way) are called **filters**. A command line that uses pipes and filters is called a **pipeline**. You can use either shell commands or scripts as filters in pipelines.

To use a group of commands as a filter, enclose them in parentheses using the following format:

```
( command_1; command_2 ) | command_3
```

The shell passes output from the commands enclosed in parentheses (*command_1* and *command_2*) to the command to the right of the vertical bar *command_3*. For example:

```
$ (ld my_dir1 -c -nhd; ld my_dir2 -c -nhd) | srf > list
```

This example concatenates the output of the two **ld** commands and then sorts the reported file names, placing the output in the file called **list**.

The Command Line Parser

Many of the shell commands that we supply share a standard command line parsing procedure. This procedure, called the **command line parser**, determines how each command processes command line information. The shell command descriptions in the *Aegis Command Reference* and the online help files identify which commands use the command line parser.

Commands that use the command line parser allow you to:

- Specify multiple pathnames as pathname arguments.
- Use pathname wildcards to specify existing pathnames and to derive pathnames from other pathnames on the command line.

Commands that use the command line parser also accept the command parser options listed in Table 6-3. These options let you:

- Control how a command queries you to verify wildcard matches.
- Direct a command to use standard input to read command line input.

Table 6-3. Command Line Parser Options

Option	Description
-ae	Causes the command to abort if it cannot find a name in a pathname. By default, processing continues to the next name.
-nq	Do not issue a query to verify wildcard names. This is the default for commands that don't delete or modify objects.
-qw	Issue a query to verify wildcard names.
-qa	Issue a query to verify all names.
-	Read additional data from standard input.
* [pathname]	Read the specified file for additional pathname argument. If a pathname is omitted, read the additional pathname argument from standard input.

Using Query Options

Commands that delete or modify objects query you to verify names that you specify using wildcards. You can control how a command queries by using any of the query options listed in Table 6-4.

By default, commands use error output to query you by writing selected names to the transcript pad with a question mark (?), prompting you for a response. The command uses the error input stream to read your response from the shell input pad.

To respond, type one of the responses listed in Table 6-4 and press <RETURN>.

Table 6-4. Command Query Responses

Response	Action
h[elp]	Displays help information.
y[es]	Operates on the file with that name.
n[o]	Ignores the file with that name.
q[uit]	Quits immediately.
g[o]	Operates on the file with that name and suppresses further name queries.
d[efault]	Resets the default response.

By default, queries require a response; if you simply type <RETURN> without a response, the command queries you again. To change the default, use the **d** response, followed by either **yes**, **no**, or **none**. For example,

```
? d yes
```

sets the default to **yes**. If you respond to subsequent queries by typing <RETURN>, the command uses the new default and operates on the file with that name. A value of **none** indicates that you must specify a response.

Reading Data from Standard Input

When you enter a shell command, the command normally reads input data from arguments that you give in the command line. For example, the **prf** command reads data from the specified files and prints it. To direct the command to read data from standard input instead of an input file, use a hyphen character (-) as shown here:

```
$ prf -  
Print this line on the line printer  
And this one too  
CTRL/Z
```

Standard input uses the shell input pad by default, so **prf** reads data from the input pad. To input data to the **prf** command, type in data as shown in the example. The CTRL/Z control key inserts an end-of-file (EOF) that signals the end of input and causes **prf** to print any data that you typed in the input pad.

To access a file whose name begins with a hyphen (-), use an escape character (@) on the command line as follows:

```
$ command @-file
```

This eliminates the shell's interpretation of *-file* as an option.

Some commands and scripts receive data from both a list of files and standard input. For these commands, specify the hyphen character (-) as a pathname argument. For example:

```
$ fmt file_1 - file_2
```

The **fmt** command formats **file_1**, formats data typed in standard input, and finally formats **file_2**.

Reading Pathnames from Standard Input

To direct a command to read pathnames rather than data from standard input, use the asterisk symbol (*). The **prf** command in the following example reads pathname arguments that you type in the shell input pad:

```
$ prf *  
file_1  
file_2  
file_3  
CTRL/Z
```

Pressing CTRL/Z inserts an end-of-file (EOF) that signals the end of input, which causes **prf** to print the contents of each file.

You can also use the asterisk symbol and redirect standard input to read pathnames from a **names file**, a file that contains the pathnames of other files, as follows:

```
$ prf *jobs
```

Here, **prf** prints the contents of each file in the names file **jobs**.

Using Pathname Wildcards

Most shell commands accept pathnames as arguments. The commands that use the command line parser also accept **wildcards** as part of their pathname arguments. Wildcards are characters or text strings that you can use in pathnames to represent one or more text strings in a pathname. For example, the wildcards here match every file that ends with **.ftn** in the current working directory:

```
$ ls ?*.ftn
  └─
wildcards
```

The question mark wildcard (?) matches any single character except <RETURN>. The asterisk wildcard (*) matches zero or more occurrences of the character preceding it. Thus, this wildcard combination matches zero or more occurrences of any character. Table 6-5 lists pathname wildcards and briefly describes how they work.

Table 6-5. Summary of Pathname Wildcards

Character	Description
?	Matches any single character but <RETURN>, e.g., z? matches any two-character name beginning with the letter z (za and z1)
%	Matches zero or more characters up to, but not including, a period. For example, %.bak matches any name ending in .bak (sales.bak but not sales.bak.bak) demo.% matches any name beginning with demo , up to and including the period (demo.bak and demo.pas) demo.%.% matches demo.pas.bak

(Continued)

Table 6-5. Summary of Pathname Wildcards (Cont.)

Character	Description
*	<p>Matches zero or more occurrences of the character that precedes it. After the ? character, it matches zero or more occurrences of any character except <RETURN>, e.g.,</p> <p>file9* matches file, file9, and file999.</p> <p>de?* matches demo, desk, and department</p> <p>demo.*% matches any name that demo.% matches (see the % example); also matches the period (demo.fmt and demo)</p>
[<i>string</i>]	<p>Matches any single character listed in <i>string</i>, e.g.,</p> <p>file [0-9] matches any five-character name that begins with file and ends in a single digit (file4 and file8)</p> <p>file[a-d] matches filea but not filem</p> <p>file[axy] matches filea, filex, and filey</p>
[~ <i>string</i>]	<p>Matches any single character not appearing in <i>string</i>, e.g.,</p> <p>file.[~a-z] matches file and file.9 but not filea or file.p</p>
...	<p>Matches zero or more directories subordinate to the starting point, e.g.,</p> <p>//my_node/... matches all directories in my_node</p> <p>/owner/.../demo matches any object named demo in a subdirectory of owner</p> <p>.../jan?* matches any object starting with jan in subdirectories of the working directory</p>

(Continued)

Running Programs in a Background Process

The command shell has another set of special characters, called **parsing operators**, that control how a command parses (interprets and categorizes) the individual components on a command line. We've already seen how to use some of these parsing operators: the semicolon (;) to separate multiple commands on a command line, the escape character (@) to continue commands on more than one line, and blank spaces to separate command arguments and options.

Another parsing operator is the **ampersand character (&)**. It instructs the shell to run a program in a **background process** (a process that runs without pads and windows). This use of & is unrelated to its use as an input request character in DM scripts.

To run a command or program in a background process, enter the command line or program name in the shell input pad, followed by the & character. For example:

```
$ bind file_1.bin -map > prog.map &
```

This command line runs the binder as a background process to bind the file **file_1.bin** and writes a complete map to the file **prog.map**.

By default, the shell directs output to the device **/dev/null** (which discards the output). You can display output from the background process by specifying the shell command **bon**. The **bon** command directs the shell to where the background process was invoked to display output in its transcript pad. To turn output off (direct output to **/dev/null**), specify the **boff** command.

The remainder of the shell parsing operators are used most frequently in scripts. Chapter 11 contains a complete list of parsing operators and describes how to use them in writing shell scripts.



Chapter 7

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 let you 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, see Chapter 6.

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 *Aegis 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 or naming directory. To move around the naming tree, you need to know how to set your working directory and naming directory. Table 7-1 summarizes the commands used to move around the naming tree.

Table 7-1. Commands for Setting the Working and Naming Directory

Task	Shell Command
Set or display working directory	<code>wd [pathname]</code>
Set or display naming directory	<code>nd [pathname]</code>

Setting the Working Directory

The working directory is where the system begins its search for objects when you omit the object's full pathname. At login, the system sets your initial working directory to the home directory designated in your user account (see Chapter 2). Each subsequent process that you create uses the working directory of the previous process as its working directory.

To display the name of a process's current working directory, type the `wd` (working directory) command without any arguments or options as follows:

```
$ wd
```

To change a process's working directory to another directory, specify the **wd** command in the following format:

```
wd [pathname]
```

The *pathname* argument specifies the pathname of the directory you want to use as the working directory. For example:

```
$ wd //my_node/owner/forms
```

sets the working directory for the current process to **forms**. Once set, anytime you omit the full pathname of an object, the system starts its search at the directory **forms** by default.

Setting the Naming Directory

The system searches the naming directory's **com** directory (**~/com**) as part of the default command search rules (see Chapter 6). As described in Chapter 1, the naming directory is also where the system begins its search for an object when you precede an object's pathname with the tilde and slash symbols combined (**~/**).

At login, the system sets the naming directory to the home directory designated in your user account (see Chapter 2). Each subsequent process that you create uses the naming directory of the previous process as its naming directory.

To display the name of a process's current naming directory, specify the **nd** (naming directory) command without any arguments or options as follows:

```
$ nd
```

To change a process's naming directory to another directory, specify the **nd** command in the following format:

```
nd [pathname]
```

The *pathname* argument specifies the pathname of the directory you want to use as the naming directory. For example:

```
$ nd /user_1/reports
```

sets the naming directory to the directory **reports**. Once set, you can use a tilde and slash (~/) in place of **/user_1/reports** at the beginning of any pathname. Thus, **~/cal_85** would be identical to **/user_1/reports/cal_85**.

Table 7-2 summarizes the commands for managing files.

Table 7-2. Commands for Managing Files

Task	Shell Command
Create a file	ce file (DM command), or catf > target
Rename a file	chn old_name [new_name]
Copy a file	cpf source [target]
Move a file	mvf source [target]
Append file to another file	catf source >> target
Print a file	prf [pathname]
Display file attributes	ld [pathname]
Delete a file	dlf [pathname]
Copy display image to file	cpscr [pathname]
Compare ASCII files	cmf source [target]
Compare sorted files	cmsrf [option] source [target]

Creating Files

To create normal text files, specify the DM command **ce** (create edit pad) along with the pathname of the file you want to create. 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 typing CTRL/Y. 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 or naming directory, the system uses the working or naming 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 working directory of the current process 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.

Whenever you create a file, the system assigns the file a set of initial ACLs from the file's parent directory. After you create a file you can change its ACLs with the **edacl** (edit ACL) command. Chapter 10 explains ACLs and describes how to use this command.

Renaming Files

To change the name of a file, use the **chn** (change name) command in the following format:

```
chn old_name [new_name] [options]
```

The *old_name* argument specifies the current pathname of the file you want to rename, and *new_name* specifies the new name of the file. For example:

```
$ chn /owner/john paul
```

changes the name of the file **john** to **paul**. Notice that the *new_name* argument affects only the rightmost component (**john**) of the *old_name* argument.

To append a naming suffix to the new filename, specify any of the following naming options:

Option	Description
-d	Appends the current month and day to the new name (<i>new_name.mm.dd</i>).
-y	Appends the current year, month, and date to the new name (<i>new_name.yy.mm.dd</i>).
-u	Forces the system to create a unique new name by appending a sequence of number(s) to the end of the name.

If you omit the *new_name* argument, you must specify one of the options in the previous list; the system creates a new name by copying the *old_name* and appending the proper suffix as shown below.

```
$ chn /owner/john -d
```

When specified on June 16, this command changes the name of the file **john** to **john.06.16**.

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 **cpf** (copy file) command in the following format:

```
cpf source [target] [options]
```

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 6 apply to both command arguments.

The **cpf** command always creates a copy of the source file at the location specified by the target. For example:

```
$ cpf 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, **cpf** assigns the copy the name specified by the target, which in this example is **new_memo**.

If the target specifies the pathname of a directory, **cpf** 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:

```
$ cpf memo /user_1
```

copies the file **memo** from the current working directory to the target directory **user_1**. Because **cpf** assigns the copy the name of the source file, the new file has the pathname **/user_1/memo**.

If you omit the target pathname entirely, **cpf** creates a copy of the source in the current working directory, unless the source file also resides there. In the previous example, since the source file **memo** is in the current working directory, **cpf** can't create another file named **memo** in the same directory. In this case, **cpf** displays the error message, "... can't copy a file or tree to itself" and does not make a copy.

By default, the system assigns the target file the default file ACLs of its parent directory (see Chapter 10). So, in the previous example, the system assigns the target file the default file ACLs of the directory **user_1**.

To replace an existing file with a copy of another file, use the **-r** option as follows:

```
$ cpf /owner/june_report latest_report -r
```

This command replaces the contents of the file **latest_report** (in the current working directory) with a copy of the contents of **june_report**. As a result, **latest_report** now contains a copy of **june_report**.

You can copy or replace several files using a single **cpf** command by either specifying multiple pairs of source and target pathnames (each pair separated by a space) or by using pathname wildcards. The following command uses pathname wildcards to copy all the files ending with **plan** to the current working directory:

```
$ cpf /owner/?*plan -lf
```

The **-lf** option directs the **cpf** command to list the name of each file it copies. For more information on using pathname wildcards, see Chapter 6.

Moving Files

When you move a file, you literally relocate the file in the naming tree. Use the **mvf** (move file) command the same way you use the **cpf** command described in the previous section. In fact, moving a file has the same effect as copying the file to another location and then deleting the original. Unlike a copy operation, however, when you move a file, it retains its original ACLs.

To move a file or group of files from one location in the naming tree to another, use the **mvf** command in the format:

```
mvf source [target] [options]
```

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**:

```
$ mvf /designer/floorplan /builder/plans/cape
```

In this example, the target specifies the pathname for a nonexistent file named **cape**. The **mvf** 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, **mvf** moves the source file into the target directory (or current working directory if you omit the target pathname). For example, the following command moves the file **floorplan** into the current working directory:

```
$ mvf /designer/floorplan
```

In this example, since no target filename was specified, the file retains the name of the source file (**floorplan**).

You can move a file to replace an existing file in another directory by using the `mvf` command with the `-r` option, as follows:

```
$ mvf /owner/report latest_report -r
```

This command replaces the contents of the file `latest_report` (in the current working directory) with the contents of the file `report` (in the directory `/owner`).

To move several files in one operation, you can specify multiple pairs of source and target pathnames or use pathname wildcards. Chapter 6 describes how to use pathname wildcards.

Appending Files

To append the contents of one or more files to the end of another file, use the `catf` (catenate file) command in the following format:

```
catf [source ...] >> target
```

The *source* argument specifies the pathname of the file whose contents you want to append, and *target* specifies the pathname of the file to which you want to append. When you specify more than one source file, separate each file with a space. The system concatenates the source files and appends them to the target file.

The `catf` command reads source files in order and normally writes them to standard output, which is, by default, the shell's process transcript pad. The double right-angle brackets (`>>`), however, redirect output from standard output and append the output to the target file. For example:

```
$ catf memo_1 memo_4 >> plan_memos
```

reads the files `memo_1` and `memo_4` in that order, and appends them to the file `plan_memos`.

Chapter 6 provides more information on how to use I/O control characters.

Printing Files

In the Aegis environment, there are essentially two methods used to print files. You may specify a print command, or you may use a special print menu interface to print your files. The following sections describe both of these methods.

Using the `prf` Command

To print one or more files, use the `prf` (print file) command in the following format:

```
prf [pathname ...] [options]
```

The *pathname* argument specifies the name of the file you want to print. You can print multiple files using a single `prf` command by either specifying multiple pathnames (each pathname separated by a space) or by using pathname wildcards. The following command uses pathname 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
```

The `prf` command itself doesn't actually print files; a server process called `prsvr` (print server) does. (The `prsvr` process runs on the node that is physically connected to the printer.) The `prf` command queues a file for printing by copying the file to a directory, where it waits for `prsvr` to get it and print it.

If you normally use a printer connected to your node, the directory `/sys/print/queue` is the name of the queue directory on your node. 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 9 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 *Aegis Command Reference* entry for `prf` and to *Printing in the Aegis Environment*.

Using the Print Menu Interface

In addition to the `prf` command, you can print files using 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, use the `-dia` option:

```
$ prf -dia
```

As shown in Figure 7-1, 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.

The screenshot shows a window titled "Print" with a menu bar containing "Quit", "information", "control", and "shell commands". Below the menu bar, there is a "Print" button, a "File to print:" field with an arrow cursor, and a "Printer:" field with the value "p". The main area is titled "Job Properties" and contains several options:

<input checked="" type="checkbox"/> text	char specs	copies
<input type="checkbox"/> bitmap	columns	for whom
<input type="checkbox"/> other	margins	spool node
<input type="checkbox"/> filters	headers	notify
	carriage	banner
	word wrap	

Figure 7-1. 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>, as shown in Figure 7-2:



Figure 7-2. 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 7-3 describes the submenus for the **shell commands** menu item.

Table 7-3. 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 *Aegis Command Reference*.

Displaying File Attributes

To display a file's attributes, such as its size, creation date, and access rights, use the `ld` (list directory) command as follows:

```
ld pathname [options]
```

The *pathname* argument specifies the pathname of the file, and *options* specifies which attributes you want displayed.

The `ld` command in the following example displays attribute information for the file `memo`, as shown in the following example. The command specifies two attribute options: `-r`, which displays the file's access rights, and `-drc`, which displays the file's creation date and time.

```
$ ld //node/user/memo -r -drc
```

```
rights      date/time      name
          created
pgndcalr 85/01/04.09:16 //node/user/memo
```

```
1 entries, 2 blocks used.
```

To display an entire set of attributes for a file, use the `-a` option as follows:

```
$ ld //node/user/memo -a
```

You can display the attributes of several files by either specifying multiple file pathnames, (each pathname separated by a space) or by using pathname wildcards. The command in the following example uses the question mark (?) and asterisk (*) pathname wildcards to display attribute information for all files in the current working directory that have the suffix, `_plan`:

```
$ ld ?*_plan -r -drc
```

You can also display the attributes of all the files in a particular directory by specifying the name of the directory as the pathname argument. The “Displaying Directory Information” section of Chapter 8 describes how to use `ld` to list the contents of a directory and display attribute information about its contents.

Deleting Files

To delete one or more files, use the `dif` (delete file) command in the following format:

```
dif [pathname ...] [options]
```

The *pathname* 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 `sales_plan` and `report` from the current working directory:

```
$ dlf sales_plan report
```

You can also use pathname wildcards to delete related groups of files. For example:

```
$ dlf *.bak -l
```

The percent sign (%) wildcard character causes `dlf` to delete all of the files in the current working directory that end in `.bak`. The `-l` option lists each file as `dlf` deletes it.

