DOMAIN/IX
User's Guide

Order No. 005803
Revision 01

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July, 1985
Latest Printing: December, 1986

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

Preface

Audience

The *DOMAIN/IX User's Guide* is based on various papers normally found in the *UNIX Programmer's Manual* supplied by AT&T and the University of California at Berkeley. Although we've modified the papers where necessary to reflect the DOMAIN® system operating environment, we remain aware of the history of the UNIX® product as a multiuser system, and have included the more important references to operations conducted at terminals.

This *User's Guide* is intended for users who are familiar with UNIX software, AEGIS™ software, and DOMAIN networks. We recommend that you read one of the following tutorial instructions if you are not already familiar with the UNIX system:

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

This document also assumes a basic familiarity with the DOMAIN system. The best introduction for those who want to use UNIX software on a DOMAIN node is *Getting Started With Your DOMAIN/IX System* (Order No. 008017). This manual explains how to use the keyboard and display, read and edit text, and create and execute programs. It also shows how to request DOMAIN system services using interactive commands.

Structure of This Manual

The manual is organized as follows:

Chapter 1 Provides an overview of important DOMAIN/IX™ system features.

Chapter 2 Briefly introduces the basic principles of using the shells available through the DOMAIN/IX system.

Chapter 3 Supplies a detailed explanation of Bourne Shell (**sh**) usage, both System V and BSD4.2 versions.

Chapter 4 Describes how to use the C Shell (**csh**), both System V and BSD4.2 versions.

Chapter 5 Tells how to use the BSD4.2 version of the **mail** program to communicate with other users.

Related Manuals

Getting Started With Your DOMAIN/IX System (Order No. 008017) is the first volume you should read. It explains how to log in and out, manage windows and pads, and execute simple commands. It presents user-oriented examples and includes a glossary of important terms.

The *DOMAIN/IX Support Tools Guide* (Order No. 009413) describes the UNIX support tools and utilities available to DOMAIN/IX users. It contains extensive material on tools such as **awk**, **sed**, and **yacc**, which help process programs; **lex** and **lint**, which help analyze programs; and **make** and **sccs**, which help maintain programs. It also describes support tools that preprocess macros (**m4**) or FORTRAN code (**ratfor**), perform arbitrary precision arithmetic (**bc**), operate an interactive desk calculator (**dc**), and provide terminal screen handling with optimal cursor motion (**curses**).

The *DOMAIN/IX Text Processing Guide* (Order No. 005802) describes the UNIX text editors (**ed**, **ex**, and **vi**) supported by the DOMAIN/IX system. It also contains material on the formatters **troff** and **nroff**, the macro packages **-ms**, **-me**, and **-mm**, and the preprocessors **eqn** and **tbl**.

The *DOMAIN/IX Command Reference for System V* (Order No. 005798) describes all the UNIX System V shell commands supported by the *sys5* version of the DOMAIN/IX software. This manual documents various general purpose, communications, and graphics commands and application programs. It also describes games available to the System V user.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005799) describes all the UNIX System V system calls; C, standard I/O, and mathematical library subroutines; file formats; character set tables; and macro packages supported by the *sys5* version of the DOMAIN/IX software.

The *DOMAIN/IX Command Reference for BSD4.2* (Order No. 005800) describes all the BSD4.2 UNIX shell commands supported by the *bsd4.2* version of the DOMAIN/IX software. This manual documents various general purpose, communications, and graphics commands and application programs. It also describes games available to the System V user.

The *DOMAIN/IX Programmer's Reference for BSD4.2* (Order No. 005801) describes all the BSD4.2 UNIX system calls; C, standard I/O, mathematical, internet network, and compatibility library subroutines; special files; file formats and conventions; and macro packages and language conventions supported by the *bsd4.2* version of the DOMAIN/IX software.

System Administration for DOMAIN/IX BSD4.2 (Order No. 009355) and *System Administration for DOMAIN/IX Sys5* (Order No. 009356) describe the tasks necessary to configure and maintain DOMAIN/IX system software services such as TCP/IP, line printer spoolers, and UNIX-to-UNIX communications processing. Also explains how to maintain file system security, create user accounts, and manage servers and daemons.

The *DOMAIN C Language Reference* (Order No. 002093) describes C program development on the DOMAIN system. It lists the features of C, describes the C library, and gives information about compiling, binding, and executing C programs.

The *DOMAIN System Command Reference* (Order No. 002547) gives information about using the DOMAIN system and describes the DOMAIN commands.

The *DOMAIN System Call Reference* (Order No. 007196) describes calls to operating system components that are accessible to user programs.

Documentation Conventions

Unless otherwise noted in the text, this manual uses these symbolic conventions:

command	Command names and command-line options are set in bold type. These are commands, letters, or symbols that you must use literally.
`output`	Typewriter font words in command examples represent literal system output, including prompts.
filename	Italicized terms or characters represent generic, or metanames in example command lines. They may also represent characters that stand for other characters, as in **d***x*, where *x* is a digit. In text, the names of files written or read by programs are set in italics.
[ ]	Square brackets enclose optional items in formats and command descriptions.
< >	Angle brackets enclose the name of a key on the keyboard.
†D	The notation † followed by the name of a key indicates a control character sequence. Hold down <CTRL> while typing the character.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
⋮	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. You can view an on-line description of the command used to submit a UCR (**/com/crucr**) by typing:

```
% /com/help crucr <RETURN>
```

(Although we use a C Shell prompt in our example, you may type this command from any type of shell available to users of the DOMAIN/IX system.)

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Contents

Chapter 3 Using the Bourne Shell

Chapter 4 Using the C Shell

Chapter 5 Using the BSD4.2 Mail Program

Tables

Chapter 1

An Overview of the DOMAIN/IX System

The DOMAIN/IX system is an implementation of the UNIX operating system that runs on DOMAIN nodes. It supports the DOMAIN distributed file system, and multiple networks using bit-mapped, high-resolution displays. In addition to bringing the benefits of a networked architecture and a true single-level store to the UNIX system, the DOMAIN/IX system offers many features seldom found on either time-sharing or workstation implementations of software.

There are two versions of DOMAIN/IX software. The *sys5* version is compatible with UNIX System V, Release 2 from AT&T Bell Laboratories; the *bsd4.2* version is compatible with 4.2BSD, from the University of California at Berkeley. We supply both versions to all DOMAIN/IX customers. In this chapter, we introduce those DOMAIN system features not found in other UNIX systems. We also explain how to use the *bsd4.2* and *sys5* UNIX versions concurrently.

1.1 DOMAIN System Architecture

A DOMAIN system comprises two or more nodes connected by a high-speed local area network. When we mention the term network in this manual, we generally refer to one that has a ring topology, and uses a token-passing protocol to prevent collisions between messages being sent from one node to another. However, DOMAIN systems may also run on other types of networks (e.g., an ETHERNET network). Each node is a functional workstation, with its own central processor, memory, and memory management hardware. Programs and data required by processes running on a node may be demand-paged across the network.

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

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

1.2 The User Interface

We provide for a more varied user interface by supplying features that significantly differ from those provided in other UNIX implementations. The most important difference, from the user's point of view, is the ability of a DOMAIN node to display "windows" into many processes (shells, programs, etc.). These windows have some unique features not found on the "dumb" terminals largely used in the development of UNIX System V and 4.2BSD software.