The `dlf` command is an example of one that queries you to verify names that you specify with pathname wildcards. If you enter the previous example, the `dlf` command then asks you to verify the deletion of each file that matches the pathname `*.bak`.

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 pathname [-inv]
```

The *pathname* argument specifies the pathname of the file to which you want to copy the display image. The `-inv` 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 as follows:

```
cpscr pathname -gpr
```

The `-gpr` 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 `cmf` (compare file) command in the following format:

```
cmf source [target ...] [options]
```

The *source* argument specifies the pathname of the file to which `cmf` compares one or more *target* files; `cmf` reports all differences in relation to the source file. If you specify multiple target pathnames, separate them with spaces. If you omit a target pathname, `cmf` compares the source with text from standard input.

The `cmf` command in Figure 7-3 compares the contents of the file `speech` to the contents of `speech.bak`.

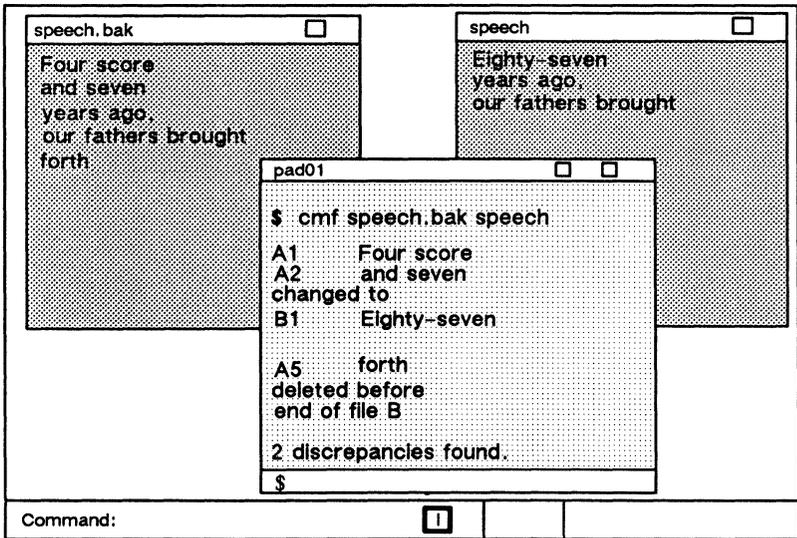


Figure 7-3. Comparing Two ASCII Files



Chapter 8

Managing Directories

Directories are the naming tree components that contain other objects. Table 8-1 summarizes the commands for managing directories.

Table 8-1. Commands for Managing Directories

Task	Shell Command
Create a directory	crd <i>pathname</i>
Rename a directory	chn <i>old_name</i> [<i>new_name</i>]
Copy a directory tree	cpt <i>source</i> [<i>target</i>]
Replace a directory tree	cpt <i>source</i> [<i>target</i>] -r
Merge directory trees	cpt <i>source</i> [<i>target</i>] -ms
Compare directory trees	cmt <i>source target</i>
Display contents of a directory	ld [<i>pathname</i>]
Delete a directory tree	dlt <i>pathname</i>

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 **crd** (create directory) command in the following format:

```
crd pathname ...
```

The *pathname* 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**:

```
$ crd /owner/reports
```

The **crd** command creates the directory **reports** as a subdirectory of the parent directory **/owner**. The new directory, (**reports**) also receives an initial set of ACLs from the initial directory ACLs of the parent directory (**/owner**). You can change the initial ACLs with the **edacl** (edit ACL) command. Chapter 10 explains ACLs and describes how to use this command.

Renaming Directories

To change the name of a directory, use the **chn** (change name) command in the following format:

```
chn old_name [new_name] [options]
```

The *old_name* argument specifies the pathname of the directory you want to rename, and *new_name* specifies the new name of the directory. For example, this command line changes the name of the directory **reports** to **progress**:

```
$ chn /owner/reports progress
```

Notice that the *new_name* argument applies to the rightmost component (**reports**) of the *old_name* argument. You cannot use **chn** to change the name of a directory embedded in a pathname.

To append a naming suffix to the new directory name, specify any of the following naming options:

Option	Description
-d	Appends the current month and day to the new name (<i>new_name.mm.dd</i>).
-y	Appends the current year, month, and date to the new name (<i>new_name.yy.mm.dd</i>).
-u	Forces the system to create a unique new name by appending a sequence of number(s) to the end of the name.

If you omit the *new_name* argument, you must specify one of the options in the previous list; the system creates a new name by copying the *old_name* and appending the proper suffix as shown here:

```
$ chn /owner/reports -d
```

When specified on June 16, this command changes the name of the directory **reports** to **reports.06.16**.

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 8-1.

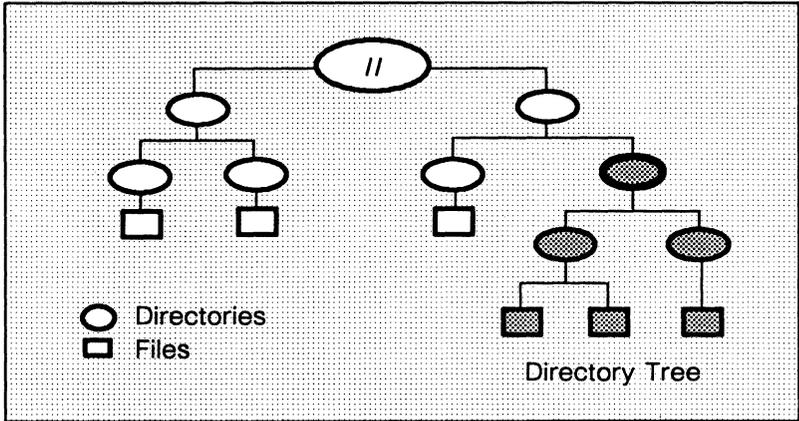


Figure 8-1. Sample Directory Tree

To copy a directory tree to another location, use the **cpt** (copy tree) command in the following format:

cpt *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 8-2 illustrates how the **cpt** command in the following example copies a directory tree:

```
$ cpt reports //boston/user_1/prog -l
```

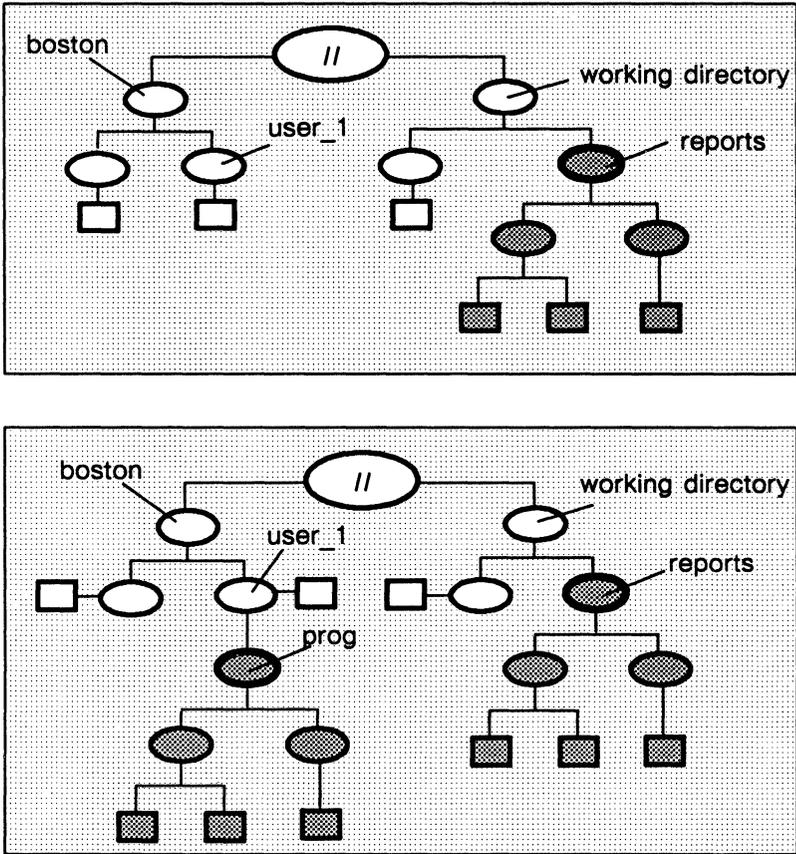


Figure 8-2. Copying a Directory Tree

The `cpt` command creates a copy of the directory tree `reports` and names the copy `prog`. The copy is placed in the directory `user_1`. The `-l` option lists the name of each object as it is copied.

Replacing Directory Trees

To replace one directory tree with another directory tree, specify the `-r` option with the `cpt` (copy tree) command described in the previous section. The `-r` option directs `cpt` to delete the directory

tree specified by the target pathname and to create a copy of the source tree in its place. Figure 8-3 illustrates how the following command replaces a directory tree.

```
$ cpt reports //boston/user_1 -r
```

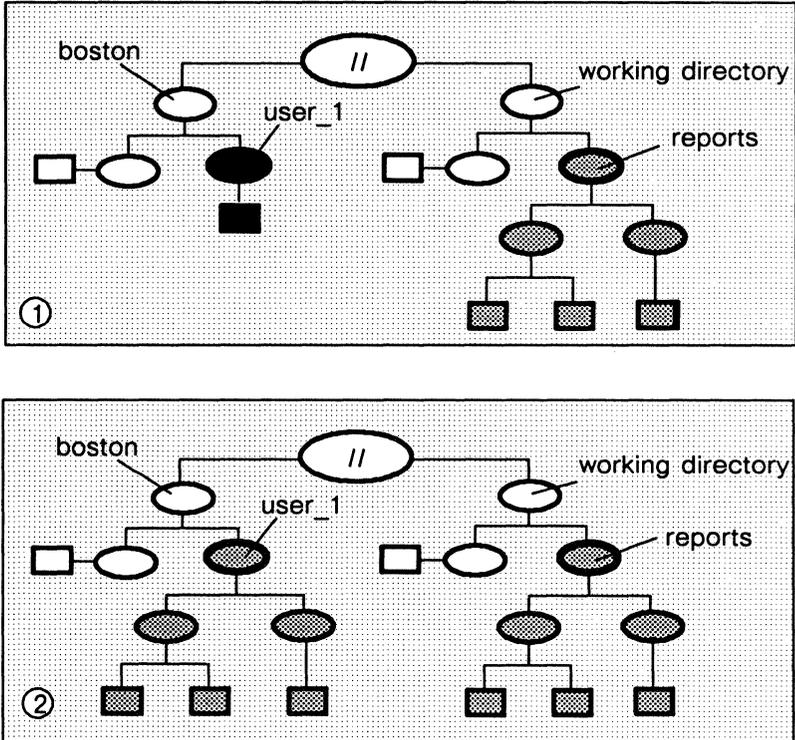


Figure 8-3. Replacing a Directory Tree

The `cpt` command in Figure 8-3 deletes the target tree starting at the pathname `//boston/user_1`. It then replaces the target tree with a copy of the entire `reports` directory tree.

Merging Directory Trees

You can merge directory trees using either the **-ms** or **-md** option with the **cpt** (copy tree) command described in the “Copying Directory Trees” section discussed earlier. When merging directory trees, **cpt** first compares the source and target directories object by object. It then merges the directories according to the option you specified.

When you specify the **-ms** option, **cpt** uses the following process to merge the source directory with the target directory:

- Objects that exist in the source but not in the target are created in the target.
- Objects that exist in the target but not in the source remain unchanged.
- Files and links with the same name in both the source and target are deleted from the target and replaced by the source version.
- Directories with the same name in both source and target are merged.

The **cpt** command continues this process until it reaches the end of the source tree.

The following command merges the source directory **progress** with the target directory **reports**.

```
$ cpt //boston/user_1/progress reports @
$_ -md -l
```

The **-l** option directs **cpt** to list all objects that it creates in the target directory.

If you specify the **-md** option, the merging process is similar to that of **-ms**; however, files and links with the same name in the source and target are left unchanged in the target.

Comparing Directory Trees

To compare the contents of one directory tree to another, use the `cmt` (compare tree) command in the following format:

```
cmt source target [options]
```

The `cmt` command compares all of the objects in the *source* directory tree against all the objects in the *target* directory tree. It reports the following:

- 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, `cmt` ignores the differences.

For example, let's assume that directories `dir_1` and `dir_2` contain the files shown in Figure 8-4.

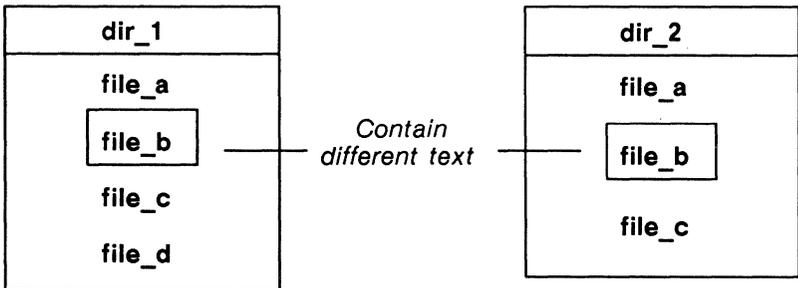
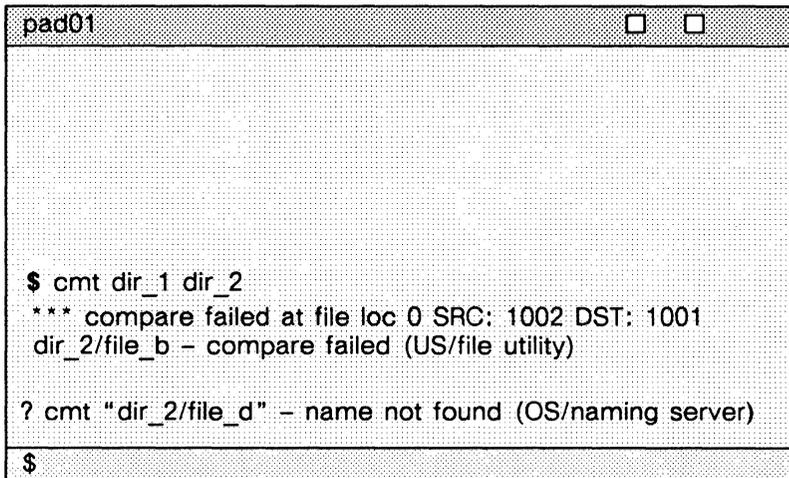


Figure 8-4. Two Sample Directories

Let's also assume that the contents of all the files in `dir_1` and `dir_2` are identical, except for `file_b` which contains different text. Figure 8-4 illustrates how the `cmt` command compares the files in `dir_1` against those in `dir_2`.



```
pad01
$ cmt dir_1 dir_2
*** compare failed at file loc 0 SRC: 1002 DST: 1001
dir_2/file_b - compare failed (US/file utility)
? cmt "dir_2/file_d" - name not found (OS/naming server)
$
```

Figure 8-5. Comparing Directory Trees

Notice in Figure 8-5 that the first message reports a difference between the contents of each directory's **file_b**. The second message reports that **file_d** in **dir_1** did not appear in **dir_2**.

Displaying Directory Information

To list the contents of a directory and report information about the objects the directory contains, specify the **ld** (list directory) command in the following format:

```
ld [pathname...] [options]
```

The *pathname* argument specifies the pathname of the directory, and *options* specifies the types of information you want **ld** to report about the objects it lists. If you omit the *pathname* argument, **ld** lists the contents of the current working directory.

The command in the following example lists the contents of the directory **reports**; the options **direct** tell to report each object's creation date and time, system object type, and rights (ACLs). Below is a sample display produced by the following **ld** command:

```
$ ld /owner/reports -dtr -st -r
```

```
Directory "/owner/reports":
```

sys type	rights	date/time created	name
dir	pgndcalr	85/01/04.09:16	april
dir	pgndcalr	85/01/04.09:16	july
dir	pgndcalr	85/01/04.09:16	june
dir	pgndcalr	85/01/04.09:16	may
file	pgndwrx	85/01/04.09:18	procedure
link			progress
file	pgndwrx	85/01/04.09:16	sample
file	pgndwrx	85/01/04.09:18	template

```
8 entries, 7 blocks used.
```

To list the contents of an entire directory tree, specify the ellipsis wildcard (...) as part of the pathname argument. For example:

```
$ ld /owner/...
```

This command lists the contents of the directory **owner**, as well as the contents of all its subdirectories.

You can also use **ld** to report information about specific files by specifying the pathname of the file as an argument. The “Displaying File Attributes” section discussed in Chapter 7 describes how to use **ld** to report file information.

Deleting Directory Trees

To delete a directory tree, use the **dlt** (delete tree) command in the following format:

```
dlt pathname ...
```

The *pathname* argument specifies the pathname of the directory you want to delete.

NOTE: The pathname you specify does not have to be a directory. If you specify the pathname of a file or link, **dlt** will delete the object with no warning message. To delete files, we recommend that you use the **dlf** (delete file) command; to delete links, use the **dll** (delete link) command.

The **dlt** command deletes the specified directory and all of the objects it contains. For example, the following command deletes the directory tree shown in Figure 8-6:

```
$ dlt reports -l
```

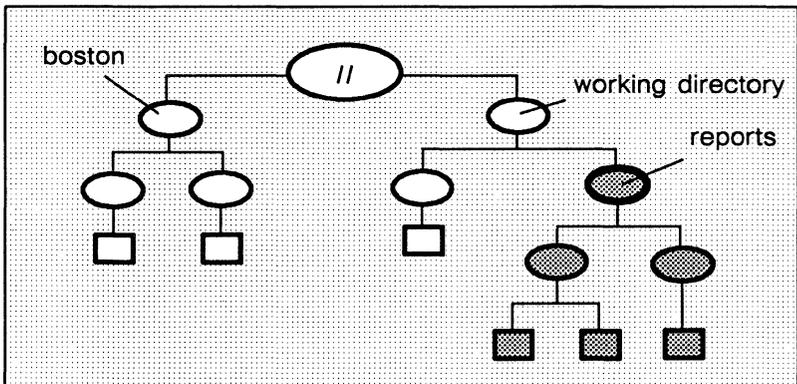
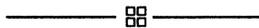


Figure 8-6. Deleting a Directory Tree

The command in the previous example deletes the directory tree starting at the directory **reports** in the current working directory. The **-l** option directs **dlt** to list each object it deletes.



Chapter 9

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**.

A link is a special object that contains the name of another object. When you specify a 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. This section describes how to manage links on your system. Table 9-1 summarizes the commands used to manage links.

Table 9-1. Commands for Managing Links

Task	Shell Command
Create a link	<code>cr1 link_name object_name</code>
Display link resolution names	<code>ld [pathname] -ll -lt</code>
Redefine a link	<code>cr1 link_name object_name -r</code>
Rename a link	<code>chn old_name [old_name]</code>
Copy a link	<code>cpl source [target]</code>
Delete a link	<code>dll link_name</code>

Creating Links

To create a link, specify the **crl** (create link) command in the following format:

```
crl link_name object_name
```

The *link_name* argument specifies the pathname of the link, and *object_name* specifies the pathname of the object to which the link points. The rules for pathnames described in Chapter 1 apply to both arguments.

The following command creates a link:

```
$ crl reports /owner/april/progress_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**. As shown in the following example, when you specify **reports** as a pathname or part of a pathname, the pathname **/owner/april/project_reports** is substituted. For example, instead of specifying

```
$ dlf /owner/april/progress_reports/mr_jones
```

to delete the file **mr_jones**, you could specify the following command line:

```
$ dlf reports/mr_jones
```

You can also use the **crl** command to create more than one link by specifying link name/object name pairs as shown below:

```
$ crl bugs /maintenance/reports starts /sys/dm
      └──────────────────────────┘ └──────────────────┘
                        pair                pair
```

This command creates two links: **bugs** and **starts**.