1.3 Co-resident Software

DOMAIN/IX software is co-resident with the DOMAIN system's AEGIS operating system, sharing many of the same underlying kernel functions. As a result:

- UNIX programs supplied with the DOMAIN/IX system have the same file format as AEGIS programs
- UNIX Shells provided with DOMAIN/IX software can exist on the same screen with AEGIS Shells
- UNIX commands can be executed in an AEGIS Shell
- AEGIS commands can be executed in a UNIX Shell

Normally, there is no distinction between processes that run UNIX programs and those that run other DOMAIN programs. UNIX programs and AEGIS programs can coexist within the same process, even within the same pipeline. In rare cases, naming conflicts (i.e., cases where UNIX and AEGIS programs have the same name) require that you rename or alias a command.

1.4 The Display and the Display Manager

Your node's display is your "window" into the DOMAIN system. Unlike most dumb terminals that dedicate their entire display to a single program or process, DOMAIN

nodes let you divide the display screen into multiple environments for running programs, and reading or editing files. With each new environment you create, the DOMAIN system creates a set of display components through which you can enter input and view output.

What you see through a window is either a "frame" containing graphics or a "pad" containing text. Refer to the *DOMAIN System Command Reference* for more information about frame mode and graphics. Our primary concern in this section is with pads. *Getting Started With Your DOMAIN/IX System* has detailed information on pads, windows, and window legends.

1.4.1 Pads and Windows

There are two principal types of pads: "edit" pads and "transcript" pads. An edit pad is a window into the buffer that the DM sets up when you want to edit a file. A read-only edit pad is a special instance of an edit pad that, either because you have opened the pad in read-only mode or because you have opened a window into a file for which you lack "write" permission, doesn't allow you to modify the contents of the buffer.

All shells run in a window that consists of an "input pad" and a "transcript pad." The input pad echoes the standard input, and the transcript pad gives a running transcript of the standard output. Because it is unwise (or even illegal) to edit history, the transcript pad is unalterably read-only. (The only legal writer is the program.) This combination of an input pad and a transcript window is at least the equivalent of a "terminal," in the sense that the word is used in much of this book. In addition, it has features that go far beyond what most terminals can manage.

An input pad is actually special instance of an edit pad. It can't be made read-only, and it "grows" as necessary when you type input faster than the shell (or other program) can use it. Programs using input pads read input sequentially, one line at a time. As an input line is read, it is scrolled up into the transcript pad, where it remains until the shell is closed. Even after text has scrolled out of the top of the window, the transcript pad never loses any information. Using the pad scroll keys, you can scroll through the transcript pad to review or copy text from any part of the transcript.

When you stop a shell or other program running in a window, the DM normally closes both the input and transcript pads and displays a

```
*** Pad Closed ***
```

message in the window. You can then issue the DM command **wc** (window close, normally mapped to ↑N) to remove the window from the screen.

Note: To save the information contained in a transcript pad, do one of the following (see *Getting Started With Your DOMAIN/IX System* for details on either of these):

- Copy all or part of the pad into an edit pad, paste buffer, or file
- Use the DM's **pn** (pad name) command to write the pad to a disk file.

Edit pads don't interact with programs; they are simply files that you can view or edit using the DM editor. You can also open an edit pad in "read-only" mode if you want to read rather than to edit it.

At the top of every window is a "window legend" that displays the name (or number) of the process running in the window. If the window opens onto a file (i.e., if it is an edit pad, read-only or otherwise), the window legend displays the full pathname of the file and such additional information as the edit mode (insert or typeover), rights (read/write or read-only), file line number of the top line in the window, and horizontal offset if greater than 0.

1.4.2 Default Windows and Shells

In addition to the various shells and edit pads that you may open while logged in to a DOMAIN node, two windows are usually opened by default: one when the node is booted, and another when you log in.

When a node is booted, it normally loads the DM and opens a DM input pad, DM alarm window, and DM output pad. On a landscape display, these windows are each one line high and are placed side by side along the bottom of the screen. When no one is logged in, the DM input pad displays the login prompt:

```
login:
```

Note: If your node is not set up for a UNIX-style login (i.e., UNIXLOGIN is not set to 'true'), your login prompt will be the slightly different AEGIS prompt:

```
Please Log In:
```

(UNIXLOGIN is a DM environment variable that we later detail.)

After you log in, the DM input pad displays this prompt:

```
Command:
```

Pressing <CMD> brings the cursor to the DM input window.

The DM output pad (actually, a window into the file */sys/dm/output*) is broken into two windows: the alarm window and the output window. The alarm window appears to the right of the input window on both landscape and portrait displays. Whenever the DM writes output to a partially obscured or hidden window, it alerts you by sounding the node's alarm beeper and displaying a visible alarm in the form of two "bell" characters in the DM alarm window. The bells are cleared when you <POP> the obscured window to the top of the window stack.

The DM output window appears at the right of the alarm window on landscape displays, or at the bottom of portrait displays. The output window displays DM messages and output from those DM commands (e.g., **kd** and "=") that generate output.

By default, the DM first opens an AEGIS Shell when you log in, and then executes your personal login script of DM commands (e.g., *user_data/startup_dm.19l* in your home directory). You may, of course, arrange for the DM to open a UNIX Shell instead. For more information about DM startup scripts, see the *DOMAIN System User's Guide*.

1.4.3 DM Commands

The *DOMAIN System Command Reference* documents all DM commands that we currently support. The *DOMAIN/IX Text Processing Guide* also covers those used to edit text or examine transcripts. In general, all DM commands can be

- Entered in the DM input window
- Placed in a command file for execution as needed (e.g., at log-in)
- Bound to DOMAIN keyboard keys, via the DM's **kd** (key definition) command.

1.4.4 Regions

Some DM commands deal with the whole screen, although most pertain to an individual window, or a region within the pad on which a window opens. Since the concept of a screen divided into regions may be new, we briefly introduce it here.

When you move the cursor to the DM input window, the DM first notes the cursor's location on the screen. Thus, it derives information such as the current working directory of a shell, the location of the cursor in an edit pad, or the current location of a window that you intend to move or "grow". (Actually, the DM gets this "current context" information from the last place on the screen where an event took place.) The same is true when you press a key that has been defined to invoke a DM command sequence. For example, when you press <EDIT>, the DM first notes the current working directory of the shell window that the cursor last occupied. If you type

```
edit file: foo <RETURN>
```

the DM looks for a file named *foo* in the current working directory of that shell. If the file exists, the DM opens an edit pad onto it. Otherwise, the DM creates *foo* and opens a blank edit pad.

Even though you may have many windows open on your screen, the DM assumes that you can only be actively addressing one at a time. By keeping track of the cursor, the DM keeps track of your involvement with processes running in windows on your node. Since it also keeps track of what all processes (even those not occupied by the cursor) are doing, the DM can also alert you when something occurs in a hidden or partially obscured window. By operating in this manner, the DM can provide services to all processes running in windows on your node.

1.4.5 Moving the Cursor

While there are many ways to get the DM to move the cursor, the arrow keys at the left of the keyboard are the most obvious. Many keyboards are also equipped with a mouse (which has three programmable function keys) or a touchpad. Both are effective tools for large-scale cursor movements. See *Getting Started With Your DOMAIN/IX System* for more on the mouse and touchpad.

Explicit DM commands also move the cursor, although they rarely see interactive use. The arrow keys and the other keys that move the cursor are simply defined at startup so that these commands are executed. <CMD> moves the cursor to the DM input window, and <NEXT WNDW> moves the cursor to the next unobscured shell input pad or read/write edit pad.

Note: The DM considers a window to be obscured if any part of it is covered by another window. If no unobscured shell windows or read/write edit pads appear on the display, <NEXT WNDW> has no effect. A read-only edit pad isn't a candidate for <NEXT WNDW> either.

1.4.6 Keyboard Mapping

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

The DOMAIN system supports three types of keyboards: the Low-Profile Model I keyboard, the Low-Profile Model II keyboard, and the 880 (high-profile) keyboard.