Displaying Link Resolution Names

To display the resolution names for all the links listed in a particular directory, use the `ld` (list directory) command in this format:

```
ld pathname -lt [-ll]
```

The *pathname* argument specifies the pathname of the directory that contains the link, and `-lt` directs `ld` to display the resolution name of each link. Normally, `ld` lists all the objects in the directory, including files and subdirectories. The `-ll` option directs `ld` to list only the links.

The command in the following example displays the link resolution names for all links in the node entry directory, as shown below:

```
$ ld / -lt -ll

bugs      "/maintenance/reports"
starts    "/sys/dm"
news      "//my_boss/owner/project/status"

30 entries, 3 listed.
```

Redefining Links

You can redefine an existing link by changing its link resolution name. To redefine a link, use the `-r` option with the `crl` (create link) command as follows:

```
$ crl reports /owner/may/progress_reports -r
```

This command replaces the object name for the existing link `reports` that we created in the “Creating Links” section earlier. The new link name points to the subdirectory `may` instead of `april`.

Renaming Links

To change the name of a link, use the **chn** (change name) command in the following format:

```
chn old_name [new_name] [options]
```

The *old_name* argument specifies the pathname of the link you want to rename, and *new_name* specifies the new name of the link. For example, the following command changes the name of the link **reports**, in the current working directory, to **progress**:

```
$ chn reports progress
```

You can specify these naming options with the **chn** command:

Option	Description
-d	Appends the current month and day to the new name (<i>new_name.mm.dd</i>).
-y	Appends the current year, month, and date to the new name (<i>new_name.yy.mm.dd</i>).
-u	Forces the system to create a unique new name by appending a sequence of number(s) to the end of the name.

If you omit the *new_name* argument, you must specify one of the options in the previous list; the system creates a new name by copying the *old_name* and appending the proper suffix as shown here:

```
$ chn reports -u
```

This command changes the name of the link **reports** to **reports.1**.

Copying 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 link, use the **cpl** (copy link) command in the following format:

```
cpl source [target ...] [option]
```

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 **cpl** command always creates a copy of the source link at the location specified by the target. For example:

```
$ cpl 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, **cpl** assigns the copy the name specified by the target, which in this example is **status**.

If the target specifies the pathname of a directory, **cpl** 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:

```
$ cpl reports /user_1
```

copies the link **reports** from the current directory to the target directory **user_1**. Because **cpl** assigns the copy the name of the source link, the new link has the pathname **/user_1/reports**.

To replace an existing link with a copy of another link, use the **-r** option as follows:

```
$ cpl reports /user_2/progress -r
```

This command replaces the link **progress** with a copy of the link **reports** (from the current working directory).

You can copy or replace several links using a single **cpl** command by either specifying multiple pairs of source and target pathnames (each pair separated by a space) or by using pathname wildcards. The following command copies all of the links in the current working directory to the directory **/user_2**.

```
$ cpl ?* /user_2/my=
```

The wildcards (**?***) that make up the source pathname direct **cpl** to copy all links in the current working directory. The wildcard (**=**) in the target pathname directs **cpl** to derive the name of each new link from the source link names. For example, the link **reports** becomes **myreports** in the target directory.

Deleting Links

To delete one or more links, use the **dll** (delete link) command in the following format:

```
dll [link_name ...] [options]
```

The *link_name* argument specifies the pathname of the link you want to delete. If you specify multiple pathnames to delete multiple links, separate each pathname with a space.

The following command deletes the link **reports** from the current working directory:

```
$ dll reports
```

You can also use pathname wildcards to delete related groups of links. For example:

```
$ dll /.../status
```

The ellipsis wildcard (...) directs the **dll** command to delete every link named **status** in all directories subordinate to the node entry directory.



Chapter 10

Controlling Access to Files and Directories

You can protect your files and directories from unauthorized use with a system protection mechanism called an **access control list (ACL)**. Every file and directory in the system has an access control list that defines:

- Who can use the object
- What operations these users can perform on the object

An ACL for a file, for example, can authorize some users to read the file, and permit others to edit it.

This chapter describes the following ACL topics:

- The structure of an ACL and its component parts
- How the system assigns initial ACLs to objects
- How you can use shell commands to display, change, and copy ACLs
- Protected subsystems and the commands you use to create and use them

ACL Structure

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 10-1 shows the elements that make up an ACL entry.

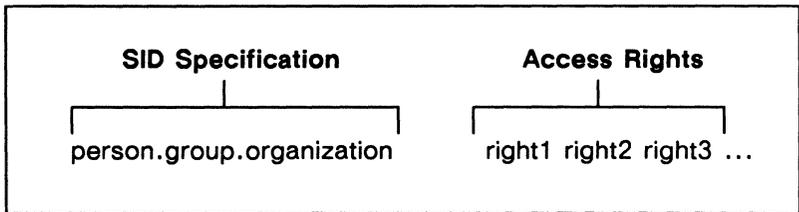


Figure 10-1. Structure of an ACL Entry

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:

person.group.organization

The SID consists of three fields: *person*, *group*, and *organization* (abbreviated **pgo**). 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 10-2 shows a set of two ACL entries for a file.

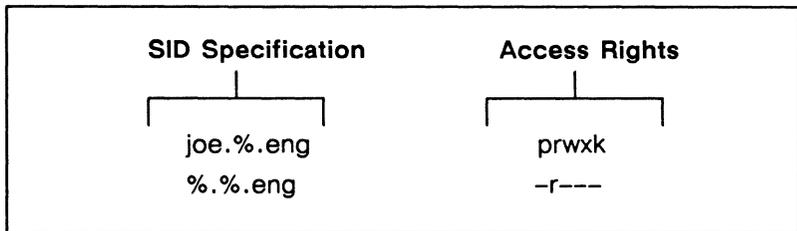


Figure 10-2. Sample ACL Entries

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 10-2 (%.%.eng) contains wildcards in the *person* 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, checking the specific required entry before extended entries of the same type. 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 10-2 (**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**).

As described above, 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). 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 10-3 shows an object with required and extended ACLs.

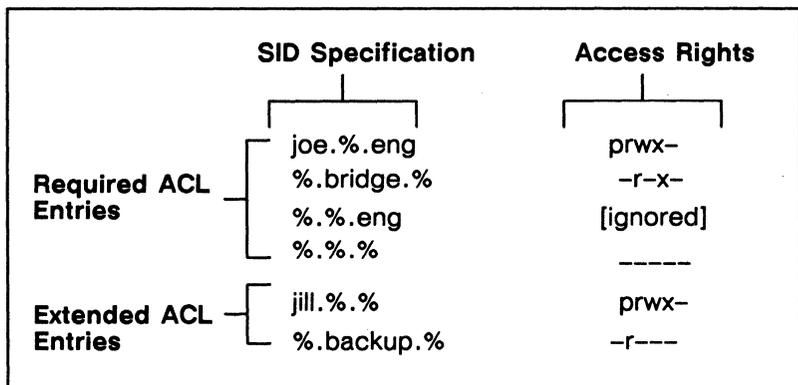


Figure 10-3. Sample Extended ACL Entries

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 10-1 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 not mentioned by other entries:

```
%.%.%                -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 available 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.

Table 10–1 describes the types of access for directories as well as for files.

Table 10-1. 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.
Inherit SID	i	Inherit SID for process (initial ACLs only).	None.

NOTE: To delete a directory tree, you need write rights to the containing directory. If objects are protected with keep rights, you must have protect rights to the object as well.

Searching Directories and Deleting Objects

To access an object, in addition to appropriate rights to the object, you must have appropriate rights to the object's parent directory. To access an object, its parent must grant you search(**x**) rights. To delete an object, its parent must grant you write(**w**) rights. Consider the following example:

\$ `ld /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 delete the subdirectory `reports`, you need write rights to `/owner`.

If **reports** contains additional objects, you need write rights to **reports** to delete them. Therefore, to delete 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 display, edit, and copy an object’s ACL using the following shell commands:

- **acl** (displays and copies ACLs)
- **edacl** (displays and edits ACLs)

Displaying ACLs

To display an object’s ACL, use the shell command **acl** (access control list) in the following format:

```
acl pathname ...
```

The *pathname* argument specifies the pathname of the object whose ACL you want to list. For example:

```
$ acl /owner/report
```

This command lists the ACL entries for the file **report**. Figure 10–4 shows a sample display produced by this command.

Acl for /owner/report:	
Required entries	
jill.%.%	prwx-
%.bridge.%	prw--
%.%.eng	-rw--
%.%.%	-----
Extended entry rights mask: -rwx-	
Extended entries	
user.%.%	---x-
%.netdev.%	-rw--
%.backup.%	-r---

Figure 10-4. Sample ACL Display

By using pathname wildcards, you can list the ACLs for a specific group of objects. For example, the following command lists the ACLs for all the files in the current working directory that have the suffix `.bin`:

```
$ acl ?*.bin
```

You can also display the ACL for an object using the `edacl` (edit access control list) command as follows:

```
$ edacl /owner/report -l
```

The next section, “Editing ACLs,” provides more information on how to use the `edacl` command to display and edit ACLs.

Editing ACLs

You can edit an object’s ACL using the shell command `edacl` (edit access control list). The `edacl` command allows you to display, add, change, and delete ACL entries.

You can also use the **edacl** command to edit a directory's initial ACLs. (The "Editing Initial ACLs" section describes how to use the **edacl** command to edit initial ACLs).

Like most shell commands, you can direct **edacl** to perform specific operations by specifying options on the command line. In addition to its command options, the **edacl** command also accepts a special set of ACL editing commands. The way you specify these editing commands depends on the mode in which **edacl** operates.

The **edacl** command operates in two modes: **command line mode** and **interactive mode**. In command line mode, you specify editing commands as options on the command line. For example, **edacl** in the following example uses the editing command **-l** to display the ACL for the file **report**:

```
$ edacl report -l
```

If you specify the **edacl** command without any editing commands on the command line, **edacl** enters interactive mode and prompts you for editing commands. When you specify editing commands in interactive mode, do not precede the command with a hyphen (-). For example:

```
$ edacl report
report
* l
```

In this example, since no editing commands appear on the **edacl** command line, **edacl** enters interactive mode. Specifying the **l** command (without a hyphen) at the asterisk (*) prompt directs **edacl** to list the ACL entries for **report**.

Once you enter interactive mode, you can continue to specify **edacl** editing commands to perform a series of edit operations. To exit interactive mode and save the changes you've made, type an End-of-File (EOF), normally CTRL/Z. The **q** command quits interactive mode without saving your changes.

You can edit the ACLs of several objects either by specifying multiple pathnames (separating each pathname with a space) or by using pathname wildcards.

This section describes how to use the **edacl** command to list and edit ACLs. The examples presented in this section show how to use **edacl** commands in command line mode. For a complete description of **edacl**, see the *Aegis Command Reference*. Table 10-2 summarizes the commands used to edit ACLs.

Table 10-2. Summary of Commands for Editing ACLs

Task	Command
Display an object's ACL	edacl <i>pathname</i> -l
Add an ACL entry	edacl <i>pathname</i> -a <i>pgo</i> <i>rights</i>
Add rights to an ACL entry	edacl <i>pathname</i> -ar <i>pgo</i> <i>rights</i>
Change access rights for an ACL entry	edacl <i>pathname</i> -c <i>pgo</i> <i>rights</i>
Delete rights from an ACL entry	edacl <i>pathname</i> -dr <i>pgo</i> <i>rights</i>
Delete an ACL entry	edacl <i>pathname</i> -d <i>pgo</i> <i>rights</i>
Set the required entry for person (p)	edacl <i>pathname</i> -p [<i>p</i>] <i>rights</i>
Set the required entry for group (g)	edacl <i>pathname</i> -g [<i>g</i>] <i>rights</i>
Set the required entry for organization (o)	edacl <i>pathname</i> -o [<i>o</i>] <i>rights</i>
Set the required entry for world	edacl <i>pathname</i> -w <i>rights</i>

Rules to Specify ACL Entries

Most of the **edacl** 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 **edacl** 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 10-1 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:

```
$ edacl report -a joe rw  
                  └┘  
                  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 a hyphen character (-) as follows:

```
$ edacl report -a joe -  
                  └┘  
                  no access rights
```

You can also use any of the special class names in Table 10-3 to specify a set of commonly used rights. For example:

```
$ edacl report -a joe -user  
                  └┘  
                  class name
```

The `-user` class name in this example specifies a set of rights that you commonly grant to other users of the system. For both files and directories, `-user` grants all rights to the object except the ability to change the object's ACL. Since the object in this example is a file, `-user` grants read (`r`), write (`w`), and execute (`x`) access.

NOTE: The `edacl` 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.

System users with the project name `backup` may create backup copies of files and directories on magnetic tape. Users with the project name `backup` need read (`r`) access to files, and read (`r`) and execute/search (`x`) access to directories. (Check with your system administrator to determine whether it is necessary to add backup rights to your ACLs.) If the object does require backup rights, edit the ACL again and add an entry that grants the `backup` project (`%.backup.%.%`) read (`r`) access. The next section, "Adding ACL Entries," describes how to add an ACL entry.

Table 10-3. Class Names for Commonly Assigned Rights

Name	Meaning	Directories	Files
-owner	All rights.	pwx	pwx
-user	All rights except ability to change ACL.	wrx	wrx
-read	File read access.	Not allowed.	r
-exec	File read access. Execute access to object files.	Not allowed.	rx
-ldir	List directories.	rx	Not allowed.
-adir	List directories and add entries.	wrx	Not allowed.
-none	Grant no rights.	None.	None.
-ignore	Ignore a required entry for rights.	See Note 1.	See Note 1.
-inh_rights	Inherit rights from current process.	See Note 2.	Not allowed.
-inh_all	Inherit <i>pgo</i> data and rights from current process.	See Note 2.	Not allowed.

Note 1: Each object must have the required entries of *pgo*. However, it is sometimes useful to have these specified by not used for rights checking. This may be done by using the **-ignore** abbreviation (only valid for *pgo* entries).

Note 2: Initial ACLs may be either specified in the directory or may inherit information from the process that is creating the object. Either the SID information or the rights information (or both) may be inherited. The **-inh_rights** and **inh_all** allow this inheritance to be specified. These are only valid for initial ACLs.

Adding ACL Entries

To add an entry (SID and rights) to an ACL, use the **-a** option to the **edacl** command as follows:

```
edacl pathname ... -a pgo rights
```

The *pgo* argument specifies the SID for the new entry and *rights* specifies the set of access rights. The **-a** option directs **edacl** to add the specified SID and access rights to the ACL. For example:

```
$ edacl report -a %%.man -owner
```

The command in this example adds a new ACL entry to the ACL for the file **report**. The **-owner** rights class name (see Table 10-3) specifies a full set of rights for the entry. The new entry grants full access (**prwx**) to anyone in the organization named **man**.

Changing Entry Rights

To change the access rights for an SID, use the **-c** option to the **edacl** command as follows:

```
edacl pathname ... -c pgo rights
```

The *pgo* argument specifies the SID for the entry you want to change, and *rights* specifies the new set of access rights. The **-c** option directs **edacl** to change the access rights for the specified SID.

For example, suppose the file **report** has the following ACL entry granting full rights:

```
%%.man                prwx
```

The following command changes the access rights for **%%.man** to read (**r**) access:

```
$ edacl report -c %%.man r
```

As a result, the new ACL entry now looks like this:

```
%.%.man.%          -r---
```

If you try to change the access rights for an entry that doesn't exist, you will receive an error message.

Adding Entry Rights

To add access rights to an existing ACL entry, use the `-ar` option to the `edacl` command in the following format:

```
edacl pathname ... -ar pgo rights
```

The *pgo* argument specifies the SID for the entry you want to change, and *rights* specifies the new set of access rights. The `-ar` option directs `edacl` to add rights to the existing list of access rights for the specified SID.

For example, suppose the file `report` has the following ACL entry:

```
%.%.man          -r---
```

The following command adds write(`w`) and execute(`x`) rights to the current access rights for `%.%.man`.

```
$ edacl report -ar %.%.man wx
```

As a result, the ACL entry now looks like this:

```
%.%.man          -rwx-
```

If you try to add rights to an entry that doesn't exist, you will receive an error message.

Deleting Entry Rights

To delete the set of rights from a particular ACL entry, use the **-dr** option to the **edacl** command in the following format:

```
edacl pathname ... -dr pgo rights
```

The *pgo* argument specifies the SID for the entry you want to change, and *rights* specifies the access rights you want to delete. The **-dr** option directs **edacl** to delete the access rights for the specified SID.

For example, suppose the file **report** has the following ACL entry:

```
%.%.man                -rwx-
```

The following command deletes write (**w**) access from the current access rights for **%.%.man**:

```
$ edacl report -dr %%.man w
```

As a result, the ACL entry now looks like this:

```
%.%.man                -r-x-
```

If you try to delete rights from an entry that doesn't exist, you will receive an error message.

Deleting ACL Entries

To delete an entry (SID and rights) from an ACL, use the **-d** option to the **edacl** command in the following format:

```
edacl pathname ... -d pgo
```

The *pgo* argument specifies the SID for the entry you want to delete. For example:

```
$ edacl report -p %.%man.%
```

This command deletes the entry `%.%.man` from the ACL for the file `report`.

NOTE: You may not delete a required entry. You should either set the rights to “ignore” or specify a new required entry, using the `p`, `g`, `o`, or `w` commands.

Setting Required Entries

To set a required entry (SID and rights) for a specified person, use the `-p` option to the `edacl` command in the following format:

```
edacl pathname ... -p person rights
```

The *person* argument specifies the name of the user for whom you want to specify rights. For example:

```
$ edacl report -p mary.%.% -owner
```

This command assigns `mary` owner rights for the file `report`.

To set a required entry for a specified group, use the `-g` option to the `edacl` command; to set a required entry for a particular organization, use the `-o` option.

Copying ACLs

To copy an ACL from one object to another, use the `acl` (access control list) command in the following format:

```
acl target source
```

The *target* argument specifies the pathname of the object to which you want the ACL copied. The *source* argument specifies the pathname of the object whose ACL you want to copy.

The following command copies the ACLs from the directory `/owner` to the directory `/user_1`:

```
$ acl /user_1 /owner
```

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 10-5 shows how the system assigns initial ACLs to files and directories.

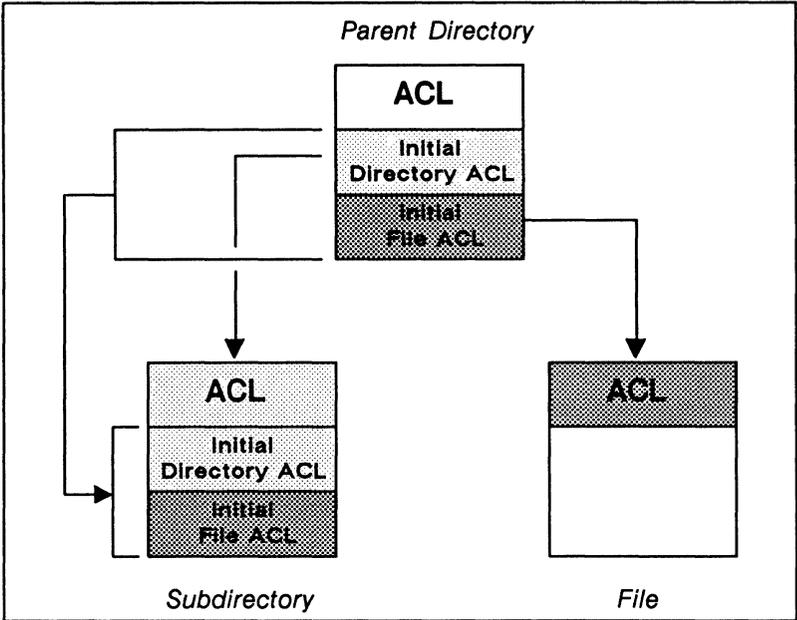


Figure 10-5. Initial ACLs for Files and Directories

Table 10-4 summarizes the commands used to change and copy initial ACLs.

Table 10-4. Summary of Commands for Editing and Copying Initial ACLs

Task	Command
Edit initial directory ACL	<code>edacl pathname -id command</code>
Edit initial file ACL	<code>edacl pathname -if command</code>
Copy both initial ACLs	<code>acl target source -l</code>
Copy initial directory ACL	<code>acl target source -id</code>
Copy initial file ACL	<code>acl target source -if</code>

Editing Initial ACLs

To edit a directory's initial ACL, use **edacl** with the **-id** option in the following format:

```
edacl pathname -id -command
```

The **-id** option directs **edacl** to edit initial directory ACLs, and *-command* specifies one of the ACL editing commands described in the "Editing ACLs" section discussed earlier. For example:

```
$ edacl /owner -id -l
```

The command in this example displays the initial directory ACL for the directory **/owner**.

To add an entry to the initial directory ACL for **/owner**, use the **-a** option as follows:

```
$ edacl /owner -id -a %.%eng rwx
```

The following example uses the **-dr** option to take away write (w) rights from the entry we added in the previous example:

```
$ edacl /owner -id -dr %.%eng w
```

To edit the initial file ACL, use the **edacl** command with the **-if** option in the following format:

```
edacl pathname -if -command
```

The **-if** option directs **edacl** to edit initial file ACLs, and *-command* specifies one of the ACL editing commands described in the "Editing ACLs" section discussed earlier. For example:

```
$ edacl report -if -l
```

The command in this example displays the initial file ACL for the file **report**.

Copying Initial ACLs

You can copy a directory's initial ACLs using the `acl` command in the following format:

```
acl target source option
```

The *option* argument specifies one of the options listed in Table 10-5. The *target* argument specifies the pathname of the object to which you want the initial ACL copied. The *source* argument specifies the pathname of the object whose initial ACL you want to copy.

Table 10-5. Options for Copying Initial ACLs

Option	Description
<code>-i</code>	Copies both the initial file and initial directory ACLs from the source to the target.
<code>-id</code>	Copies the initial directory ACL from the source to the target.
<code>-if</code>	Copies the initial file ACL from the source to the target.

The command in the following example uses the `-i` option to copy the initial file and directory ACLs from the directory `/owner` to the directory `/user_1`.

```
$ acl /user_1 /owner -i
```

To copy only the initial file ACL, use the `-if` option as shown in the following example:

```
$ acl /user_1 /owner -if
```

For a complete description of how to use the `acl` command to copy initial ACLs, see the *Aegis Command Reference*.

Protected Subsystems

Another method of controlling access to files is through a protection mechanism called a **protected subsystem**. Protected subsystems allow you to designate a collection of data (a protected group of files) for use solely by specific programs.

A protected subsystem is composed of one or more programs and a set of data files. The programs are called the **managers** of the protected subsystem; the data files, called **data objects** (or **protected objects**), are owned by the subsystem. Thus, files in a protected subsystem have either manager or data object status.

Protected subsystems permit broad groups of users to access data objects through the programs, or managers, of the subsystem. You typically create a protected subsystem when you want only specific programs to act on data files, regardless of the SIDs of the processes in which the programs run.

For example, you might have a group of data files produced and used by a specific program. If you want to prevent these files from being used for any other purpose, you can assign protected subsystem status to both the program and the data files. As a result, only those users authorized to run the subsystem manager program can use the files protected by the subsystem.

This section explains how to create a protected subsystem and how to assign subsystem status to files.

How Protected Subsystems Work

In order to understand how to assign subsystem status to files, you must first understand how the system handles protected subsystems. Figure 10-6 presents a flowchart that shows how the system controls access to protected subsystem files.

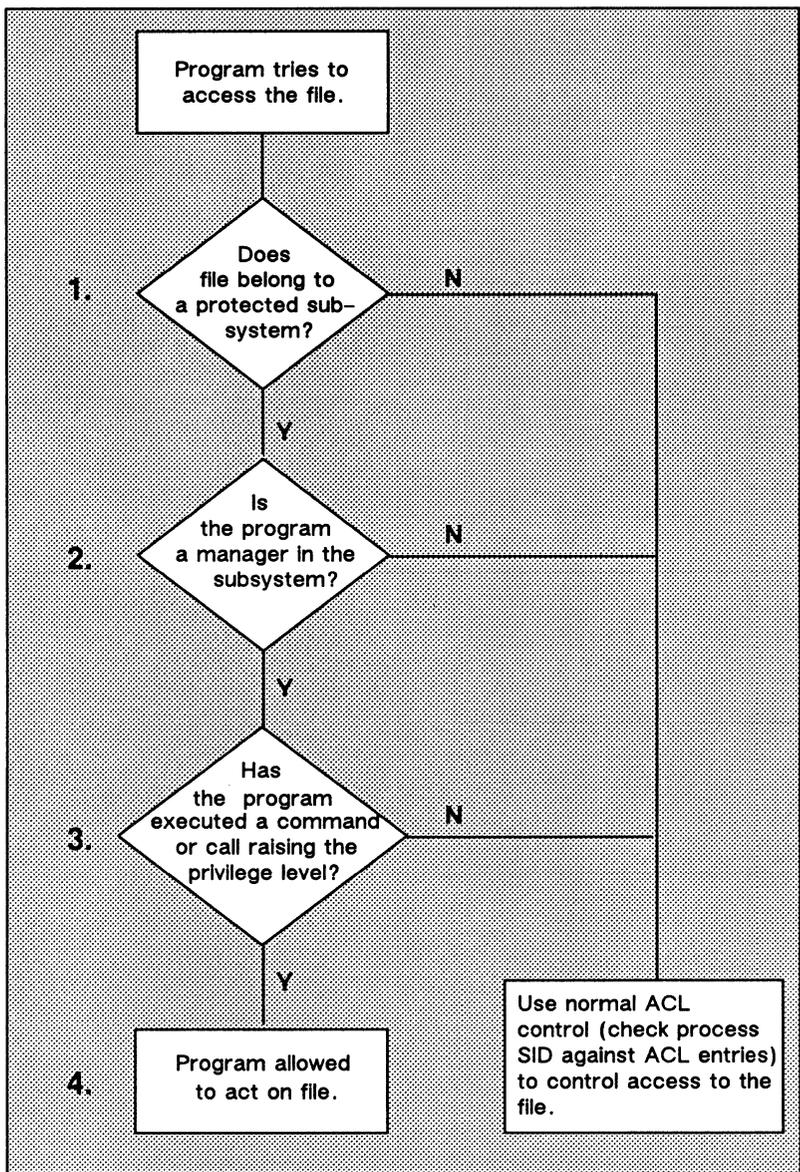


Figure 10-6. Controlling Access to Protected Subsystem Files

The following descriptions explain the sequence of events shown in Figure 10-6:

1. When a program in a protected subsystem requests access to a file, the system first checks whether the file belongs to a protected subsystem. If the file does *not* belong to a protected subsystem, the system uses the file's ACL information to control access.
2. If the file does belong to a protected subsystem, the system determines whether the requesting program owns the file (whether the program is a manager in that subsystem). If the program is not a manager in that subsystem, the system treats it like any other program and uses ACL information to control access.
3. If the program *is* a manager in that subsystem, the system verifies that the program has executed a command or system call that raises the manager program's privilege level. If a manager program hasn't raised its privilege level, the system treats it like a non-manager program and uses ACL information to control access.

The system allows you to raise a program's privilege level by using either the shell command `subs` (subsystem) or a set of programming calls. For more information, see the `subs` command in the *Aegis Command Reference* or the `aclm` call descriptions in the *Domain/OS Call Reference*.

4. If the manager program has raised its privilege level, the system allows it to operate on the file.

To use a protected subsystem, you must first create it, and then enter it to add files. The following sections describe how to create and enter a protected subsystem.

Creating a Protected Subsystem

To create a protected subsystem, use the `crsubs` (create subsystem) command in the following format:

```
crsubs subsystem_name
```

The *subsystem_name* argument specifies the name you want to assign to the subsystem. For example, the following command creates a protected subsystem named **protector**:

```
$ crsubs protector
```

When you create a protected subsystem, the system assigns it the subsystem name that you specify. The system also assigns the subsystem name to a subsystem shell in the node's */sys/subsys* directory. The subsystem shell is actually a copy of the shell program. This shell program is the first manager program in your newly-created subsystem.

The operating system uses the managers in the */sys/subsys* directory when it checks for the names of protected subsystems. Internal to the ACL for each of these managers, and to the ACL for any file, is a field for protected subsystem status. Only the operating system can see this field. If the file belongs to a protected subsystem, the field contains an internal identifier for that subsystem. All files in a particular subsystem, including the files in */sys/subsys*, have the same internal identifier.

When you display an object's ACL (see the "Displaying ACLs" section discussed earlier), the system looks at the ACLs subsystem field. If the field contains a subsystem identifier, the system looks in */sys/subsys* for a file with the same internal identifier. The system then displays the name of that file as the name of the subsystem.

To use **crsubs** to create a protected subsystem, you must have write rights to the */sys/subsys* directory. The initial file ACL for this directory must also grant read and execute rights to any file created in */sys/subsys*. You should normally limit these rights to the creator of the subsystem or to the system administrator.

Assigning Protected Subsystem Status

Before you can assign subsystem status to files, you must first enter the subsystem using the **ensubs** (enter subsystem) command in the following format:

```
ensubs subsystem_name
```

The *subsystem_name* argument specifies the name of the subsystem you want to enter. (To use **ensubs** to enter a subsystem, you must have read and execute access to the subsystem file in */sys/subsys*.) For example, the following command lets you enter the subsystem named **protector**:

```
$ ensubs protector
```

When the dollar sign prompt appears after you specify **ensubs**, you are “inside” the subsystem. Once inside, you can assign manager or data object status to files using the **subs** (subsystem) command in the following format:

```
subs pathname subsystem_name option
```

The *pathname* argument specifies the name of the file, and *subsystem_name* specifies the name of the current subsystem. The *option* specifies either **-mgr** for manager status or **-data** for data object status. For example:

```
$ subs prog_259 protector -mgr
$ subs data_1 protector -data
$ subs data_2 protector -data
```

The commands in this example assign subsystem status to files of the subsystem **protector**. The first command assigns manager status to the program file **prog_259**. (You can assign manager status to either a binary program or a script.) The remaining commands assign data object status to the files **data_1** and **data_2**.

When you’re finished assigning status to files, you can leave the protected subsystem by typing an End-of-File (normally CTRL/Z). You’ve exited the protected subsystem when the EOF marker appears and the dollar sign prompt returns.

```

# Create the subsystem.
$ crsubs protector
# Change ACL entries for the subsystem.
$ edacl /sys/subsys/protector -p fran -owner
$ edacl /sys/subsys/protector -cf %.sys_admin -owner
$ edacl /sys/subsys/protector -cf %.%.% -none
# Check to make sure entries are right.

$ acl /sys/subsys/protector
Acl for /sys/subsys/protector:
Subsystem protector manager
Required entries
fran.%.% pr-x-
%.sys_admin.% pr-x-
%.%.r_d [ignored]
%.%.% -----
Extended entry rights mask: -----

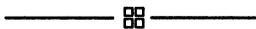
# Enter the subsystem.
$ ensubs protector
# Assign subsystem status to two files.
# The files must already exist.
$ subs /owner/my_prog protector -mgr
$ subs /owner/data_1 protector -data
# List the subsystem status to check for mistakes.
$ subs /owner/my_prog
"/owner/my_prog" is a protector subsystem manager
"/owner/my_prog" is a file subsystem data object
$ subs /owner/data_1
"/owner/data_1" is a nil subsystem manager
"/owner/data_1" is a protector subsystem data object

# Type an End-of-File (normally CTRL/Z to exit
# the subsystem.
$ ***EOF***
$

```

Figure 10-7. Sample of a Protected Subsystem Transcript

Figure 10-7 contains a transcript that shows how a user created a protected subsystem, entered it, created a subsystem manager and data object, and exited the subsystem.



Chapter 11

Writing Shell Scripts

Most of the shell command examples that you've seen so far show you how to use commands interactively by specifying them in the shell input pad. You can also use shell commands in **shell scripts**. Shell scripts are essentially programs made up of shell commands and other valid shell characters, operators, and expressions. Think of scripts as programs written in the "shell language".

This chapter describes how to write shell scripts using shell commands, operators, and expressions. Although you can use many of the commands and conventions presented in this chapter when you use the shell interactively, they have their most practical applications in scripts.

Creating Your Own Commands

In its simplest form, a script is a file containing shell commands that you create to perform some customized operation. For example, a shell script can contain a sequence of commands that you specify frequently, such as `wd` to set the working directory, and `ld` to list the directory's contents. Or the script could contain a single command with a long list of options. By including commands such as these in a script, you can execute them at any time by specifying a single command name.

For example, when you use the `ld` command to list the contents of a directory, it displays only the name of each object by default. Suppose, however, that you want to display each object's access rights, creation date, and object type. Normally, each time you type the `ld` command you have to specify the same list of options. Instead, you can create a shell script named `list` that contains this command line:

```
ld -r -drc -st
```

Whenever you specify the command name `list`, the shell lists the access rights, creation date, and object type of each object in the current working directory.

Of course, you can write much more complicated scripts that perform more sophisticated tasks. This section describes some of the basic components for writing scripts. Refer to Appendix C for examples of more complex shell scripts.

Creating Scripts

To create a script, create a file and insert shell command lines. Command lines in scripts use the same command line format described in Chapter 6.

Like commands that you enter in the shell input pad, you can use parsing operators such as the semicolon (;) to separate commands on a command line, and the escape character (@) to continue a command on more than one line. Other operators, like the pound sign character (#), have functions more suited for use in scripts. The # character lets you include comments in your scripts, since it directs the shell to ignore anything that follows it on the command line. Table 11-1 lists the shell parsing operators you'll use when writing scripts.

Table 11-1. Shell Parsing Operators

Character	Function
#	Direct the shell to ignore anything that follows it on the command line.
;	Separate commands on a line.
&	Run a command or program in the background without pads and windows.
^ <i>n</i>	Substitute <i>n</i> th parameter (<i>n</i> is a number).
^*	Substitute all parameters (not including the command itself).
! <i>n</i>	Substitute parameter for <i>n</i> (a number) and rescan it.
!* 'string'	Substitute and rescan all parameters (not including command name itself).
"string"	Quoted string, no parameters inserted.
@	Quoted string, parameter may be inserted.
	Escape character
	Space (separates arguments).

An important consideration when creating scripts is where to create them. Remember, when you specify a command name, the shell searches for the corresponding file according to a set of command search rules. By default, the second directory the shell searches is your personal command directory `~/com`. Therefore, you should normally create your own personal scripts there. In fact, all of the examples in this chapter assume that the scripts reside in your `~/com` directory. For more information on command search rules, refer to Chapter 6.

Passing Arguments to Scripts

Let's take a look at a slightly more sophisticated script. This script is in a file called **compile** and contains the following lines:

```
# compile
#
# This file compiles and binds prog
#
pas prog -l -map -opt
bind prog.bin -map >prog.map
args "prog compiled and bound."
```

When you specify **compile** in the shell input pad, the shell executes the script. The script compiles and binds the program in file **prog** and produces various output files (listings and maps), all in the current working directory. When finished the script uses the **args** command to display the message "prog compiled and bound" in the transcript pad.

The **args** command uses standard output to write its arguments (one per line) to the shell transcript pad. You can use the **args** command in scripts to display the results of expressions (see the "Using Expressions" section) or to display messages and diagnostics (as in the previous example). In fact, many of the examples in this chapter use the **args** command to show how the shell evaluates various strings and expressions. You can also use the **args** command with the **-err[out]** option to write arguments to error output.

The shell script **compile** isn't very useful, since it only operates on a single file named **prog** and performs fixed compilation and binding operations. A script is more versatile if you can pass arguments to it when you specify the command to invoke it. Consider the following script named **compile2**:

```

#
# compile2
#
# This file compiles and binds a program whose name
# you pass to it as (^1).
#
pas ^1 -l -map -opt
bind ^1.bin -map >^1.map
args "^1 compiled and bound."

```

Specifying the following command in the shell input pad causes the shell to find and execute the script **compile2**:

```
$ compile2 test_prog
```

The shell substitutes **test_prog**, which is the first argument on the command line, for every occurrence of the (^1) in the script. As a result, the script compiles **test_prog**, binds **test_prog.bin**, writes a map to **test_prog.map**, and when complete, writes the message:

```
test_prog compiled and bound.
```

Arguments that you type on the command line correspond to symbols in the script, called **substitution parameters**. Each substitution parameter is composed of a caret character (^) and a number. The caret character (^) instructs the shell to substitute an argument for the parameter; the number refers to the position the argument occupies on the command line that invoked the script.

In the previous example, ^1 refers to **test_prog**, which is the first argument after the command name **compile**. You can use any number of substitution parameters in shell scripts (beginning with ^0 which refers to the command name itself).

Our **compile2** script is still very specific, since the compile and bind operations are still fixed. To make those operations variable, simply pass in more parameters. Consider the following script named **compile3**:

pile3. However, this still won't work, because the shell tries to interpret the entire string as a single argument, instead of the three separate options the string represents.

Let's look at a fourth and final compile script to see how to solve our problem.