Note: The 880 keyboard is no longer shipped with new nodes.

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

- *std_keys3* keyboard definitions for the Low-Profile Model II keyboard
- *std_keys2* keyboard definitions for the Low-Profile Model I keyboard
- *std_keys* keyboard definitions for the 880 keyboard

1.4.7 UNIX Key Definitions

Alternate versions of the standard key definitions, modified to provide necessary UNIX functions, reside in the */sys/dm directory*. These alternate versions are named as shown in Table 1-1. (Equivalent files for the 880 keyboard are *sys5_keys* and *bsd4.2_keys*.)

Table 1-1. UNIX Key Definition Files

Filename	Contents
sys5_keys2	System V UNIX keyboard definitions for the Low-Profile Model I keyboard
bsd4.2_keys2	BSD 4.2 UNIX keyboard definitions for the Low-Profile Model I keyboard
unix_keys2	Generic UNIX keyboard definitions for the Low-Profile Model I keyboard
sys5_keys3	System V UNIX keyboard definitions for the Low-Profile Model II keyboard
bsd4.2_keys3	BSD 4.2 UNIX keyboard definitions for the Low-Profile Model II keyboard
unix_keys3	Generic UNIX keyboard definitions for the Low-Profile Model II keyboard

The BSD4.2 and the System V definitions files include commands that bind various keys to certain version-specific (or shell-specific) features. They are described in detail in Chapters 3 and 4 of this manual, which deal with the Bourne Shell and the C Shell, respectively. Initially, none of these key definitions files are automatically invoked, although you may arrange for them to be, as we shall explain later.

To put any key-definition file into effect, execute the **cmdf** (command file) command at the Display Manager prompt, where the filename argument is one of the key defini-

tions files mentioned earlier. For example, this invokes the BSD4.2 version UNIX key definitions on a Low-Profile Model I keyboard:

```
Command: cmdf /sys/dm/bsd4.2_keys2 <RETURN>
```

When the keyboard is remapped to *bsd4.2_keys2*, the following keys are redefined:

<SHELL> This DM function key executes the DM command **cp /bin/start_csh**, which creates a C Shell and runs a personal UNIX log-in file (*.login*).

<TAB> When shifted, this key inserts a literal ASCII tab character.

<READ> This DM function key, which calls the DM to read a file, displays a different prompt -- "read file: " rather than "Read file: ". Also, arguments supplied as input are treated with case-sensitivity.

<EDIT> This DM function key, which calls the DM editor, takes a different prompt -- "edit file: " rather than "Edit file: ". Also, arguments supplied as input are treated with case-sensitivity.

↑I This control-key sequence generates an interrupt signal.

↑D This control-key sequence produces an end-of-file (EOF) signal.

↑Z This control-key sequence generates a suspend signal.

↑J This control-key sequence breaks a previous suspend signal (produced by using ↑Z).

On a Low-Profile Model I keyboard, invoking the System V (*sys5_keys2*) or the generic UNIX key definitions (*unix_keys2*) produces similar results. <TAB>, <READ>, and <EDIT>, and the ↑I and ↑D control-key sequences, work as described above, but these other keys behave differently:

<SHELL> This key executes **cp /bin/start_sh**, which creates a Bourne Shell and runs a personal UNIX log-in file (*.profile*).

↑Z This control key sequence produces an EOF signal.

↑J This control key sequence does nothing.

1.4.8 DM Environment Variables

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

```
name = value
```

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

For processes that use multiple program levels, environment variables are markreleased so that, while a new program level inherits all environment variables from a previous level, a new level can't affect the environment variables of a previous level. When a new process is created, all environment variables of the creating process are inherited

by the new process. All process creation mechanisms (e.g., **pgm_$invoke, fork, vfork**) provide for this inheritance.

Note: Environment variables still existing in a process when an AEGIS Shell is created are automatically inherited by that shell. The Bourne and C Shells handle environment variables as defined by UNIX semantics. (See Chapters 3 and 4.)

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

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

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

A program interface for environment variable usage is defined in the */sys/ins/ev.ins.?** files. C language programs may manipulate environment variables through these interfaces. Alternatively, C programs may use the UNIX calls **getenv**(3) and **putenv**(3) or access the external environ variable. All interfaces are compatible with one another; e.g., a variable defined with **putenv**(3) may be read using **ev_$get_var**.

Certain environment variables are well-known. Some are predefined by the system at login; others have special significance to system software or other special attributes. Table 1-2 shows the environment variables used by the DOMAIN/IX system.

Table 1-2. Summary of Environment Variables Used by the DOMAIN/IX System

Variable Name	Description
USER	User's login name.
LOGNAME	Synonomous with USER. The synonyms are provided to support both versions of DOMAIN/IX.
PROJECT	Project (group) ID under which the user logged in.
ORGANIZATION	An additional group ID that the user may specify.
NODEID	The unique node identifier for the node on which the process is running; expressed in hexadecimal.
NODETYPE	The type of node on which the process is running.
HOME	The user's home directory pathname, established at login.
TERM (Predefined)	The device name of the "terminal" in use. We define this for the sake of C or UNIX programs that are terminal-dependent. Values for our displays are of the form "apollo_xxx". See /bsd4.2/etc/termcap for a list of all valid terminal types.
SHELL	The pathname of the shell in which the process is running (i.e., /bin/sh, /bin/csh)
TZ (Predefined)	The timezone string. Like TERM, this variable is defined for the sake of C or UNIX programs. The value format is SSSnDDD, where SSS is the standard timezone name (e.g., EST), n is the difference in hours between the standard timezone and UTC, and DDD is the daylight timezone name.
COMPILESYSTYPE	Defines the target UNIX system version.
SYSTYPE	UNIX system version in use (i.e., bsd4.2, sys5, bsd4.1, sys3)
UNIXLOGIN	Specifies that a UNIX-style login sequence is to be used in place of the DOMAIN login sequence. This feature is available in the Display Manager, Server Process Manager, and /com/login. Valid values for UNIXLOGIN are true and false. Only used at startup, not in shells environment.
PROJLIST	Specifies multiple groups to which a user belongs. This variable is automatically set only if UNIXLOGIN is true and SYSTYPE = bsd4.2. System V users may still manually set this variable to get the multiple group feature. Only used at startup, not in shells environment.
NAMECHARS	Specifies a set of characters to which special semantics are values described later in this chapter.

1.5 Support For Multiple UNIX Versions

The two versions of the UNIX operating system supported by DOMAIN/IX software provide a variety of similar – though seldom identical – system services through kernel and library functions. Often, while function *x* exists in both the *sys5* and the *bsd4.2* environments, the semantics of the function, and sometimes even its arguments, may be subtly different.

Let's consider the kernel function **setpgrp**(2). AT&T Systems III and V define it as:

```
int setpgrp ()
```

to "set the process group id of the calling process to the process id of the calling process and return the new process group id." 4.1BSD and 4.2BSD define a function with the same name and similar semantics, but a different calling sequence:

```
setpgrp (pid, pgrp)
int pid, pgrp ;
```

"sets the process group of the specified pgrp. Zero is returned if successful; -1 is returned and errno is set on a failure."

Nearly every non-trivial C program written to run under the UNIX operating system assumes the run-time environment to be UNIX software of a certain lineage (AT&T or Berkeley), or even a specific version (AT&T System V or 4.2BSD). The UNIX version acts as a modifier of the compile-time environment, and, to a greater extent, of the environment in which the program executes. Our multiple version support is based on this assumption.

At compile time, you select the version of UNIX software for which your program is targeted. This version selector is an environment variable called SYSTYPE. The value of SYSTYPE determines, among other things, which version of */usr/include* the compiler goes to when it needs an include file. The object module produced by the compiler is stamped with the SYSTYPE in effect when the module was compiled. When the program is executed, the loader checks this stamp and makes sure that the proper semantics and calling sequences are used when invoking system and library functions.