```
#
# compile4
#
# ***** (This example is correct) *****
#
# This file compiles and binds a program whose name
# you pass to it as ^1, and whose options you pass
# to it as !2
#
pas ^1 !2
bind ^1.bin -map >^1.map
args "^1 compiled and bound."
```

Now, typing the following command will work:

```
$ compile3 my_prog ' -l -map -opt '
```

The diagram shows the command `$ compile3 my_prog ' -l -map -opt '`. A horizontal bracket is drawn under the text `my_prog`, with a vertical line extending down to the number `1`. Another horizontal bracket is drawn under the text `' -l -map -opt '`, with a vertical line extending down to the number `2`.

Like the caret (^), the exclamation point (!) parsing operator causes the shell to substitute the string in quotation marks for the second parameter. However, the exclamation point directs the shell to rescan the command line before executing it. When the shell scans the line a second time it breaks apart the three options in the string. As a result, the shell interprets the options correctly.

Using Quoted Strings

The proper use of quotation marks can make a big difference in the way the shell interprets quoted strings. To use quoted strings correctly in scripts, you must understand the subtle differences in the shell's interpretation of single and double quotation marks.

When you want the shell to interpret a string literally, you can use either single or double quotation marks as follows:

```
args 'compiled and bound'
```

or

```
args "compiled and bound"
```

Both commands use standard output to write the message, "compiled and bound" to the shell's transcript pad. But suppose you wanted to substitute arguments inside the quoted string.

To substitute arguments inside a quoted string, you *must* use double quotation marks. For example, let's use a line from the script, **compile4** that we created in the "Passing Arguments to Scripts" section.

```
args "^1 compiled and bound"
```

When you use double quotes, the shell performs substitutions in the quoted string. In this example, if the argument passed to the script is **test_prog**, the shell outputs the string:

```
test_prog compiled and bound.
```

On the other hand, if you enclose the string in single quotation marks:

```
args '^1 compiled and bound.'
```

the shell will not perform the substitution. Instead, it displays the message:

```
^1 compiled and bound.
```

Using In-Line Data

Certain shell commands use standard input to read data from the shell input pad. When you use these commands in scripts, you can redirect standard input to read data from within the script itself.

For example, the **ed** command normally uses standard input to read special editing commands that you enter in the shell input pad.

Using the I/O redirection character (<<), you can redirect standard input to read commands from inside the script instead. Following is a script in which the `ed` command reads in-line data.

```
#
# This is a sample script that uses in-line data
#
ed my_file <</
editing commands
.
.
.
/
```

In the example above, the list of editing commands between the two slash characters (`/`) is called a **here document**. The I/O redirection character (<<) redirects standard input to read the data (in this case commands) contained in the here document.

The sample script above uses a slash character (`/`) as a delimiter to indicate both the beginning and end of the here document. You can use any character as a delimiter, as long as the beginning and ending characters are the same. Also, in order for the shell to recognize the end of the here document, you must specify the ending delimiter as the first and only character on the line.

Executing DM Commands from Shell Scripts

You can invoke DM commands from the command shell or from within a shell script using the shell command `xDMc` as follows:

```
xDMc dm_command
```

The *dm_command* argument specifies the name of the DM command you want to execute. For example:

```
xDMc cv news
```

This command executes the DM command `cv` to open a read-only pad and window for the file `news`.

Debugging Shell Scripts

Normally, when a script runs, it doesn't display commands as it executes them. As a result, when a script doesn't work, it is difficult to locate which command or commands cause the errors.

To debug a shell script, invoke the script using the `sh` command in the following format:

```
sh option script
```

The *script* argument specifies the name of the script, and *option* specifies one of the options in Table 11-2. Each option activates a specific function.

The following command executes the script `compile` and writes each command line to standard output immediately before execution:

```
$ sh -x compile
```

Table 11-2. Script Verification Options

Option	Function
<code>-x</code>	Writes each command line in the script to standard output immediately before execution. Provides the complete pathname for each command and evaluates all expressions.
<code>-v</code>	Writes each command line in the script to standard output. Each variable is expanded, but expressions are not evaluated and command pathnames are not expanded.
<code>-n</code>	Interprets commands without actually executing them.

If you want to turn either of these features on or off without using the `sh` command options, you may specify the shell commands `von`, `voff`, `xon`, or `xoff` and then run your script directly using the current shell. For example, the following are equivalent:

```
$ sh -x compile
```

or

```
$ xon
$ compile
$ xoff
```

You can also include these commands within the script itself to enable or disable verification. For example, to debug part of a script, you can place `xon` and `xoff` commands around the segment of the script you want to debug. Or, to debug an entire script, include the `xon` command as the first line in the script. When the script completes, control of verification returns to the shell.

Using Expressions

Like programs written in a high-level programming language such as FORTRAN or Pascal, scripts allow you to use expressions to perform mathematical, string, and Boolean operations. Table 11-3 provides a summary of the operators you can use in expressions.

To evaluate an expression, you must enclose the expression within **expression delimiters** (a set of double parentheses) as follows:

```
args (( 4 + 2 ))
```

The only exception to this rule is the case where you use the assignment operator (`:=`) to assign an expression to a variable:

```
total := 4 + 2
```

In this example, the shell evaluates the expression and assigns the resulting value to the variable `total`. While the assignment operator doesn't require you to use expression delimiters, you can use them

if you prefer; no error will occur if you do use them. The “Defining Variables” section describes how to use the assignment operator to assign values to variables.

Operands in Expressions

You can use any of the following as operands in expressions:

- Single integer, string, or Boolean values
- Operations that result in integer, string, or Boolean values
- Variables assigned integer, string, or Boolean values (the “Shell Variables” section describes variables).

Certain types of operations in expressions take precedence over others. For example, the shell will perform a mathematical operation in an expression before a comparison operation. As shown in Table 11-3, the shell performs operations according to a specific order of precedence where 1 is the lowest (last performed) and 9 is the highest (first performed).

The last operation performed in an expression (the operation with the lowest precedence) determines the type of value, either integer, string, or Boolean, returned by the expression.

When you create expressions, refer to Table 11-3 to check the precedence of the operators you use. Understanding the order in which the shell performs operations will reduce the possibility of an expression resulting in an unexpected answer. Many of the examples that we’ll see in this chapter demonstrate operator precedence.

Table 11-3. Summary of Expression Operators

Type	Char.	Function	Legal Operands	P
Grouping Operators	()	Group operations	Any value	8
Math Operators	+	Positive value	Integer	7
	-	Negative value	Integer	7
	**	Op1 to the Op2	Integers	6
	Mod	Mod Op1 by Op2	Integers	5
	*	Multiply	Integers	4
	/	Divide	Integers	4
	-	Subtract	Integers	3
String Operators	+	Concatenate	Strings	3
	-	Subtract last occurrence of Op2	Strings	3
Math or String Comparison Operators	=	Compare for equality	Integer or string	2
	<	Less than	Integer or string	2
	>	Greater than	Integer or string	2
	<=	Less than or equal to	Integer or string	2
	>=	Greater than or equal to	Integer or string	2
	<>	Not equal	Integer or string	2
Logical Operators	or	Logical or	Boolean	1
	and	Logical and	Boolean	1
	not	Logical negate	Boolean	9

Mathematical Operators

Use mathematical operators in expressions to perform calculations on integers. The result of a mathematical operation is always an integer. For example:

```
args (( 5 + 4 * 3 - 2 ))
```

returns the value 15. If you're wondering why the answer isn't 9 (9 times 1), the reason is that the shell performs multiplication operations in this expression before it performs addition and subtraction operations. In our example, the shell multiplied 4 by 3 before it added 5 and subtracted 2.

To perform the addition and subtraction first, you could use the **grouping operators** (parentheses) to group the addition and subtraction operations within the expression as follows:

```
args (( ( 5 + 4 ) * ( 3 - 2 ) ))
```

The shell always performs operations inside parentheses first, from left to right. In this example, the shell first adds 5 and 4 and subtracts 2 from 3, and then multiplies the resulting values. Table 11-3 lists the order of precedence for all operators where 1 is the lowest and 9 is the highest precedence.

Since all mathematical operators perform integer arithmetic, expressions always result in whole numbers; the shell truncates fractional values.

String Operators

Use string operators to either concatenate or reduce strings. For example:

```
args (( "file" + ".pas" ))
```

uses the plus sign (+) operator to concatenate two strings and form the string **file.pas**.

Using the minus sign (-) operator to reduce a string is a little trickier. Let's look at a simple example first:

```
args (( "file.pas" - ".pas" ))
```

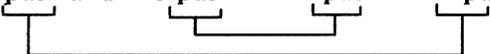
This operation subtracts the second operand from the first operand to return the string `file`. The behavior of the minus sign operator gets more complicated when the first operand contains more than one occurrence of the second operand. In this case, the shell string you are subtracting matches the last occurrence in the first operand. For example:

```
args (( "prog.pas and file.pas" - ".pas" ))
```

A diagram consisting of two horizontal brackets. The first bracket is positioned under the text "file.pas" in the command above. The second bracket is positioned under the text ".pas" in the command above. A vertical line connects the right end of the first bracket to the left end of the second bracket, indicating that the ".pas" at the end of "file.pas" is the one being subtracted.

This expression subtracts the last occurrence of the second operand (`.pas`) from the first operand. The result of this example is `prog.pas and file`. To subtract both occurrences of the string `.pas` in the first operand, use the following expression:

```
args (( "prog.pas and file.pas" - ".pas" - ".pas" ))
```

A diagram consisting of two horizontal brackets. The first bracket is positioned under the text "prog.pas" in the command above. The second bracket is positioned under the text ".pas" in the command above. A vertical line connects the right end of the first bracket to the left end of the second bracket, indicating that the ".pas" at the end of "prog.pas" is the one being subtracted.

This expression performs two operations, each subtracting the last occurrence of the string `.pas` in the first operand. The result is the string `prog and file`.

When you use string operators, the shell treats all operands as strings. If an operand in a string operation is an integer, the shell converts the integer to a string. For example:

```
args (( 50 + "shares at $" + 30 + "a share is $" @  
+ (50 * 30) ))
```

returns the string

```
50 shares at $30 a share is $1500
```

Notice that the shell performs the mathematical operation inside the grouping operators first. The result of this operation ($50 * 30$) is the integer 1500. Since this integer is part of a string concatenation operation, the shell converts it to a string. Even if you omitted the grouping operators, the shell would still multiply the two integers first, since multiplication operations have a higher precedence than string concatenation operations (see Table 11-3).

Comparison Operators

Use comparison operators to compare either integer or string values. The result of a comparison operation is always a Boolean value (true or false). The following expression compares two integers:

```
args (( 5 > 2 ))
```

This expression results in the Boolean value **true**, because the integer 5 is greater than the integer 2.

When you compare strings, the shell compares them according to the sequential position they hold in the ASCII character set. For example:

```
args (( a < b ))
```

results in the value **true** because **a** holds a lower position than **b** in the character set. Also, the shell is case-sensitive when comparing strings. For example, the following expression results in the value **false**:

```
args (( A = a ))
```

Logical Operators

Use logical operators to perform logical operations with Boolean values. The result of a logical operation is always a Boolean value. For example:

```
args (( 5 > 2 or 5 > 6 ))
```

results in the value **true**. In this example, the first operand (the result of the integer comparison) is true, while the second operand is false (5 is not greater than 6). With the **or** operator, if either one of the operands results in the value **true**, then the result of the operation is **true**.

When you use the **and** operator, both operands must be true for the operation to result in a **true** value. For example:

```
args (( 5 > 2 and 5 > 6))
```

This expression results in the value **false** because both operands are not **true**; the second operand is **false**.

Shell Variables

You use variables in shell scripts as symbolic names for specific integer, string, or Boolean values. Once you assign a value to a variable name, you can refer to that value in the script by its variable name rather than its actual value.

The shell allows you to use variables in any of the following:

- Command lines as commands, arguments, or options
- Here documents
- Strings enclosed in double quotation marks
- Expressions

Defining Variables

To define a variable, use the assignment character (**:=**) in the following format:

```
variable_name := value
```

The *variable_name* field specifies the name of the variable, and *value* specifies the value you want to assign to the variable. Variable names can contain alphanumeric characters, as well as the underscore (`_`) and dollar sign (`$`). Names can be up to 1023 characters in length. You must, however, begin all variable names with a letter. (Variables are not case-sensitive.) The following statement assigns the integer value 30 to the variable name `work_days`:

```
work_days := 30
```

Unlike many programming languages that require you to declare variable types, the shell automatically assigns the variable a type based on the assigned value. In the previous example, since the value 30 is an integer, the shell assigns the variable `work_days` the type `integer`. Table 11-4 lists the rules the shell uses to assign variable types.

Table 11-4. Rules for Assigning Variable Types

Type	Assignment Rule
Integer	When the assigned value is an integer, constant, an integer expression, or another integer variable, e.g., <pre>int :=7</pre> or <pre>int :=5 + (4-2)</pre>
String	When the assigned value is a quoted string, a string constant, a string expression, or another string variable, e.g., <pre>str := "april" + ^var2</pre>
Boolean	When the assigned value is a Boolean constant (true or false), a Boolean expression, or another Boolean variable, e.g., <pre>bool := ^var1 = var2</pre>

When you define a variable at the current shell level, you define it for all levels below the current shell level. For example, suppose you define the variable `d:= 25` in the shell input pad, and then execute a shell script. Since scripts run at a lower shell level, the value assigned to variable `d` remains 25, unless the script redefines the variable by changing its value. If the script does change the value for variable `d`, the value returns to 25 when the script completes execution.

Using Shell Variables

To use a shell variable, precede the variable name with the substitution character (^). When the shell encounters the substitution character in a command line, it substitutes the value of the variable for the variable name. Variable names are *not* case-sensitive. Let's look at an example.

Suppose we assign the variable `cities` a string value:

```
cities := "Boston and NY"
```

To use the variable `cities`, simply precede it with the substitution character as follows:

```
args (( "Cities with early flights are " + ^cities ))
```

The `args` command uses standard output to display the result of the expression to the transcript pad. In this example, the shell substitutes the string value "Boston and NY" for the variable name `cities`. The expression concatenates the first string and the second string to form the following output string:

```
Cities with early flights are Boston and NY
```

The shell automatically substitutes values for (evaluates) shell variables when you use them as operands in expressions (as shown in the previous example). However, you may want to evaluate a variable that isn't part of an expression. Consider the following example:

```
args "Cities with early flights are ^cities"
```

By default, the shell won't evaluate the variable `cities` since the variable is not used in an expression. In order for the shell to evaluate variables outside of expressions, you must turn on evaluation using the `eon` command.

You can either specify the `eon` command before you run a script to turn on evaluation for the current shell, or include the `eon` command in the script itself. The `eon` command, when used in a script, turns on evaluation for the script only, not for the current shell. To turn evaluation off, use the `eoff` command.

With variable evaluation turned on, the command in the previous example evaluates the variable `cities` and displays the following string:

```
Cities with early flights are Boston and NY
```

You can also turn evaluation on when you create a shell by specifying the `-e` option with the `sh` command. By default, when you create a shell, evaluation is off.

Variable Commands

The shell provides three commands that let you verify or delete variables. Table 11-5 lists these commands.

The `existvar` and `lvar` commands verify variables defined at the current script level and every script level above. For example, when you specify the `lvar` command from within a nested script, the command lists variables defined in this script, as well as variables defined at higher levels between the current level and the shell level (one level above). When you specify `lvar` at the shell level, the command lists only variables defined at the shell level.

The `dlvar` command deletes only the variables defined at the current level. For example, suppose you defined the variable `d := 25` at the shell level. If you executed a script that used the `dlvar` command to delete the variable `d` (assuming that you didn't redefine `d` in the script), you'd receive an error. In this example, if the script redefined `d` by assigning it a new value, the command would delete the new value, and `d` would return to the value defined at the shell level.

Table 11-5. Variable Commands

Command	Description
existvar	Verifies whether the variable(s) you specify as arguments exist. If all of the variables specified exist, the command returns the value true . If any one of the variables does not exist, the command returns the value false .
lvar	Lists the type, name, and assigned value of the variable(s) you specify as arguments. If you don't specify any variables, lvar lists information about currently assigned variables.
dlvar	Deletes all variables that you specify as arguments.
export	Changes all specified variable names into environment variables. If the specified variable does not exist, export creates it.

Use the **export** command to create **environment variables** or change variables into environment variables. Environment variables store global state information about the system. We supply a set of default environment variables that you can list using the **lvar** command.

Defining Variables Interactively

So far, we've looked at variables that you either define at the shell level or from within scripts. When you define variables in a script, you assign them initial values. These initial values are used every time you execute the script, unless you edit the script to change the values prior to each execution.

Instead of including values for variables directly in scripts, you can direct the script to read values supplied by the user of the script. To read user input into variables, use the **read** command as follows:

```
read [option] variable_list
```

The *variable_list* argument specifies one or more variables that receive the input values. The example below shows a sample script that shows how to use the **read** command to read user input.

```
# stocks
#
# This script calculates the value of stock holdings.
# It reads in both the number of shares held by the user, and
# the current market price per share.
#
# Read in number of shares
#
read -prompt "Number of shares: " shares
#
# Read in current market price
#
read -prompt "Current market value: " price
#
# Calculate value of holdings and display value.
#
args (( " ^shares shares at $ ^price per share $ " + @
( ^shares * ^price ) ))
```

By default, the shell uses standard input to read values that the user of the script types in the shell input pad. Our sample script above uses two **read** commands: one reads in the number of shares and assigns the value to the variable **shares**, the other reads in the current price of each share and assigns the value to the variable **price**. Notice that each **read** command uses the **-prompt** option to prompt the user for the proper input. To see just how this script works, create your own copy and execute it.

The sample script expects the user to supply integer values. But what if the user entered a string or Boolean value? The script would use the value, and as a result, the final calculation (**^shares * ^price**) would result in an error. To keep a user from entering the wrong variable type, use the **-type** option with the **read** command as follows:

```
read -prompt "Number of shares: " -type integer @
shares
```

The `-type` option in this example directs the `read` command to accept only integer values as input. If the user specifies any other type of value, the shell will display an error and prompt the user again to enter the proper value.

Other `read` commands, like `readc` and `readln` also enable you to read user input into scripts. For more information on these commands, refer to the *Aegis Command Reference*.

Using Active Functions

You can use active functions in scripts to include string output from a command, program, or other script. When you use an active function, the system replaces it with a string containing standard output from the command, program, or script used in the function. Active functions have the following format:

```
^"command"
```

The *command* argument specifies the name of a command, program, or script whose output you want to use. You can use either single or double quotes according to the rules described in the "Using Quoted Strings" section discussed earlier. Note that output from an active function cannot exceed 1024 characters. If output does exceed this limit, the system displays an error.

You can use active functions in the same way you use variables. For example, suppose you want to use a string that shows the current date and time. (The shell command `date` displays the current date and time.) By using `date` in an active function, you can substitute the standard output string in the script as follows:

```
eon  
args "The date is ^^date' "
```

In this example, the system substitutes the standard output string from the active function 'date' to display the following line:

```
The date is Monday, May 1, 1989 10:59:28 (EDT)
```

Note that the system deletes the trailing carriage return from the output string; however, any internal carriage returns remain.

By assigning active functions to variables, you can define your own “shell functions.” For example, suppose you wrote a program called `get_process_name` that displays the current process name. To make use of this program in a script, you can refer to the program in an active function. For example:

```
eon
#
# Assign active function to variable
#
procname := ^"get_process_name"
#
# Execute DM command to make process window
# invisible
#
xdmc wi -w ^procname
#
# Go off and do something else
.
.
.
# Make process window visible again
#
xdmc wi -i ^procname
```

The script in this example assigns the active function to the variable `procname`. It uses `procname` with the DM command `wi` to make the current process window invisible and then visible again. The system substitutes the output string generated by the active function for the variable `procname`.

Controlling Script Execution

In all of the scripts we've seen in this chapter, the shell executes each command in sequence, following an unaltered path from the beginning of the script to the end as shown in Figure 11-1. Thus, these scripts perform the same basic operations each time you execute them.

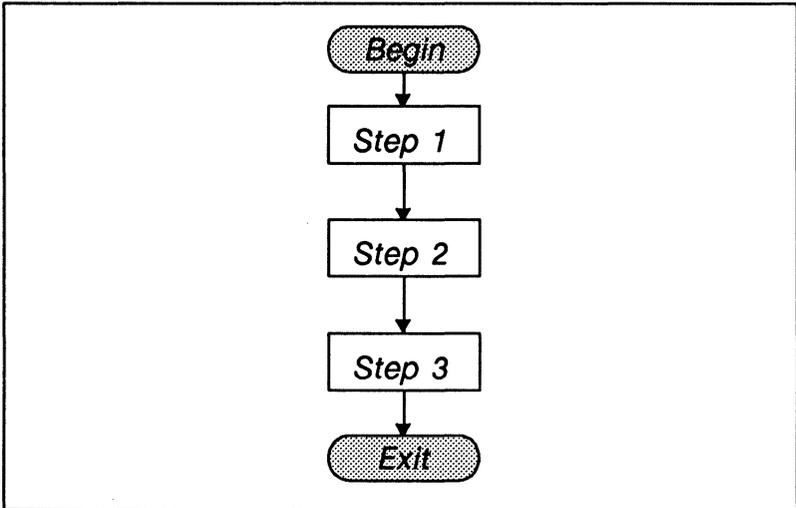


Figure 11-1. Flow of Execution in a Simple Script

You can also create scripts in which the flow of execution varies according to the results of tests performed in the script. To perform these tests in a script, you use **conditional statements**.

Conditional statements test to see if the results of a command or expression are **true** or **false**. Then, based on the result of the test, they execute a particular command or sequence of commands. Figure 11-2 shows an example of a conditional statement called an **if statement**. The if statement in Figure 11-2 controls the flow of execution by executing step 2 only if the result of the conditional statement is **true**. In this way, the script executes different commands depending on different conditions in the script.

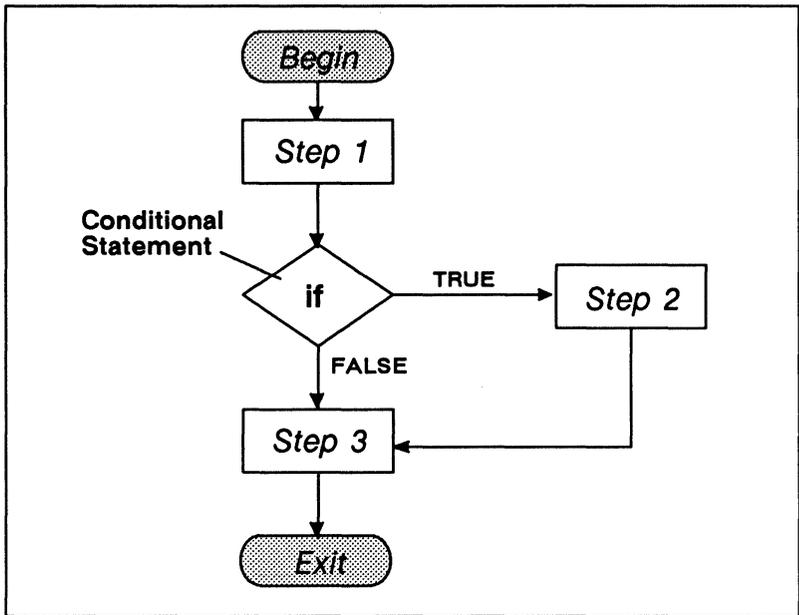


Figure 11-2. Flow of Execution with a Conditional Statement

Figure 11-2 shows a very basic example of how to use an if statement to control execution. As you'll see later in this section, you can use one or more conditional statements to create more sophisticated flow patterns in scripts.

The shell supports four different types of conditional statements:

- **if** statement
- **while** statement
- **for** statement
- **select** statement

The sections that follow describe these conditional statements and the commands that execute them.

Using the If Statement

The **if** command and all its arguments make up an **if** statement that executes one or more commands depending on the result of a Boolean test. The **if** command has the following format:

```
if com_1 then com_2 ... [ else com_3 ... ] endif
```

The *com_1* argument specifies a command, program, expression, or Boolean variable to be tested for “truth.” A test of a command or program is **true**, if the command or program executes successfully (returns an abort severity level of zero). A test of an expression or Boolean variable is **true** if they result in a **true** value.

The *com_2* argument specifies one or more commands or expressions to execute if the result of the test on *com_1* is **true**. The **endif** command signifies the end of an **if** statement. For example:

```
eon
if (( ^a < 100 ))
then args " ^a is less than 100 "
endif
```

The **if** statement in this example tests whether the value for variable **a** is less than 100. If the value for **a** is 55, then the result of the test is **true** (the expression results in a **true** value), and the **args** command executes displaying the message:

```
55 is less than 100
```

In this example, if the result of the test is **false**, the next command in the script (following **endif**) executes.

The *com_3* argument, which is optional, specifies one or more commands to execute if the test on *com_1* is **false**. For example:

```
eon
if (( ^A < 100 ))
then args " ^a is less than 100 "
else args " ^a is greater than or equal to 100"
endif
```

In this example, if the value of **a** is 900, then the test results in the value **false** (900 is not less than 100). As a result, the **args** command following the **else** statement executes displaying the message:

```
900 is greater than 100
```

When the **if** statement completes, execution of the script continues sequentially with the next command following **endif**.

Using the While Statement

The **while** command and all its arguments make up a **while statement** that executes one or more commands as long as the result of a Boolean test is **true**. The **while** command has the following format:

```
while com_1 ... do com_2 ... enddo
```

The *com_1* argument specifies a command, program, expression, or Boolean variable to be tested for “truth.” A test of a command or program is **true**, if the command or program executes successfully (returns an abort severity level of zero). A test of an expression or Boolean variable is **true** if it results in a **true** value.

The *com_2* argument specifies one or more commands or expressions to execute as long as the result of the test on *com_1* is **true**. For example:

```
i := 0
while (( i < 5 ))
do
    args (( i ))
    i := ( i ) +1
enddo
```

The **while** statement in this example tests whether the value for the variable *i* is less than 5. As long as *i* is less than 5, the **args** command displays the value of *i* and the next command adds 1 to its value. Thus, the **while** statement executes the **args** command 5 times and produces the following display:

0
1
2
3
4

On the sixth pass, the test results in a **false** value (5 is not less than 5). As a result, the script leaves the **while** “loop” and continues execution at the next command in sequence.

You can also use two special commands with the **while** statement:

- **next**
- **exit**

The **next** command returns to the top of the **while** loop. You normally use the **next** command to return prematurely to the top of the loop before executing additional commands. For example, consider the following section from a shell script:

```
while ((true))
do read -prompt "Enter number: " -type integer a
    if (( ^a < 50 )) then next endif
    args (( ^a ))
enddo
```

This **while** loop executes three commands:

- A **read** command to read in an integer value.
- An **if** command to test whether the value is less than 50.
- An **args** command to display the value.

If the integer value is greater than 50, the **if** statement is **false** and the command **args** executes. If the value is less than 50, however, the **if** statement is **true** and the **next** command executes, returning execution to the top of the **while** loop. As a result, this section of the script displays any value that is greater than or equal to 50.

The **exit** command exits the **while** loop. You normally use the **exit** command to exit a **while** loop prematurely before executing additional commands. For example:

```

while ((true))
do   read -prompt "Enter number: " -type integer a
     if (( ^a < 50 )) then exit endif
     args (( ^a ))
enddo
args "Finished"

```

The **while** loop in this example is very similar to the loop in the previous example, except that the **if** statement uses the **exit** command instead of **next**. If the integer value is greater than or equal to 50, the **if** statement is **false**, and the command (**args**) executes. If the value is less than 50, however, the **if** statement is **true**, and the **exit** command executes.

The **exit** command causes execution to exit the **while** loop and skip to the next command outside the loop (after **enddo**). As a result, this section of the script displays any value that is greater than 50, but exits the loop if you enter a value less than 50.

Using the For Statement

The **for** command and all its arguments make up a **for statement** that executes commands as long as the result of a Boolean test is true. The **for** command has two formats: one for using integer expressions, and one for using string expressions.

The **for** command used with integer expressions has this format:

```

for variable := exp_1 [ to exp_2 ] [ by exp_3 ]
    command ...
endfor

```

Here *exp_1*, *exp_2*, and *exp_3* are all expressions that result in integer values. The *exp_1* argument specifies the initial integer value assigned to *variable*.

The *command* argument specifies one or more commands to execute as long as the test on *variable* results in a **true** value. Before each iteration, the **for** statement tests to see if the current variable value is less than the value specified by *exp_2*. As long as the variable value is less than the value for *exp_2*, the result is **true**, and *command* executes.

Like the **while** statement, you can use the **for** statement to execute commands repetitively in a loop. The **for** statement is different, however, because it increments its variable automatically after each iteration. For example, the **while** and **for** statements in the following example perform the same operation:

```
#Example using while statement
#
a := 0
while (( a <= 10 )) do
    args ^a
    a := ^a + 2
enddo
#
#
#Example using the for statement
#
for a := 0 to 10 by 2
    args ^a
endfor
```

In this example, both the **while** loop and the **for** loop execute the **args** command six times. By default, the **for** statement increments the value of the variable by one after each iteration. Notice, however, that this example uses **by 2** to increment the variable by two after each iteration. Instead of the **for** loop counting from 0 to 10 by 1, it counts to 10 by 2. The result is:

```
0
2
4
6
8
10
```

The **for** command used with string expressions has this format, where *exp* specifies a string expression:

```
for variable in exp [ by [char] [word] [line] ]
    command ...
endfor
```

By default, during each iteration, **for** reads a word from the string *exp* and assigns it to *variable*. You can also direct **for** to read the string *exp* by character or line by specifying **by** with the appropriate option.

The *command* argument specifies one or more commands to execute as long as the test on *variable* results in a **true** value. Before each iteration, the **for** statement tests to see if any more characters, words, or lines exist (depending on the **by** argument specified). As long as a value exists to assign to the variable, the result is **true**, as in this example:

```
eon
for file in "tom dick harry" by word
    args "The current file is ^file"
endfor
```

In this example, with each pass through the **for** loop, **for** assigns the variable *file* a word from the string. When **for** runs out of words, it exits. As a result, the **for** statement in this example displays the following lines, and then exits:

```
The current file is tom
The current file is dick
The current file is harry
```

Using the Select Statement

The **select** command and all its arguments make up a **select statement** that executes commands according to the results of one or more Boolean tests. The **select** command has the format

```
select arg_1 [oneof|allof]
    case arg [to arg]
        commands ...
    [ case ...
        commands ... ]
    [otherwise
        commands ...]
endselect
```

The *arg_1* argument specifies the argument that **select** compares to the **case** argument, *arg*. All arguments are either integers, strings, variables, or expressions.

The shell uses each **case** statement to perform a separate Boolean test on the initial **select** argument. If the **case** argument is equal to the **select** argument, the result of the test is **true**, and the command following the **case** statement executes. Let's look at a simple example:

```
eon
select ^a allop
  case 1
    args "First case will execute if ^a = 1"
  case (( 2 + 4 ))
    args "Second case will execute if ^a = 6"
  case 6
    args "Third case will execute if ^a = 6"
endselect
```

In this example, the first case tests to see if the variable **a** equals 1, and the second and third cases test to see if **a** equals 6. The **allop** statement directs **select** to execute the commands associated with all cases that result in **true**. If **a** is 6, the **select** statement in this example executes the commands for the second and third case to display the following:

```
Second case will execute if 6 = 6
Third case will execute if 6 = 6
```

If you specify **oneof** (the default), **select** executes only the first case that results in a **true** value. In the previous example, where **a** equals 6, **select** executes only the second case to display the following:

```
Second case will execute if 6 = 6
```

You can also use the **next** and **exit** commands to control execution within the **select** statement. For example, when using **oneof** you can use the **next** statement to direct **select** to execute another case as shown in the following example:

```

eon
select ^a oneof
  case 1
    args "First case will execute if ^a = 1"
    next
  case (( 2 + 4 ))
    args "Second case will execute if ^a = 6"
    next
  case 6
    args "Third case will execute if ^a = 6"
endselect

```

In this example, if variable **a** equals 6, the second **case** executes. Although this script uses **oneof**, the **next** command following the second case directs **select** to execute the next case that's **true**. Since the third case is **true**, the script in this example executes the third case.

Using the **to** statement, you can specify a range for a case argument. The case in the following script tests for a value in the range of 1 to 10:

```

eon
select ^a allof
  case 1 to 10
    args "Variable a is the number ^a"
endselect

```

You can also use the **to** statement to test for a range of string characters. For example:

```

eon
select ^a allof
  case a to z
    args "Variable a is the letter ^a"
endselect

```

The case in this example tests for a string value between **a** and **z**. Note that this range is case-sensitive, so the case is **true** for example, if **a** equals **r** but not **R**.

Use the **otherwise** statement when you want to perform an operation if the test on a case is **false**. For example:

```
eon
select ^a
    case 0 to 10
    args "Value for a is a number from 0 to 10"
    otherwise
    args "Value for a is greater than 10"
endselect
```

In this example, if the value for **a** is a number between 0 and 10, the case is **true**. As a result, **select** displays the following:

```
Value for a is a number from 0 to 10
```

If the value is a number greater than 10, the case is **false**, and the command following **otherwise** executes, displaying this:

```
Value for a is greater than 10
```

If you include several cases on the same line, **select** separates each case with an implied **or** operator (see the “Logical Operators” section discussed earlier). You can also use the **@** character to escape newline characters and continue an “ored” case on more than one line. For example:

```
eon
select ^a
    case 1 case 3 case 5
    args "Variable a matches 1, 3, or 5"
    case 2 @
    case 4 @
    case 6
    args "Variable a matches 2, 4, or 6"
endselect
```



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*)

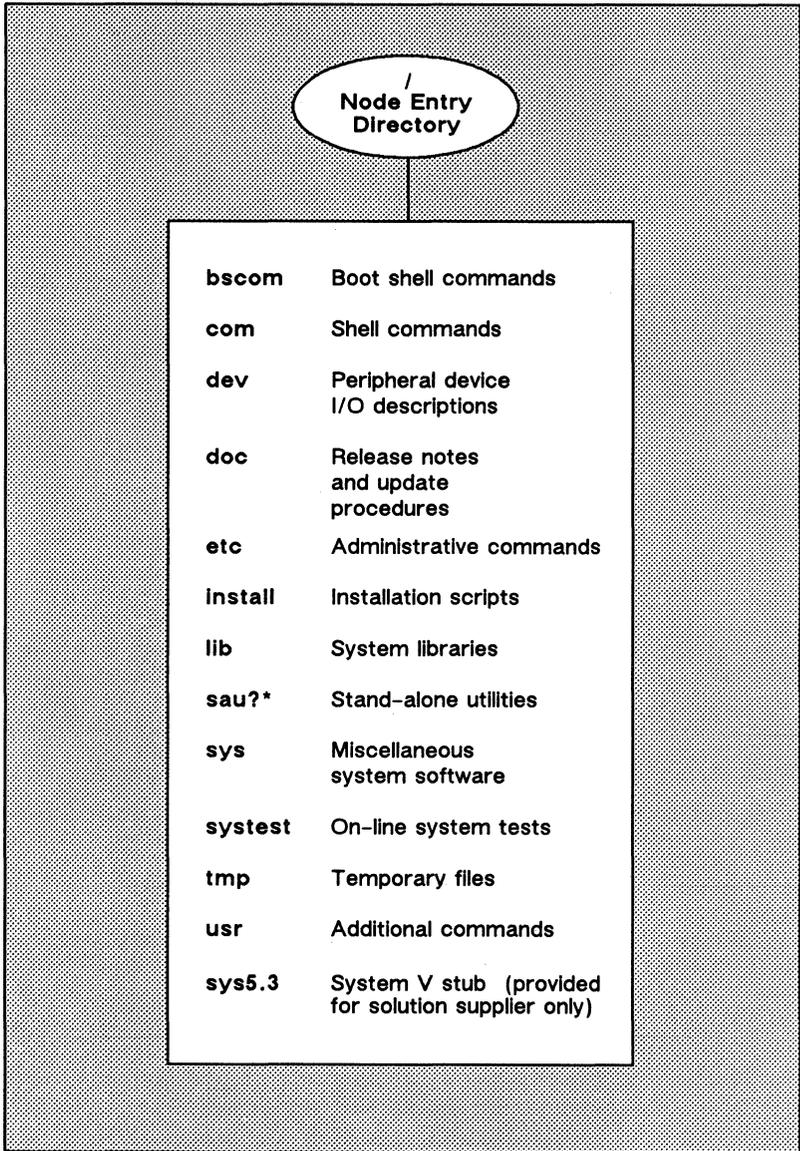


Figure A-1. The Node Entry Directory (/) and Subdirectories

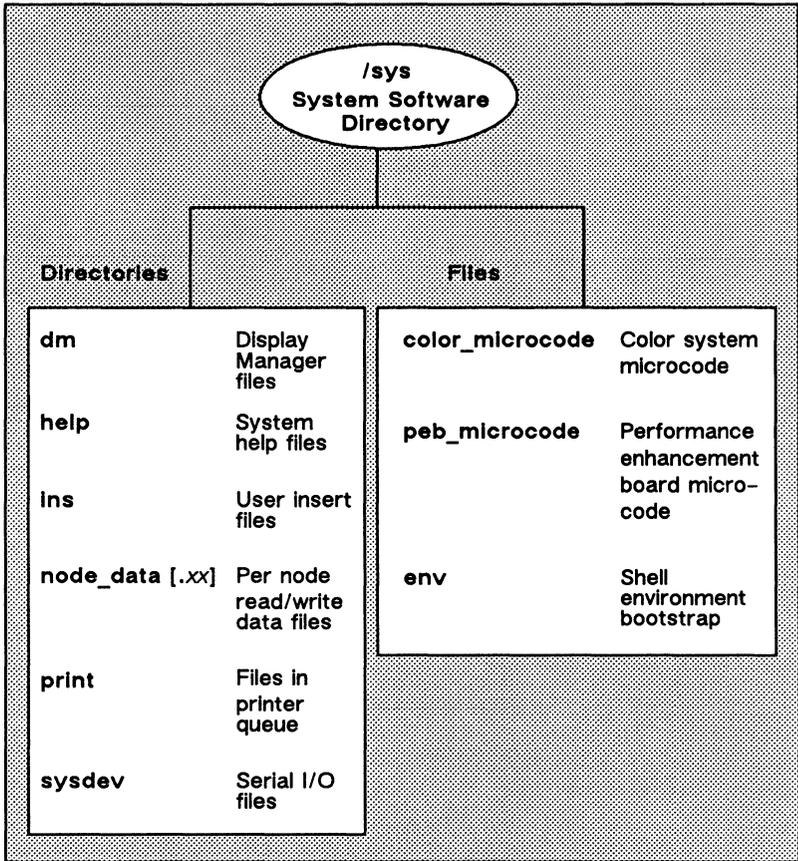


Figure A-2. The System Software Directory (/sys)

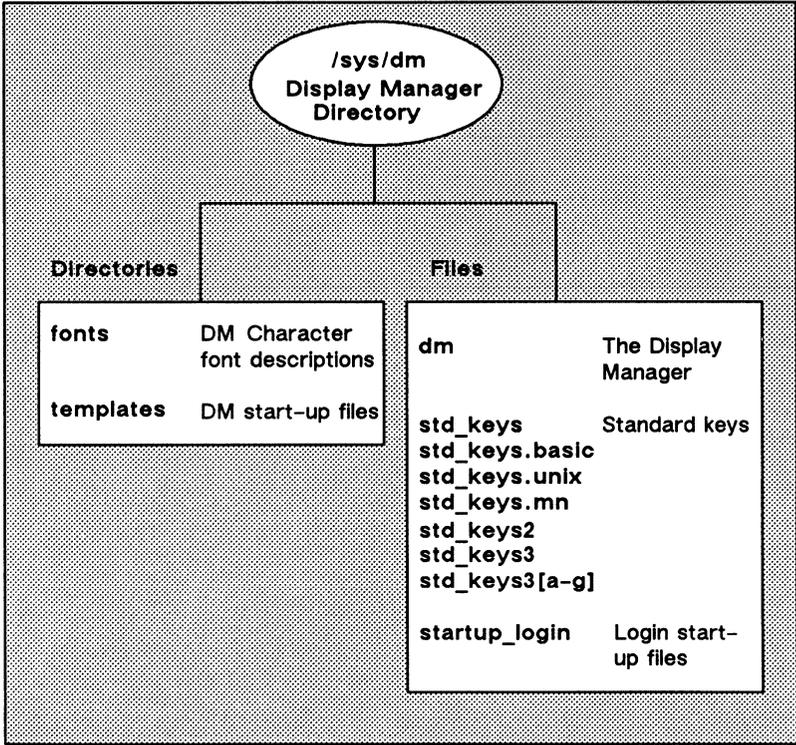


Figure A-3. The Display Manager Directory (/sys/dm)

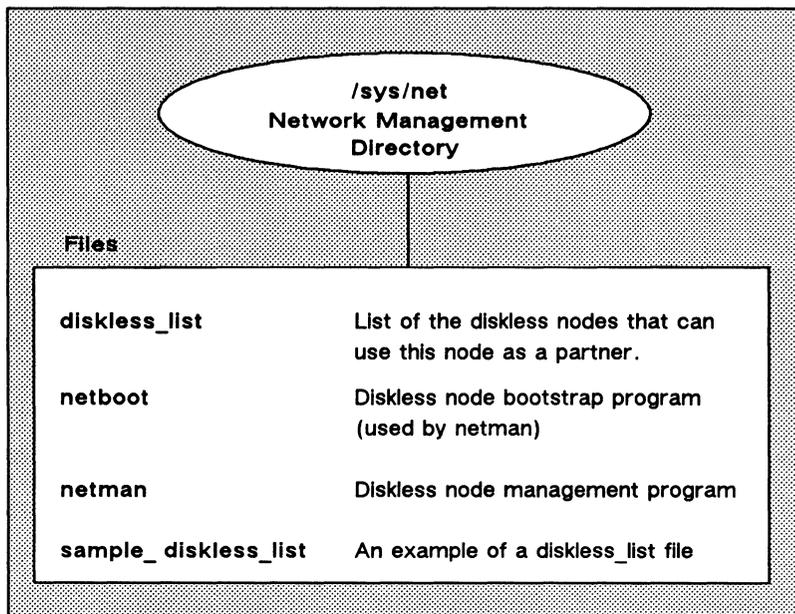


Figure A-4. The Network Management Directory (/sys/net)

Appendix B

Summary of Predefined Standard Key Definitions

This appendix summarizes the predefined standard key definitions read during DM startup for all keyboards. These are found in the file `/sys/dm/std_keys.basic`. This appendix also includes special operating considerations for the Multinational keyboard. 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	<TAB>
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 previous window	tlw	CTRL/L
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	<i>cp pathname</i>	None
Create new process without pads or windows	<i>cpo pathname</i>	None
Create a server process	<i>cps pathname</i>	None

Table B-3. Controlling Processes

Task	DM Command	Predefined Key
Quit, stop, or blast process	<i>dq [-b -s -c nn]</i>	CTRL/Q
Suspend execution of process	<i>ds</i>	None
Resume execution of a suspended process	<i>dc</i>	None

Table B-4. Creating Pads and Windows

Task	DM Command	Predefined Key
Create an edit pad and window	<i>ce pathname</i>	<EDIT> (R4)
Create a read-only window	<i>cv pathname</i>	<READ> (R3)
Create a copy of an existing pad and window	<i>cc</i>	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
Changing window size	wg	CTRL/G
Changing 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 autohold mode	wa; ws	CTRL/A
Set hold mode	wh [-on -off]	<HOLD> (R6)

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 <i>n</i> pages	pp [-]<i>n</i>	  (LD, LF)
Move pad <i>n</i> lines	pv [-]<i>n</i>	SHIFT/  (L8S) SHIFT/  (LES)
Move pad <i>n</i> characters	ph [-]<i>n</i>	  (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 <i>grp_name</i> [<i>entry_name</i>]	None
Remove a window from window group	wgrr <i>grp_name</i> [<i>entry_name</i>]	None
Make windows invisible	wi <i>entry_name</i>	None
Change windows to icons	icon [<i>entry_name</i>] [<i>character</i>]	SHIFT/<POP>
Set icon positioning and offset	idf	None
Display list of windows in group	cpb <i>group_name</i>	None

Table B-9. Setting Edit Modes

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

Table B-10. Inserting Characters

Task	DM Command	Predefined Key
Insert string at cursor	es ' <i>string</i> '	Default DM operation
Insert newline character	en	<RETURN>
Insert a new line after current line	tr; en; tl	<F1>
Insert end-of-file mark	eef	CTRL/Z

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> (BS)
Delete "word" of text	dr;/[-A=Z-9\$_]/xd	<F6>
Delete from cursor to end of line	es '' ;ee;dr;tr xd;tl;tr	<F7> (L3A)
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 pathname] [-r]	<COPY> (L1A)
Cut (delete) text and write it to a paste buffer or file	xd [name -f pathname] [-r]	<CUT> (L1AS)
Paste (write) text from a paste buffer or file into a pad	xp [name -f pathname] [-r]	<PASTE> (L2A)

Table B-13. Commands for Searching for Text

Task	DM Command	Predefined Key
Search forward for string	<i>/string/</i>	None
Search backward for string	<i>\string\</i>	None
Repeat last forward search	<i>//</i>	CTRL/R
Repeat last backward search	<i>\\</i>	CTRL/U
Cancel search or any action involving the <i>echo</i> 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 <i>string2</i> for all occurrences of <i>string1</i> in a defined range	s/string1/string2/	None
Substitute <i>string2</i> for the first occurrence of <i>string1</i> 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.

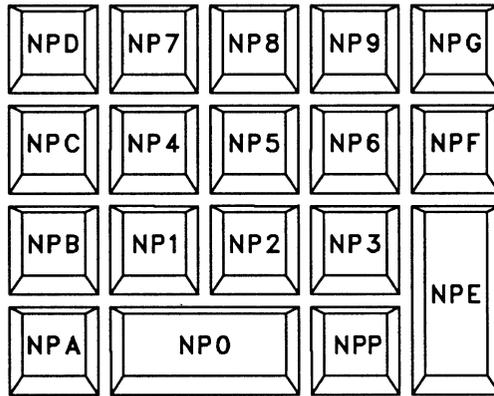


Figure B-1. Multinational Keyboard Numeric Keypad

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 3-3. 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 DM on such a node.

Appendix C

Sample Shell Scripts

This appendix contains sample shell scripts to supplement those provided in Chapter 11 of this manual.

Script 1: Prompting For and Checking a Target Node Name

The following script prompts for and checks the target node name. Include this script in others scripts (via the `source` command) that need to query the user for a node name.

```
#!/com/sh
# Prompt for and check target node name
# Usage: ask_target [node_name]
#
# Return node name as "^node"
eon
if not eqs ^1 "" then
    args ^1 | tlc [A-Z] [a-z] | read -type string node
else
    node := " "
endif
valid_node := false
while ((not ^valid_node)) do
    select ^node oneof
        case " "
            # Prompt for node
            args ""
            readln -p "Target node (//node or <<RETURN> for this node)? " node
            if ((^node = "")) then
                lusr -me | chpat {?*}{/*?}* @2 | read -type string node
            endif
        otherwise
            # Test node name
            if existf ^node then
                valid_node := true
            else
                args "" " ^node is unavailable. Check the name and try again later."
                return -M
            endif
        endselect
    enddo
```

Script 2: Generic Routine Prompting for a Yes or No Answer

The following script provides for generic routine prompting for a “yes” or “no” answer. Include this script in other scripts (via the source command) that need to query the user for a Boolean response.

```
#!/com/sh
# Generic routine prompting for YES or NO answer.
# Usage: ask_yn "prompt_string"
#
# Return user response as "^response": value 'y' or 'n'
eon
valid_answer := false
read -p "^*" "answer"
args "^answer" | tlc A-Z a-z | read answer
while ((not ^valid_answer)) do
  select ^answer oneof
  case 'y' case 'yes'
    response := 'y'
    valid_answer := true
  case 'n' case 'no'
    response := 'n'
    valid_answer := true
  otherwise
    args "" " Please answer 'yes' or 'no'."
    read -p "^*" "answer"
    args "^answer" | tlc A-Z a-z | read answer
  endselect
enddo
```

Script 3: Disk Cleanup Utility

The following script deletes files with a `.bak` extension, as well as Interleaf desktop files with a `_1` (Interleaf backup files) extension. It asks you to confirm each deletion. Typing “g” at the prompt lets you do all files with no further prompting. Note that `tip` and `joe` are only supplied in the script as examples; to use the script, you must specify your user id and the node on which the disk cleanup is to take place.

```

#!/com/sh
eon
# Don't let it abort if objects are in use, etc
abtsev -P
args "
args "***** Cleanup Utility *****"
args "Starting cleanup..."
args ""
args 'Your current disk space is...'
lvofls
lvofls | readln blank header initial_space
args ""
args "First we'll look for files in your Interleaf clipboard. Generally, files are placed"
args 'there for deletion, but one may forget to open the clipboard to delete them.'
args ""
args 'Looking for Interleaf clipboard files...'
args 'At the prompt, type "g" if you do not want further prompting'
args ""
dlf ~/desktop/clipboard.clp/?* -a -l
lvofls | readln blank header after_clipboardargs ""
args "Now we'll look for files with a _n extension. These are the backup files that"
args 'Interleaf makes automatically.'
args ""
args 'Looking for Interleaf backup files...'
args 'At the prompt, type "g" if you do not want further prompting'
args ""
dlf ~/desktop/.../?*_1 -a -l
lvofls | readln blank header after_interleaf
args ""
args "Now we'll look for .bak files."
args "It can take a few seconds to find them..."
args 'At the prompt, type "g" if you do not want further prompting'
args ""
dlf ~/.../?*.bak -a -l
lvofls | readln blank header after_dbak
args ""
args "***** Cleanup Results *****"
args ""
args "          ^header"
args ""
args " Initial space..... ^initial_space"
args " Cleaned out clipboard..... ^after_clipboard"
args " Removed Interleaf backups... ^after_interleaf"
args " Removed .bak files..... ^after_dbak"
args " ====="
args " Final Results..... ^after_dbak"
args ""
date | readln when
args " ^user ^when" >> //tip/joe/user_data/usage_cleanup
args " Initial space..... ^initial_space" >> //tip/joe/user_data/usage_cleanup
args " Cleaned out clipboard... ^after_clipboard" >> //tip/joe/user_data/usage_cleanup
args " Removed Interleaf backups. ^after_interleaf" >> //tip/joe/user_data/usage_cleanup
args " Removed .bak files... ^after_dbak" >> //tip/joe/user_data/usage_cleanup
args " =====" >> //tip/joe/user_data/usage_cleanup
args " Final Results..... ^after_dbak" >> //tip/joe/user_data/usage_cleanup

```

Script 4: Printing a Directory's Most Recent Backup Activity

This script lists the last time each directory in a designated list of directories was backed up. It uses the **backup_history** file in each directory to report the last full backup and each incremental backup since that time.

If an argument is given, it is first assumed to be a wildcard matching one or more directories containing a file called **backup_list**. If at least one match is found, the script opens and parses each matched **backup_list** file; for each entry in each list, it finds a **backup_history** file and prints its report.

If a **backup_error_history** file exists, the last entry in it is also printed. If no matching **backup_list** is found, the script tries again, assuming the argument is a wildcard matching one or more directories directly containing **backup_history** files. If any of these are found, it prints its report for each one. If neither sort of match is found, the script complains and exits.

If no argument is given, this node's entry directory ("/") is used in the above matching process. Searching a long **backup_history** file is time consuming, so you will do well to remove any unnecessary information from the file.

```

#!/com/sh
eon
# Set the abort severity high to handle acl or other unusual problems
abtsev -o
warn := false
found_match := false
# Generate the pattern for the backup list(s)

backup_pattern := "^1" + "/backup_list"
# Loop on wildcard matches for the backup list
ld ^backup_pattern -c -nwarn >?/dev/null | @
while readln backup_list
do
    found_match := true
    args "" "For ^backup_list:"

# Type out last entry in error history file, if any
    backup_error_history := ^backup_list - "list" + "error_history"
    if existf ^backup_error_history
    then
        args "" "Last entry in ^backup_error_history:"
# Put the last error entry in a temp file
        while readln error_line
        do
            if (( (^error_line - "? (wbak)") <> ^error_line ))
            then
                args "" ^error_line >/tmp/backup_temp
            else
                args "" ^error_line >>/tmp/backup_temp
            endif
        enddo <^backup_error_history # error history file line loop
# Run catf on the temp file to print last error, then delete it
        catf /tmp/backup_temp
        dlf /tmp/backup_temp -f >?/dev/null
    endif # backup error history exists
# Parse through the backup list file and check each entry:
# Skip comment lines, blank lines, and the backup list itself
fpat <^backup_list -x -i "% *#" "% *$" "backup_list$" | @
while readln list_entry
do
# Generate the full pathname for the backup_history file
    backup_history := ^list_entry + "/backup_history"
# Print out the latest backup times for the entry if it exists
    if existf ^backup_history
    then
        args "" "Backups of ^list_entry since last full backup:"
# Put the last full backup line and all incrementals since then in a temp file
        args "--No full backup found.--" >/tmp/backup_temp
        fpat <^backup_history "full" "incremental" | @
        while readln backup_line
        do
            if (( (^backup_line - "full") <> ^backup_line ))
            then
                args "" ^backup_line >/tmp/backup_temp
            else
                args "" ^backup_line >>/tmp/backup_temp
            endif
        enddo
    endif
enddo

```

```

enddo          # backup history line loop

# Run catf on the temp file to print times, then delete it
catf /tmp/backup_temp
dlf /tmp/backup_temp -f >?/dev/null
else
# No backup history found, issue warning
args "" ""?(^0) ^backup_history not found."
warn := true
endif          # backup history exists
enddo          # list_entry loop
enddo          # wildcard loop
# If backup_list wildcard failed, try backup_history
if ((not ^found_match))
then
# Generate the pattern for the backup history(s)
backup_pattern := ""^1" + "/backup_history"
# Loop on wildcard matches for the backup history
ld ^backup_pattern -c -nwarn >?/dev/null | @
while readln backup_history
do
    found_match := true
    list_entry := ^backup_history - "/backup_history"
    args "" ""Backups of ^list_entry since last full backup:"
# Put the last full backup line and all incrementals since then in a temp file
args "--No full backup found.--" >/tmp/backup_temp
fpat <^backup_history "full" "incremental" | @
while readln backup_line
do
    if (( (^backup_line - "full") <> ^backup_line ))
    then
        args "^backup_line" >/tmp/backup_temp
    else
        args "^backup_line" >>/tmp/backup_temp
    endif
endif
enddo          # backup history line loop
# Run catf on the temp file to print times, then delete it
catf /tmp/backup_temp
dlf /tmp/backup_temp -f >?/dev/null
enddo          # wildcard loop
endif
# If still no matches for backup pattern, return error
if ((not ^found_match))
then
    args "" ""?(^0) ^1/backup_list not found." ""?(^0) ^1/backup_history not
found."
    return -e
endif          # no matches
# Return warning severity if some backup history was not found
if ((^warn)) then return -w endif

```

Script 5: Resolving Links to Find an Ultimate Pathname

The following script progressively resolves links, displays the link text, and finishes by showing the true pathname. Note that it does not handle wildcards. When you invoke this script, you need to supply, as an argument, the link name that you want resolved.