Table 1-3. SYSTYPES Supported by the DOMAIN/IX System

sys5	AT&T System V, Release 2
bsd4.2	Berkeley 4.2BSD
sys3	AT&T System III (for backward compatibility)
bsd4.1	Berkeley 4.1BSD (for backward compatibility)
any	Program is independent of a particular UNIX version (highly unlikely)

You may express the targeted version or "systype" to the C compiler by including it in the source file itself. Use the **#systype** directive (supported by DOMAIN C compiler) followed by one of the values in Table 1-3. The **#systype** statement should be the first non-comment statement in the source. Here's an example:

```
#systype "sys5"
main()
{
setpgrp () ;
}
```

Note that the "systype" string is placed in double quotes; the C compiler complains if you don't follow this rule. You can also specify the "systype" on the compiler command line. For the DOMAIN C compiler, **/com/cc**, use the form **-systype** *value*. For the DOMAIN/IX system interface to **/com/cc**, **/bin/cc**, use the form **-T***value*. For example, in an AEGIS Shell, type:

$ cc berkprog.c -systype bsd4.2 <RETURN>

In a C Shell, enter this:

% cc -Tsys5 bellprog.c <RETURN>

If you specify one "systype" on the command line and a different one in the file, the compiler objects. If you don't explicitly specify a "systype" in the source text or on the command line, the value of SYSTYPE is inherited from an environment variable called COMPILESYSTYPE.

If the COMPILESYSTYPE environment variable exists, its value, which must be one of the strings listed above, is used. If COMPILESYSTYPE doesn't exist, the "systype" is inherited from the SYSTYPE environment variable. For example, to compile all programs to run in a *sys5* environment, set COMPILESYSTYPE in a *sys5* Bourne Shell:

 # COMPILESYSTYPE = sys5 <RETURN>
 # export COMPILESYSTYPE <RETURN>

In a C Shell, the line is:

% setenv COMPILESYSTYPE sys5 <RETURN>

As long as COMPILESYSTYPE is thus set, all C programs are compiled to run in the *sys5* environment. For backward compatibility, if neither COMPILESYSTYPE nor SYSTYPE environment variables exist, the object file is stamped as having a SYSTYPE of *sys3* (AT&T System III).

Note: Any newly-created process that takes its context from a process in which an environment variable was defined and exported recognizes the new variable. Processes already created (or those created later) that don't take their context from the process where the variable was defined won't apply the variable. See Chapters 3 and 4.

1.5.1 Name Space Support

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

Symbolic links placed in your node's root directory during the installation procedure let programs use either the *sys5* or *bsd4.2* versions of the */bin*, */etc*, and */usr* directories (*/tmp* and */dev* are common to both). Normally, the links to */bin*, */usr*, and */etc* are created by the installation script; if, at some time, you need to re-create them, use **ln**(1).

For example, to create a SYSTYPE-dependent link for */bin*, type this:

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

Note: The single quotes around the link text are required, to prevent the dollar sign from being interpreted as a shell metacharacter.

The SYSTYPE environment variable is used to select the UNIX file system variant, and therefore, commands, libraries, spool directories, and so on. Table 1-4 shows the top-level DOMAIN/IX directory organization.

Table 1-4. Top-Level DOMAIN/IX Directory Organization

Name	Object Type	Major Subdirectories
/usr	variant link	-
/bin	variant link	-
/etc	variant link	-
/dev	ordinary link	-
/bsd4.2	directory	/usr, /bin, /etc
/bsd4.1	directory	/usr/include
/sys5	directory	/usr, /bin, /etc
/sys3	directory	/usr/include
/tmp	ordinary link	-

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

The variant links for *sys3* and *bsd4.1* are limited to */usr/include*. References to other *sys3* directories are resolved as they would be for *sys5*. References to other *bsd4.1* directories are resolved as they would be for *bsd4.2*. This ensures that programs compiled to run in the *sys3* or *bsd4.1* environments get the proper include files, but it means that when you invoke a *sys3* or *bsd4.1* environment for interactive use, you do not get the "old" versions of, for example, commands and macro packages.

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

```
% ls /tmp       <RETURN>
cattoc              ipc.out              toc143
% ls //foo/tmp  <RETURN>
cattoc              ipc.out              toc143
%
```

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

```
% ls //foo/sys/node_data/tmp <RETURN>
dirs     ln
%
```

1.5.2 Environment Switching

The object–module stamping scheme, described earlier, lets you execute System V programs from any BSD4.2 shell and vice versa, without any knowledge of the UNIX version for which the program was targeted. When you invoke a program stamped with a *systype* other than *any*, the SYSTYPE environment variable for the process in which the program is running is set to the value found in the object module. This ensures that programs of one UNIX version that depend on certain system files continue to work when executed from a process running in another version. The */etc/systype* program displays the version stamp of the specified object files.

A shell's SYSTYPE value defines the version (*sys5*, *bsd4.2*) of system directories that are searched when a command name is given; hence, it defines the version of the command that is executed.

To simplify the execution of a version *x* command from a version *y* shell, we provide a "set–version" command. See **ver**(8) in the *DOMAIN/IX Command Reference for BSD4.2*, or **ver**(1M) in the *DOMAIN/IX Command Reference for System V*. You can use **ver** in the following three ways:

- To display the current value of SYSTYPE, execute **ver** with no arguments. For example, the following checks SYSTYPE and finds it to be set to *bsd4.2*:

  ```
  % ver <RETURN>
  bsd4.2
  ```

- To change SYSTYPE to *value*, thereby changing the version of subsequently executed commands, use the form **ver** *value*. For example, the first command line sets the SYSTYPE to *sys5*, and the second command line executes a *sys5* version of **ls** (SYSTYPE remains the same until it is reset):

  ```
  % ver sys5          <RETURN>      (Set SYSTYPE to sys5)
  % ls                <RETURN>      (Do an ls)
  prog.c
  prog.o
  testfile
  %
  ```

  Note: Using **ver** with a single argument of either *sys5* or *bsd4.2* simply changes the value of SYSTYPE. If you execute the command

  ```
  B$ ver sys5 <RETURN>
  ```

  in a *bsd4.2* Bourne Shell, it is equivalent to saying

  ```
  B$ SYSTYPE=sys5 <RETURN>
  ```

- To execute the *value* version of *command* without changing SYSTYPE, use the form **ver** *value command*. For example, the first command executes the *sys5* version of **id**(1), while the second command line executes the *bsd4.2* version (the default, in this case) of **ls**(1):

```
% ver sys5 id          <RETURN>
uid=212(kate) gid=38(unix)
% ls                   <RETURN>
prog.c prog.o testfile
```

1.6 Setting Up a UNIX–Style Login Sequence

You may arrange for the DM, Server Process Manager, and */com/login* to use a UNIX–style login sequence by including the following in a DM startup file:

```
# put this line in 'node_data/startup
# if you want to use a UNIX-style login sequence
env UNIXLOGIN true
```

When UNIXLOGIN is true:

- the prompt is changed to "login: "
- the rejection message is changed to "login incorrect"
- upon successful login, the file */etc/dmmsg* is displayed in the DM output window

BSD4.2 users whose UNIXLOGIN is set to true are automatically placed in their project lists at login. Although multiple groups do not exist in other implementations of System V software, DOMAIN/IX System V users can get the multiple group feature by manually setting the PROJLIST variable.

In addition, the "backup" group (%.backup) is special–cased out of the project list in UNIXLOGIN. This eliminates the possibility of login problems that would otherwise occur if you had multiple accounts (one of which is a backup account) and the shell you invoke with the login had no execute rights for %.backup. In such a case, you would not be allowed to invoke the shell if the first account matched was the %.backup account. The "sys_admin" and "locksmith" accounts are also special–cased out of the project list to prevent similar problems.

1.7 Message of the Day (motd)

Regardless of how UNIXLOGIN is set, the specified UNIX Shell (*/bin/sh* or */bin/csh*) reads an acceptance message from */etc/motd* and then displays it in the transcript pad of the shell window. To suppress the display of the */etc/motd* file, use the **–s** (silent) switch when invoking a shell. The following key definition creates new windows without printing the message of the day:

```
kd 15s cp /bin/start_sh '-s' ke
```

If */etc* is a variant link (the usual case), the SYSTYPE variable must be set to *bsd4.2* or *sys5*; otherwise, the DM cannot locate */etc/motd*. If *etc/motd* isn't found at startup, a standard AEGIS login acceptance message, minus the "project" and "organization" fields, appears in the DM output window.

1.8 Setting Up Server Processes

The */etc/rc* file (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 processes that must be invoked by the super-user ("root"). On DOMAIN systems, *'node_data/etc.rc* is executed by the **/etc/run_rc** command. Note, however, that SYSTYPE must be set early in the startup, before trying to run **/etc/run_rc**.

```
# put this line in 'node_data/startup if you want to run
# 'node_data/etc.rc whenever the node is booted
cps /etc/run_rc
```

The *'node_data/etc.rc* file must be owned by "root" and have a UNIX file system mode of 4755 (i.e., have the setuid bit on). The **run_rc** program runs this file as "root", then may **exec** another file, *'node_data/etc.rc.local*, to which any user may add commands that do not have to run (and will not be run) with a user ID of "root". For more information, see the manual page for **rc**(8) or **rc**(1M).

1.9 Getting Help

For information about available UNIX commands, system calls, and functions, use the **man**(1) command. This command lets you select and display on-line versions of reference material from the *DOMAIN/IX Command Reference*, the *DOMAIN/IX Programmer's Reference*, and the *DOMAIN/IX System Administration* manuals. For example, to display the manual page for the command **who**(1), type this in any UNIX Shell:

```
% man who <RETURN>
```

The **man** command then opens a read window containing a formatted version of the manual page(s) on the **who**(1) command. See the *DOMAIN/IX Text Processing Guide* for more information on how to scroll through and search for patterns (keywords) in these windows. While the manual page is displayed, you may continue to execute shell commands (including other **man** commands). When you're finished reading the manual page, type †N or press <EXIT> to close the window.

Note: The **man** command uses the symbolic links in effect for the SYSTYPE of the shell in which it is executed. When **man** is executed in a shell with a SYSTYPE of *sys5*, manual pages are read from */sys5/usr/catman*. When executed in a shell with a SYSTYPE of *bsd4.2*, they are taken from */bsd4.2/usr/man*.

If no manual page is available for the particular command, call, or function that you specify, **man** outputs the message "No manual entry for..."

1.10 Other DOMAIN/IX Features

This section contains miscellaneous facts that will be especially helpful if you plan to develop applications software to run on your DOMAIN system.

1.10.1 The Process Model

Some implementations of UNIX versions use a one-program-per-process execution model. In this model, invocation of a new program causes a separate process to be

created using the **fork**(2) system call. The DOMAIN system, on the other hand, favors a multiple-programs-per-process model in which an invoked program runs at a new program level in the invoking process. The DOMAIN/IX C Shell includes support for a shell variable called *inprocess*. When **set**, it specifies in-process execution (the standard DOMAIN process model); when **unset** (the default value), it specifies the traditional process model.

You can set *inprocess* as a DM environment variable. In fact, we recommend doing this if you plan to access, from the C Shell, objects managed under the DOMAIN Software Engineering Environment (DSEE). To set *inprocess* in the DM, put the following line in any DM command file read before the C Shell is started (e.g., *'node_data/startup*):

```
env INPROCESS 'true'
```

If *inprocess* is set in this way, the C Shell runs as if your *.cshrc* contained this line:

```
set INPROCESS
```

If an "env INPROCESS" line isn't present in a DM startup-file, or is not set to 'true' or 'TRUE', the shell variable *inprocess* determines the process model used by a shell. Remember, however, that the C Shell uses **unset** as the default value of *inprocess*. The C Shell doesn't export *inprocess* to the DM if you set it in *.cshrc* or *.login* files.

Each process model has certain advantages and disadvantages. Chapter 4, which describes the C Shell, supplies details about the use of *inprocess*, including a summary of associated benefits and drawbacks.

1.10.2 Case Mapping

DOMAIN/IX component names may contain any ASCII character except slash and null. Uppercase alphabetics and certain other characters are stored as two-character escape sequences. Component names are limited to 32 characters, including escape characters. A component name consisting exclusively of uppercase alphabetics is limited to 16 characters, since each character is stored as a two-character escape sequence. Table 1-5 shows filename case mapping considerations. Any character not listed in the first column is passed unchanged to the DOMAIN naming server.

Note: In some cases, a character requires an escape only if it is used as the first character of a component name.

Table 1-5. Filename Case Mapping

Character in UNIX name	Sequence in AEGIS name	Sequence if character is first component
<space>	:_	::_
:	::	::
A-Z	:a-:z	:a-:z
a-z	a-z	a-z
‘	‘	:‘
~	~	:~
\	:|	:|
:.	.	

In addition to the mapping rules summarized thus far, the control characters ↑A – ↑_ (hex 01–1F) are mapped using the representation

```
:#xx
```

where *xx* is the hex value of the control character. For example, a pathname component Ab↑C is mapped as

```
:ab:#03
```

When a pathname component includes an uppercase alphabetic, backslash, colon, or initial dot/tilde, that character adds two characters to the total number of characters in the component. See Table 1–6 for some examples.

Table 1–6. Other Filename Characteristics

UNIX name	AEGIS name	Length (characters)
README	:r:e:a:d:m:e	12
L–devices	:l–devices	9
passwd	passwd	6
.cshrc	:.cshrc	7

An AEGIS Shell displays uppercase letters and other characters that require an escape in their escaped form. If you need to create an uppercase (or other escaped) character in an AEGIS Shell, you must escape it with a colon when you create the name.

By default, all characters except slash and null are mapped and stored in component names. We provide an additional feature of special meanings for the tilde (~), backslash (\), and backquote (`). Use the NAMECHARS environment variable to ensure that any or all of these characters retain the following special meanings when read by the naming server:

tilde — home directory (or "naming directory")

backslash — parent directory

backquote — "this_node/sys" (e.g., *`node_data*)

Note: Although the DOMAIN/IX system kernel automatically sets

```
env NAMECHARS '~\'
```

that default setting is overridden by any "env NAMECHARS" line that you include in your DM startup file. Thus, to retain the ability to access the naming directory with a leading tilde and the parent directory with a backslash, and to reference *`node_data*, set NAMECHARS to the string ~`\ by including the above line in a DM startup file. Also, remember not to begin filenames with a backquote or tilde character.

The most obvious symptoms of an undefined backquote relate to the naming server's inability to find the directory *'node_data*. This results in such problems as the failure of server daemons to start at boot time (they are invoked from *'node_data/etc.rc*) and an inability to find */dev* (a link to *'node_data/dev*) and */tmp* (a link to *'node_data/tmp*).

Where references using special characters are coded into programs, we recommend that a network-wide standard be established for the value of NAMECHARS. For programs that are to be transported to other networks or systems, take special care with respect to this feature.