```
#!/com/sh
link := true
print := true
target := "^1"
cwd := ^wd"
eon
if eqs ^1 '' then
  args "Use: ^0 pathname"
  return -err
endif
while ((^link)) do
  if (args ^target | fpat '%[a-zA-Z$._] ' > /dev/null) then
    args "cwd/^target" | read target endif
  if existf ^target then
    if (ld ^target -en -ll -c | fpat '?*' >> /dev/null) then
      link := true
      ld ^target -en -ll -lt -c | read link_ob link_tx
      link_tx := ((^link_tx - @ - @))
      args "target is a link to ^link_tx"
    # Find out if a given link is '.' relative, if so, prepend pathname
    if (args ^link_tx | fpat '%[a-zA-Z$._] ' > /dev/null)
  then
    args ^target | chpat '%{?*/}{?*$}' "@1/" | read target
    link_tx := ((^target + ^link_tx))
    endif
    target := ^link_tx
    else link := false
    endif
  else
    link := false
    print := false
    args "target doesn't exist"
  endif
enddo
if ((^print)) then args "target is the real thing" endif
```


Appendix D

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 2 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 includes 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.

For more information on printing Latin-1 characters, see *Printing in the Aegis Environment*.

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 system Software Release Notes.

Character Compose Sequences

The following chart shows what two characters you must type to compose each individual Latin-1 character.

Table D-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
-^	ˆ	Macron
0^	°	Ring above, degree sign
+-	±	Plus-minus sign

2^	²	Superscript two
3^	³	Superscript three
”	‘	Acute accent
/u	μ	Micro sign
¶	¶	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	Capital 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'	Û	Capital letter U with 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 with 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.)

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.)

Command procedure

See Shell procedure.

Command search path

The route that a shell takes in searching through various directories for command files. A default search path exists for the Aegis shell. You may add other directories of executable files which a shell then looks through on its way to finding a particular command name.

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.

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.

Group Identification Number (GID)

A unique number assigned to one or more logins that is used to identify groups of related users.

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 procedure is called. (*See also* Shell procedure.)

Link

A special type of object that points from one place in the naming tree to another.

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.

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 directory

Each process uses a naming directory. Like the working directory, the naming directory points to a certain destination directory. The system uses your home directory as the initial naming directory.

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 Aegis system 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.

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.)

Shell

A command-line interpreter program used to invoke operating system utility programs.

Shell command

An instruction you give the system to execute a utility program. (*See also* Script.)

Shell metacharacter

Any character that has special meaning to a shell. Asterisks, question marks, and ampersands are a few examples.

Shell procedure

An executable file that is not a compiled program. It is a call to a shell to read and execute commands contained in a file. A sequence of commands may thus be preserved for repeated use by saving it in a file which can also be called a command procedure.

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.

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.

Start-up script

A file that sets up the initial operating environment on your node. This file is also known as a “boot script”. (*See also* Script.)

System administrator

The person responsible for system maintenance at your site.

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.

Index

Symbols are listed at the beginning of the index. Entries in color indicate task-oriented information.

Symbols

- ... (ellipsis), 6-18, 8-10
- ! (exclamation point), 11-7
- ? (question mark), 6-17
- ; (semicolon), 11-2
- " (double quotation marks), 11-7 to 11-8
- ' (tick character), 2-6, 2-14
- ' (single quote), 11-6, 11-7 to 11-8
- () (parentheses), 6-19, 11-14
- [] (brackets), 5-18, 6-18
-] (right bracket), 5-18
- { } (braces), 6-19
- & (ampersand), 6-20
- # (pound sign), 11-2
- \$ (dollar sign), 5-16, 11-18
- % (percent), 6-17, 10-3
- @ (at sign), 5-18, 5-19, 6-4, 11-2
- + (plus), 11-14
- (hyphen), 5-18
- (minus), 11-15
- * (asterisk), 5-17, 6-17, 6-18
- / (slash), 1-7, 5-21, 11-9
- // (double slashes), 1-7
- ^ (caret), 11-5, 11-19
- | (vertical bar), 6-12
- = (equal), 6-19
- > (left angle bracket), 6-10
- > (right angle bracket), 6-11
- >? (right angle bracket/question mark), 6-11
- >> (double right angle brackets), 6-11, 7-10
- >>? (double right angle brackets/question mark), 6-11
- (tilde), 1-10, 5-18
- _ (underscore), 11-18

A

- aa** (acknowledge alarm) command, 4-25
- ABORT** key, 4-16
- abrt** (abort) command, 4-18, 5-22

absolute pathname, 1-7

access rights, 10-4

ACL (access control list), 10-1, 10-25

- acl** (access control list) command, 10-7, 10-21
- adding entry rights to, 10-15
- changing, 7-6, 10-14
- copying, 10-17
- deleting entries from, 10-16
- deleting entry rights from, 10-16
- displaying, 10-7
- editing, 10-8
- entries
 - extended, 10-4
 - required, 10-4
- initial, 10-18
- rules for specifying entries, 10-10
- setting entries, 10-14
- structure of, 10-2

acm (access control list manager) call, 10-24

active functions, 11-23

Aegis shell, 1-3

alarms, DM, 4-25

ap (alarm pop) command, 4-25

appending output, 6-11

args (arguments) command, 11-4, 11-19

arguments, 6-3, 11-4

ASCII

- characters, 5-16
- files, comparing, 7-18

assignment character (:=), 11-17

B

background processes, 6-20

backslash character (\), 5-21

BACKSPACE key, 5-7

.bak files, 5-27

boff (background off) command, 6-20

bon (background on) command, 6-20

Boolean values, 11-18

- in expressions, 11-12

booting, 2-2

boot volume, 1-3

C

case command, 5-26

case comparisons, 5-23

case statements, 11-33

catf (catenate file) command, 6-11, 7-10

cc (create copy) command, 4-10

ce (create edit pad) command, 4-10, 7-5

changing passwords, 2-21

character class, 5-17

Character Compose Sequences, D-4

CHAR DEL key, 5-7

chn (change name) command, 7-6, 8-2, 9-4

chpat (change pattern) command, 5-15

class names, 10-11, 10-13

cmdf (command file) command, 2-20

cmf (compare file) command, 7-18

cmt (compare tree) command, 8-8

command

- arguments, 6-3
- creating your own, 11-1
- format, 6-2
- mode, for editing ACLs, 10-9
- options, 6-3, 6-4
- parser, 6-13
- search rules, 6-5, 11-3
 - default, 6-5
- shell, 6-1

comparison operators, 11-16

compose function, 3-12, D-1

conditional statements, 11-25

control key sequences, 3-9

controlling I/O, 6-8

COPY key, 5-13

copying

- display images, 5-13
- text, 5-10, 5-11

cp (create process) command, 4-6

cpb (create paste buffer) command, 4-21, 4-36

cpf (copy file) command, 7-7

cpl (copy link) command, 9-5

cpo (create process only) command, 4-7

cps (create process server) command, 4-7

cpscr (copy screen) command, 7-17

cpt (copy tree) command, 8-4

crd (create directory) command, 8-2

crf (create link) command, 9-2, 9-3

crsubs (create subsystem) command, 10-24 to 10-25

csr (command search rules) command, 6-6

ctnode, catalog node, 1-5

cursor control, 4-2 to 4-4

CUT key, 5-14

cutting text, 5-10, 5-13

cv (create view) command, 3-9, 4-10

D

daemons, 2-5, 2-12

data objects, 10-22

date (display date) command, 6-2, 11-23

debugging shell scripts, 11-10

default

- file ACLs, 7-8
- paste buffer, 5-11

defining

- keys, 3-15
- points and regions, 3-5
- ranges of text, 5-8

deleting

- characters, 5-7
- lines, 5-8
- text, 5-6
- words, 5-7

directories

- commands for managing, 8-1

- comparing, 8-8
- copying, 8-3
- creating, 8-2
- deleting, 8-10
- displaying contents, 8-9
- home, 1-9
- merging, 8-7
- naming, 1-10
- network root, 1-4
- node entry, 1-4
- parent, 1-12
- reñaming, 8-2
- replacing, 8-5
- working, 1-9

directory trees, 8-3

diskless node, 1-3, 2-4, 2-12

Display Manager (DM), 1-3, 2-5, 2-13, 3-1

- alarms, 4-25
- command scripts, 3-20
- commands
 - format of, 3-3
 - invoking interactively, 3-2
 - special characters, 3-4
 - start-up script, 2-20

dldupl (delete duplicate lines) command, 6-12

dlf (delete file) command, 7-16

dll (delete link) command, 9-6

dlt (delete tree) command, 8-10

Domain Server Processor (DSP), 2-23

dlvar (delete variable) command, 11-20

dr (define region) command, 4-15, 5-8

DSP (Domain Server Processor), 2-23

E

echo (text echo) command, 5-9

ed (edit) command, 5-7, 5-15

edacl (edit ACL) command, 7-6, 8-2, 10-7, 10-8, 10-20

edfont (edit font) command, 4-34 to 4-35

EDIT key, 4-12, 7-5

edit modes, 5-2

edit pad and window, creating, 4-12

edit pad modes, 5-2

edstr (edit stream) command, 5-15

eef (edit end-of-file) command, 5-6

ei (edit insert) command, 4-23, 5-4

en (edit newline) command, 5-5

ensubs (enter subsystem) command, 10-25 to 10-26

environment variables, 11-21

EOF (end-of-file) mark, 5-6

eoff (execution tracing off) command, 11-20

eon (execution tracing on) command, 11-20

error input, 6-9

error output, 6-9

es (edit string) command, 5-5

escape character (@), 5-18, 6-4

/etc/daemons, 2-13

/etc/daemons/llbd, 2-13

/etc/dm_or_spm, 2-13

/etc/envron, 2-4, 2-12
/etc/init, 2-4, 2-12
/etc/rc, 2-4, 2-12
/etc/sys.conf, 2-12
/etc/ttys, 2-5, 2-13
European Characters
 creating and displaying, D-1
 defining, 3-12
existvar (exist variable) command, 11-20
EXIT key, 4-6, 4-16
export (export variable) command, 11-21
expression delimiters, 11-11
expression operators, 11-13
expressions, 11-11 to 11-12, 11-17
 operands in, 11-12
extended ACL entries, 10-4

F

F6 function key, 5-7
F7 function key, 5-8
for statements, 11-30
files
 appending, 7-10
 commands for managing, 7-4
 comparing ASCII, 7-18
 copying, 7-7
 copying displays to, 7-17
 creating, 7-5
 deleting, 7-16
 displaying attributes of, 7-15
 moving, 7-9
 printing, 7-11
 renaming, 7-6

filters, 6-12
fmt (format file) command, 6-11
fp (find pattern) command, 5-15
fp (find pattern block) command, 5-15

G

getty, 2-5, 2-13
glbd server (NCS), 2-5, 2-12
GMF, 5-13
GROW key, 4-18

H

here documents, 11-9, 11-17
HOLD key, 4-24
home directory, 1-9
 changing, 2-22
horizontal offset, in windows, 5-2

I

icon command, 4-34
icons, 4-30, 4-32 to 4-33
idf (icon default) command, 4-35
if statements, 11-25, 11-27
in-line data, in shell scripts, 11-8
init process, 2-4
initial ACLs, 7-6, 8-2
 copying, 10-21
 editing, 10-20
 for new directories, 10-18
 for new files, 10-18
initial shell environment, 6-7

inlib (initialize library) command, 6-5

insert mode, 4-22, 5-3

inserting

- EOF marks, 5-6
- new lines following the current line, 5-5
- newline characters, 5-5
- raw (`noecho`) characters, 5-4
- text strings, 5-5

INS key, 5-4

integers, 11-18

interactive mode, for editing
ACLs, 10-9

I/O control characters, 6-7, 6-9

K

key definitions, 2-17

- deleting, 3-18
- displaying, 3-19
- examples, 3-15

key naming, 3-13 to 3-14

keyboards,

- defining, 3-10, 3-15 to 3-16
- definition files, 3-12
- low-profile
 - international, B-8
 - key definitions, 3-10
 - Model I, 3-10
 - Model II, 3-10
- Multinational, 3-10, 3-12
- types, 3-10

L

Latin-1 character set, creating
and displaying, 3-12

ld (list directory) command, 6-3,
7-15, 8-9, 9-3

line numbers, in window legends,
5-2

link resolution names, 9-3

links

- copying, 9-5
- creating, 9-2
- definition of, 9-1
- deleting, 9-6
- displaying, 9-3
- redefining, 9-3
- renaming, 9-4
- replacing existing, 9-5

llbd server (NCS), 2-5, 2-12

log-in shell, 2-19 to 2-20

log-in start-up script, 2-17 to
2-18

logging in, 2-17, 2-21

logical operators, 11-16

lvar (list variable) command,
11-20

M

MARK key, 4-15, 4-18, 5-8,
5-9

mathematical operators, in expres-
sions, 11-14

mbx_helper, 4-8

Mnemonic Debugger (MD), 2-4,
2-11

mouse keys, 3-7, 3-10

MOVE key, 4-19

Multinational Keyboards, D-1
key definitions, 3-10
operating considerations, B-8

multiple pathnames, 6-13
mvf (move file) command, 7-9

N

names file, 6-16
naming directory, 1-10
 setting, 7-2, 7-3
naming server helper (**ns_helper**),
 1-5
naming tree, 1-4, 7-2
nd (naming directory) command,
 7-3
netboot program, 2-11
netman program, 2-4, 2-11,
 2-12, 2-14
network, 1-2
network partner, 1-3
network root directory, 1-4
node, 1-1
 cataloging, 1-5
 diskless, 1-3, 2-4, 2-8, 2-12
node entry directory, 1-4
ns_helper (naming server helper),
 1-5

O

offset specification, 4-35
operators
 comparison, 11-16
 logical, 11-16
 mathematical, 11-14
 string, 11-14
options, command, 6-3, 6-4
otherwise statements, 11-35

overstrike mode, 5-3

P

pads
 closing, 4-15
 copying, 4-14 to 4-15
 creating, 4-9
 deleting, 4-16
 moving under windows, 4-26
 scrolling horizontally, 4-29
 scrolling vertically, 4-27 to
 4-28
parent directory, 1-12
parsing operators, 6-7, 6-20,
 11-2
password, changing, 2-21
paste buffers, 5-8, 5-10
PASTE key, 5-8, 5-15
pasting text, 5-10, 5-14
pathname, absolute, 1-7
pathname wildcards, 6-7, 6-13,
 6-17
pathnames, using, 1-6
pb (pad bottom) command, 4-27
percent sign (%), 5-16
pipe, 6-12
pn (pad name) command, 4-29
point pairs, 4-10
points, defining, 3-5
POP key, 4-21
pp (pad page) command, 4-27
prf (print file) command, 5-13,
 7-11
print menu interface, 7-13
print server, 2-4, 7-11

- printers, 2-4
- process window
 - legend, 4-22
 - modes, changing, 4-21
- processes
 - controlling, 4-8
 - creating, 4-4
 - stopping, 4-9
 - suspending/resuming, 4-9
- programs, stopping, 4-9
- PROM, 2-4, 2-11
- protected subsystems, 10-2, 10-22, 10-24, 10-25
 - controlling access to, 10-23
 - managers of, 10-22
- prsvr (print server) command, 2-4, 7-11
- pt (pad top) command, 4-27
- pv (pad line) command, 4-28
- pw (pad write) command, 4-16, 5-3, 5-27

Q

- query options, 6-14
- question mark (?), 5-17
- queuing a file for printing, 7-12
- quoted strings, 11-7

R

- READ key, 4-14
- required ACL entries, 10-4
- ranges of letters or digits, 5-17
- rc scripts, 2-4, 2-12

- read** (read user input into variables) command, 11-21 to 11-22
- read-only mode, 5-3
- read-only pad and window, creating, 4-13
- reading
 - from standard input, 6-15
 - of input from a file, 6-10
 - of input into shell scripts, 11-23
 - of input into variables, 11-22
- redirecting output, 6-10, 6-11, 6-12, 7-10
- regions, defining, 3-2, 3-5, 4-15, 4-25
- regular expressions, 5-15
- responding to queries, 6-15
- ro** (read-only) command, 5-3

S

- s** (substitute) command, 5-20
- SAVE key, 5-28
- sc** (set case) command, 5-16
- script execution, controlling, 11-25
- search operations
 - canceling, 5-23
 - repeating, 5-22
- searching for text, 5-20
- select statements, 11-32
- server processes, 2-5, 2-12
- Server Process Manager (SPM), 2-5, 2-13
- server programs, 4-4

sh (create Aegis shell) command, 6-7

shell

- definition of, 1-3
- defining functions for, 11-24
- entering commands in, 6-1
- input pad, 6-1
- log-in, 6-7
- log-in script, 2-18
- process, 6-1
- scripts, 6-2, 11-1, C-2-C-9
 - controlling execution, 11-25
 - creating, 11-2
 - debugging, 11-10
 - executing DM commands from, 11-9
 - using variables in, 11-17
 - verification options for, 11-10
- start-up files for, 6-7 to 6-8
- stopping, 4-6
- variables, 11-19

SHIFT key, 4-28

- rules for specifying, 10-10

SID (subject identifier), 4-4, 10-2

single integers, in expressions, 11-12

siologin (log-in on an sio line) command, 6-8

so (substitute once) command, 5-20

special characters, 6-7

sq (search quit) command, 5-22

srf (sort file) command, 6-12

standard input, 6-9, 6-15, 6-16

standard output, 6-9

start-up procedure

- for disked nodes, 2-2
- for diskless nodes, 2-8 to 2-12

std_keys, 2-17

std_keys.basic, 2-17

std_keys(n) files, 3-13

strings, 5-17 to 5-18, 5-23, 5-24, 5-25, 6-18, 11-18

- in expressions, 11-12
- operators for, 11-14

subject identifier (SID), 4-4, 10-2

subs (subsystem) command, 10-24, 10-26

substituting text, 5-23 to 5-26

substitution parameters, 11-5

sysboot program, 2-4

/sys/dm/login_sh, 6-7

/sys/net/diskless_list, 2-11

/sys/node_data, 2-6

/sys/print, 7-12

/sys/subsys directory, 10-25

T

tl (to left) command, 5-5

tlc (transliterate character) command, 6-10

to statements, 11-34

U

UNDO key, 5-26

undo command, 5-26

updating edit files, 5-27

user_data subdirectory, 3-16

V

- variable commands, 11-20
- variables
 - defining, 11-17
 - defining interactively, 11-21
 - deleting, 11-20
 - verifying, 11-20
- voff** (verification off) command, 11-11
- von** (verification on) command, 11-11

W

- wa** (window autohold) command, 4-24
- wc** (window close) command, 4-16, 5-27
- wd** (working directory) command, 6-3, 7-2
- wdf** (window default) command, 4-24
- wg** (window grow) command, 4-10
- wge** (window grow echo) command, 4-17
- wgra** (window group add) command, 4-31
- wgrr** (window group remove) command, 4-31 to 4-32
- wh** (window hold) command, 4-23
- while** statements, 11-28
- wi** (window invisible) command, 4-32
- wildcards, 6-17

window groups, 4-30

window-movement commands, 4-10, 4-19

windows

- changing size, 4-17 to 4-18
- closing, 4-15
- controlling process modes for, 4-21
- copying, 4-14
- creating, 4-9 to 4-10
- defining boundaries for, 4-10
- deleting, 4-16
- managing, 4-16 to 4-17
- moving, 4-19
- paste buffers for, 4-37
- pushing/popping, 4-20

wm (window move) command, 4-10

wme (window move echo) command, 4-19

working directory, 1-9

- changing, 7-3
- setting, 7-2

wp (window pop) command, 4-20 to 4-21

write mode, 5-3

writing output to a file, 6-11

ws (window scroll) command, 4-23

X

xc (copy text) command, 5-11

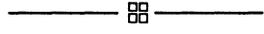
xd (cut text) command, 5-11, 5-13

xdmc (execute DM command) command, 11-9

xi (copy image) command, 5-13

xoff (execution tracing off) com-
mand, 11-11
xon (execution tracing on) com-

mand, 11-11
xp (paste) command, 5-8, 5-11,
5-14





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 Aegis Environment*

Order No.: 011021-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



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 Aegis Environment*

Order No.: 011021-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



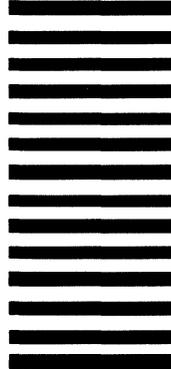
cut or fold along dotted line

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

apollo

Using Your Aegis Environment 011021-A00



011021-A00