1.10.3 Password and User Identification

The process of login verification and home-directory setting are always handled by the DOMAIN system's login mechanism, but we provide a way to generate an */etc/passwd* file so that UNIX programs that need to access it can do so. To ensure that users at a site have accounts on both the DOMAIN network registry and in */etc/passwd*, your system administrator must invoke **crpasswd**(8) or **crpasswd**(1M) each time a new user account is added or changes must be made to */etc/passwd*. See *System Administration for DOMAIN/IX BSD4.2* or *System Administration for DOMAIN/IX Sys5* for further information on this.

Users should not tamper with the */etc/passwd* file themselves, because of the danger of removing it accidentally. If this happens, UNIX user IDs assigned by a subsequent run of **crpasswd** may not map correctly to system UIDs held in */sys/node_data/acl_cache*. One result of such a mapping inconsistency is that **chmod**(1) changes the owner of a file as well as the access mode.

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

1.10.4 File Protection, Permissions, and Ownership

The normal protection mechanism in the DOMAIN environment is the access control list (ACL). Every object (file, directory, etc.) has an ACL associated with it. The ACL mechanism includes support for all of the access modes normally associated with the UNIX system, including directory search and delete-from-directory.

We provide a **default_acl**(2) system call that allows programs to specify either UNIX access mode or ACL as the means of object protection. When the default is to use ACLs, the system assigns all files, pipes, and directories created with **creat**(2), **mknod**(2), **open**(2), and **mkdir**(2) an ACL corresponding to the value of the mode specified in the call, modified by the current **umask**(1) value.

Note: If an object's ACL specifies more than one "group" owner, its UNIX access mode shows group rights for only one of the groups. In this case, ownership is determined by a uid sort (the "first" group owner in the access control list is given ownership) and is therefore non-deterministic.

Objects created in (or moved into) directories with a *nil* or unset initial file or initial directory ACL can have "DOMAIN/IX ACLs" applied to them automatically. We provide a special program, called **sup**, that converts the protection scheme of your pre-

SR9.5 directories from ACLs to DOMAIN/IX modes. Note, however, that this program should never be run on software installed by DOMAIN system installation programs (e.g., */bin*, */usr*, */etc*), For additional information, see **sup**(8) in *System Administration for DOMAIN/IX BSD4.2* or **sup**(1M) in *System Administration for DOMAIN/IX SYS5*.

The DOMAIN system's single-level store requires that file system objects must be readable if they are to be executable or writeable. When you produce a file via **creat**(2), it is readable and writeable by the owner, regardless of the mode you specify with **creat**. Use **chmod**(1) to change these permissions if necessary, but be aware that if you use it to make a file "execute only" or "write only" for owner or any group, the "read" bits are also turned on. For example:

```
# ls -l              <RETURN>
foo -rwxrwxrwx 1 bob doc 9755 May 23 11:04 foo
# chmod 111 foo      <RETURN>
# ls -l foo          <RETURN>
-r-xr-xr-x 1 bob doc 9755 May 23 11:04 foo
# chmod 555 foo      <RETURN>
# ls -l foo          <RETURN>
-r-xr-xr-x 1 bob doc 9755 May 23 11:05 foo
#
```

Note: In an AEGIS Shell, C Shell, or System V Bourne Shell, the super-user can execute a file that has no specified execute rights for user, group, or others (e.g., a file with permissions set to rw-rw-r--).

Unless the initial file ACL of the current working directory is *nil*, the DM ignores the current **umask**(1) value when assigning rights to a file it creates. Instead, it uses the default file ACL. If the default file ACL does not specify an owner, UNIX programs consider its owner to be "<none>".

Note: If using the DM editor to create *.login*, *.cshrc*, or *.profile* files, remember that UNIX Shells will read these only if they're owned by the user opening the shell.

Using **chmod**(1) on an object owned by "<none>" resets the owner ID of that object to that of the user running the **chmod** command. The DOMAIN/IX implementation of **chmod** also changes the "last time modified" associated with that file. Although the **chown**(1) command is normally used to change owner ID on a file, it is a privileged command in the *bsd4.2* version of DOMAIN/IX software (i.e., can only be run by super-user).

1.10.5 Use of the C Compiler

The DOMAIN/IX C compiler uses the DOMAIN common code generation mechanism, and it produces a non-standard *a.out* file. The DOMAIN C compiler (*/com/cc*) provides some unique options not offered along with the standard UNIX C compiler. See the *DOMAIN C Language Reference* for further details.

When invoked, the C compiler invokes */usr/lib/cc* (a soft link to */com/cc*). Since it is not hard-coded to */com/cc*, the DOMAIN/IX C compiler provides for greater flexibility and ease in linking to alternate C compilers. It should also be noted that both **cc** and the UNIX link editor, **ld**(1), look for */usr/lib/bind* (a soft link to */com/bind*). For further information about the DOMAIN/IX C compiler, see the *DOMAIN/IX Command Reference*.

Chapter 2

Introduction to Shell Usage

The DOMAIN/IX system supports several types of shells, including two UNIX shells (the Bourne Shell and the C Shell), and the standard shell used by the DOMAIN system's AEGIS operating system (the AEGIS Shell). Since we supply both the *sys5* and *bsd4.2* versions upon installation of DOMAIN/IX software, we provide two versions of the Bourne Shell. The C Shell is also available to both *sys5* and *bsd4.2* users.

Although each shell provides for I/O redirection, pipes, shell procedures (scripts), and metacharacters (wildcards), the implementation of these features frequently varies. This chapter highlights the subtle differences between shells, alerting you to shell characteristics that, while similar on the surface, may produce somewhat different results. Chapters 3 and 4 provide in-depth information about the Bourne and C Shells, respectively. The AEGIS Shell is detailed in the *DOMAIN System User's Guide*.

2.1 UNIX Shells

This section explains how to start a UNIX Shell on a DOMAIN node or on a terminal connected to a DOMAIN node.

2.1.1 Opening a Default UNIX Shell

You may arrange to have the DM (Display Manager) open a UNIX Shell whenever any user logs in to the node, or only when you log in to the node.

If you want every user to get a UNIX shell when they log in, add a **start_sh** or **start_csh** command line to one of the following files:

- For a node that has its own disk: */sys/node_data/startup_login.type* (where *type* is the type of display the node has)
- For a diskless node: */sys/node_data.xxxx/startup* on the partner node (where *xxxx* is the node ID of your node)

If you would like to get a UNIX shell only when you log in to the node, edit your own *user_data/startup_login.type* file.

The file below is for a node that has a 19-inch landscape display (e.g., a DN 320). It is executed whenever anyone logs in to this node. We have added lines that create a process running **/bin/start_sh**, one that runs **/bin/start_csh**, and another that runs the **/etc/rc** file. The pound signs (#) indicate comment lines. In addition, actual command lines in the file have been set in bold face to make them stand out in this example. (In practice, the DM has no such capability.)

```
# STARTUP_LOGIN.19L
# executed for every user logging in to this node.
#
# Assumes that the file 'node_data/startup includes the line:
# env SYSTYPE 'sys5'
#
# Open an Aegis Shell in a rectangular window
# at the the left of the screen (commented out).
#(0,500)dr;(799,955)cp /com/sh
#
# Open a Bourne Shell in the upper left-hand
# corner of the screen. (SYSTYPE is sys5).
(0,0)dr;(430,300)cp /bin/start_sh
#
# Open a bsd4.2 C Shell. (SYSTYPE in this shell will be bsd4.2).
(500,0)dr; (1023,500)cp /bsd4.2/bin/start_csh
#
# Execute the user's personal startup file (it may contain
# other key definitions or may start other processes).
cmdf user_data/startup_dm.19l
#
```

Note: In this file, we assume that the environment variable *SYSTYPE* has already been set in the DM command file *'node_data/startup.type*. Instructions for doing this are in the first chapter of this manual.

As you can see, the second **cp** command explicitly referenced */bsd4.2/bin*, since the Display Manager would override another **env** command with the SYSTYPE value it inherited from the C Shell process.

The default *sys5* Bourne Shell prompt is a pound sign (#) followed by a space. The default *bsd4.2* Bourne Shell prompt is the character sequence

```
B$
```

followed by a space. The default C Shell prompt is a percent sign (%) followed by a space. Any of these prompts can be changed from within the shell.

2.1.2 Opening Additional UNIX Shells

In addition to the shells created at login, you may need to create (and remove) other shells while you are logged in. If you have invoked one of the key definitions files discussed in Chapter 1, you may simply press (shifted) <SHELL>. The *unix_keys* and *sys5_keys* files redefine this key to invoke a Bourne Shell. The *bsd4.2_keys* file redefines this key to invoke a C Shell.

If necessary, you can change the definition of <SHELL>. The *unix_keys* file normally includes this line, which opens a login Bourne Shell (**/bin/start_sh**):

```
kd l5s cp /bin/start_sh ke
```

If you prefer to have <SHELL> open a C Shell instead, change the line to

```
kd l5s cp /bin/start_csh ke
```

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

```
Command: cp /bin/start_sh <RETURN>
```

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

```
Command: cp /bin/start_csh <RETURN>
```

The Display Manager creates the specified shell process in a window with a transcript pad and input pad. The SYSTYPE inherited from the most recent cursor position determines which */bin* is used. You may also specify */sys5/bin* or */bsd4.2/bin* to force creation of a shell with a given SYSTYPE.

2.1.3 Using a Terminal

To access a DOMAIN node via a tty device (an ASCII terminal), you must use a different procedure for creating a UNIX Shell accessible via either a hard-wired or phone line connection to a DOMAIN node's SIO (Serial Input Output) line.

From a shell running on the node to which the device is connected, type

```
# start_sh /dev/sion <RETURN>
```

where *n* is the number of the SIO (Serial Input/Output) line to which the terminal is connected. You can get the same results by going to the DM input window and typing

```
Command: cpo /bin/start_sh /dev/sion <RETURN>
```

The resulting shell process is called *sh.n* for Bourne Shells, or *csh.n* for C Shells; *n* is the UNIX process ID. Running a UNIX Shell on an SIO line affects SIO line characteristics as shown in Table 2-1.

Table 2-1. SIO Line Characteristics Affected by Running a UNIX Shell

Option	Meaning
-QUIT	Quits enabled; default char ↑]
-INT	Interrupts enabled; default char ↑ C
-NOSUSP	Process suspension not enabled
-DCD_ENABLE	Loss of data carrier detect causes hangup fault 9

The last close of the SIO line causes the node's serial I/O hardware to drop the DTR (Data Terminal Ready) signal. This causes most modems to hang up the phone. For more information about SIO line characteristics, see the **tctl** command in the *DOMAIN System Command Reference*.

Note: Be aware that DOMAIN system serial line architecture sometimes causes unpredictable results if you attempt to use a terminal that doesn't expect eight-bit characters.

When the **start_sh** and **start_csh** programs are used to start a UNIX shell on an SIO line, they bind various functions (signals) to control characters as noted in Tables 2–2 and 2–3.

Table 2–2. Control Characters Defined in a C Shell

erase	↑H (backspace)
kill	↑U
interrupt	↑C
suspend	↑Z
eof	↑D
quit	↑\

Table 2–3. Control Characters Defined in a Bourne Shell

erase	↑H (backspace)
kill	↑U
interrupt	DEL
eof	↑D
quit	↑\

When you log in via the siologin process, the initial shell that appears is determined by a line in the file *-user_data/startup_sh*. Put the pathname to the shell you want to use in this file. For example:

```
# DM file -user_data/startup_sh
# this example runs a sys5 Bourne Shell
/sys5/bin/start_sh
```

2.2 Differences Between UNIX and AEGIS Shells

Differences between the AEGIS and UNIX Shells affect the following areas:

- command search rules
- shell program execution
- wildcards
- pathname mapping
- command names and functions

A program is said to be running in the AEGIS environment if it has been invoked in an AEGIS Shell, and in a UNIX environment invoked in a UNIX Shell. Nearly all AEGIS commands reside in the */com* directory.

2.2.1 Command Search Rules

You should be familiar with the material on SYSTYPE and multiple version support in Chapter 1 of this manual. Command search rules are modified by the SYSTYPE environment variable.

Each shell, AEGIS as well as UNIX, has a built–in command search path. The exact path depends on the shell. UNIX shells look in these places for commands:

- the current directory, then
- */bin*, then
- */usr/bin* and
- (C Shells only) */usr/ucb*.

You can change the default search rules in any of our UNIX Shells by setting the shell variable called PATH. For more detail, see Chapters 3 and 4.

In the AEGIS Shell, the default command search proceeds in this order:

- working directory (.), then
- personal command directory (*-com*), and
- AEGIS command directory (*/com*).

AEGIS Shells don't recognize the PATH variable, but you can change AEGIS Shell command search rules with the shell command **csr** (command search rules). To add the directory */sys5/bin* to the AEGIS Shell's command search path, execute the following AEGIS shell built–in command:

```
$ csr -a /sys5/bin
```

Note: Since **csr** is built in to the AEGIS shell, you can't execute it from a UNIX shell.

For convenience, you may want to change the AEGIS environment search rules so that the AEGIS Shell searches the */bin* directory after it has searched the */com* directory.

2.2.2 Shell Program Execution

A shell program (shell script), is a text file that contains a series of AEGIS or UNIX commands. You can specify which shell (Bourne, C, or AEGIS) is to interpret and execute a shell program by starting the first line of each shell script with the character sequence #! followed by the pathname of the desired shell, as shown here:

#!/com/sh Specifies an AEGIS Shell script.

#!/bin/sh Specifies a Bourne Shell script. In this case, the Bourne Shell used is the one found in */SYSTYPE/bin*. If you need to be more specific, you may say:

#! /bsd4.2/bin/sh Specifies a *bsd4.2* Bourne Shell.

#! /sys5/bin/sh Specifies a *sys5* Bourne Shell.

#!/bin/csh Specifies a C Shell script.

The following example shows how this line is used in a Bourne Shell script:

```
#! /bin/sh
#
for i do
        case . . .
. . .
. . .
. . .
. . .
        esac
done
```

The shell interpreter directive #! must appear as the first line of the file in order to be interpreted correctly (remember that this information is case-sensitive), and it must comprise the first two characters of the line. Any amount of white space may appear between the exclamation point and the shell pathname.

The C Shell invokes */bin/sh* (the Bourne Shell) to interpret shell scripts when there is no explicit #! shell designation. In other shells, a script with no shell specification line is interpreted (with unpredictable results) by the shell in which it was invoked.

Furthermore, attempts to pass arguments to a subshell (e.g., one invoked to handle a shell script) fail if the invoking shell is */com/sh* or */sys5/bin/sh*. In an AEGIS or System V Bourne Shell script, the command line

```
#! /bin/sh -n
```

invokes the Bourne Shell with no arguments. The **-n** (name) option is ignored.

2.2.3 Wildcards

Every shell has its own metacharacters (wildcards). Chapters 3 and 4 detail the wildcard-handling mechanisms of the Bourne and C Shells. The *DOMAIN System Command Reference* has complete information on AEGIS Shell wildcards. The differences between the way that AEGIS Shells and UNIX Shells handle wildcards are significant. Differences even among the various UNIX shells are are important considerations. For that reason, we recommend that you use wildcarding with caution.

While all UNIX shells perform some type of wildcard expansion, the AEGIS Shell passes wildcards to commands unmodified. AEGIS commands call a handler to perform wildcard expansion, whereas UNIX commands expect a command line that has already been expanded by the shell.

As a result of this, the following precepts should govern your use of wildcards when executing a UNIX command in an AEGIS Shell, or vice-versa:

- If you're executing an AEGIS command in a UNIX shell, protect the AEGIS wildcard characters with the shell's quote mechanism. This differs from shell to shell. (See Chapters 3 and 4.)
- If you're executing a UNIX command in an AEGIS Shell, don't use wildcards.

2.2.4 Differences in Valid Pathnames

Some differences exist between the characters that are legal in an AEGIS pathname and those that are legal in a UNIX pathname. However, we perform filename mapping at the system call level with **open**(2), **creat**(2), **chdir**(2), and so on, so that you can specify pathnames containing a greater variety of characters than those allowed in the AEGIS environment.

All UNIX commands implemented by DOMAIN/IX software – as well as the AEGIS commands **cc**, **pas**, and **ftn** (the C, Pascal, and FORTRAN compilers respectively) – perform filename mapping when invoked in a UNIX shell.

2.2.5 Inprocess vs. Forked Execution

Normally, AEGIS and Bourne Shells run a command in their own process rather than by forking a child process. The shells run a command in a separate process only if the command is part of a pipeline or if it is explicitly directed to run in the background. To support job control in the C Shell, we include a shell variable that determines the process model used by that shell.

This variable, called INPROCESS, controls whether or not the C Shell runs a command as a forked child or as part of the shell process itself. Its default value is "unset", meaning that the C Shell always forks a new process to run a new command. When the C Shell is unset, certain limitations apply. See Chapter 4 of this manual for more information.

2.2.6 Changes in Working Directory

The AEGIS **wd** (working directory) command is ineffective in any UNIX Shell, that is, it does not set the shell's working directory, but only sets its own. If a program uses **wd** to change the current working directory, the shell returns to the original working directory after it executes the command.

Chapter 3

Using the Bourne Shell

3.1 Introduction

The Bourne Shell (named for its inventor, S. R. Bourne) is a language that provides a programmable interface to to the DOMAIN/IX system. Its features include control–flow primitives, parameter passing, variables, and string substitution. Constructs such as

- case
- if–then–else, and
- for

are supported, as is two–way communication between the shell and commands. String–valued parameters, typically filenames or flags, may be passed to a command. In addition, commands set a return code that may be used to determine control–flow. The standard output from a command may also serve as shell input.

The shell can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through "pipes" can be invoked. Commands are found by searching directories in the file system in a user–defined sequence. Commands can be read either from the keyboard, or from a file, which allows command procedures to be stored for later use.

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. The first part of this chapter covers most of the everyday requirements of shell users. Later sections describe those features of the shell primarily intended for use within shell procedures, including control–flow primitives and string–valued variables provided by the shell. Knowing another programming

language might help you understand this section better. The last section describes the more advanced features of the shell.

For the sake of simplicity, we use the System V Bourne Shell in our examples (note the "#" prompt), although these examples will also work using the BSD4.2 version of the Bourne Shell.

3.1.1 Special Key Definitions

The keys on DOMAIN node keyboards are bound to the functions they execute (see Chapter 1). If you have not done so already, you should now invoke one of the *sys5* key definitions files. These files bind various keys to Bourne Shell functions. To invoke these definitions, press <CMD> and enter the following DM command

```
Command: cmdf /sys/dm/file <RETURN>
```

where *file* is either

- *sys5_keys2* if you have a Low-Profile Model I
- *sys5_keys3* if you have a Low-Profile Model II keyboard
- *sys5_keys* if you have an 880 (high-profile) keyboard

Key definitions unique to *sys5_keys* (over those provided in *unix_keys*) are as follows:

```
# part of /sys/dm/sys5_keys
# ^d is mapped to eef
kd ^d eef ke
# ^\ is mapped to quit
kd '^\' dq ke
```

The key definitions files for the Low-Profile keyboards also include these entries, along with one other:

```
# del is mapped to interrupt
kd del dq -i ke
```

3.1.2 Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
# who <RETURN>
```

is a command that prints the names of everybody currently logged in to a node in the network. The command

```
# ls -l <RETURN>
```

prints a list of files in the current directory. The -l argument tells ls to print status information, size, and the creation date of each file.

3.1.3 Background Commands

When the Bourne Shell executes a command, it normally runs it from within the shell process, waits for it to finish, then prompts for more input. You may also have the shell run a command and accept additional input before the command finishes. Thus,

```
# cc pgm.c&  <RETURN>
```

calls the C compiler to compile the file *pgm.c*. The trailing ampersand (&) is an operator that instructs the shell not to wait for the command to finish. To help you keep track of such a process, the shell reports its process number following its creation. Use the **ps**(1) command to get a list of currently active processes.

3.1.4 Input/Output Redirection

Most commands produce output on the standard output (normally, the screen).

Note: In our documentation, we use "terminal" interchangeably with "node," (or, usually, "the node's keyboard"). The term "screen" refers to the transcript pad of the window in which the Bourne Shell is running.

This output may be redirected to a file by writing, for example,

```
# ls -l >file  <RETURN>
```

The shell interprets the notation *>file* and does not pass it as an argument to **ls**. If *file* doesn't exist, the shell creates it; otherwise, the original contents of *file* are replaced with the output from **ls**. You may also append output to a file by using this notation:

```
# ls -l >>file  <RETURN>
```

Here too, *file* is created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
# wc <file  <RETURN>
```

The command **wc**(1) reads its standard input (in this case, redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then this could be used:

```
# wc -l <file  <RETURN>
```

3.1.5 Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, a vertical line (|), as in

```
# ls -l | wc <RETURN>
```

Two commands connected in this way constitute a "pipeline" and the overall effect is the same as

```
# ls -l >file; wc <file <RETURN>
```

except that no *file* is used. Instead, the two processes are connected by a pipe and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

Many UNIX commands are called "filters." A filter is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1), selects from its input those lines that contain some specified string. Thus,

 # ls | grep old <RETURN>

prints those lines, if any, of the output from **ls** that contain the string *old*. Another useful filter is **sort**(1), which can be used, for example, to print an alphabetically sorted list of logged–in users as shown here:

 # who | sort <RETURN>

A pipeline may consist of more than two commands. For example,

 # ls | grep old | wc –l <RETURN>

prints the number of filenames in the current directory containing the string *old*.

3.1.6 Generating Filenames

Many commands accept filenames as arguments. For example, this prints information relating to the file *main.c*:

 # ls –l main.c <RETURN>

The shell provides a means of generating a list of filenames that match a pattern; e.g.,

 # ls –l *.c <RETURN>

generates, as arguments to **ls**, all filenames in the current directory that end in *.c*. In this context, the asterisk is a metacharacter "pattern" that matches any string including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters.

A pair of characters separated by a dash (–) matches any character lexically between the pair. For example, consider the following:

[a–z]*	Matches all names in the current directory beginning with one of the letters a through z.
/usr/fred/test/?	Matches all one–character names in the directory */usr/fred/test*. If no filename matches the pattern, then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

 # echo /usr/fred/*/*.bin <RETURN>

finds and prints the names of all files of the form *filename.bin* in sub–directories of */usr/fred*. The **echo**(1) command simply prints its arguments, separated by blanks. Using this last feature can be expensive, requiring, in this case, a scan of all sub–directories of */usr/fred*.

There is one exception to the general rules given for patterns.The period (.) at the start of a filename must be explicitly matched. Therefore,

```
# echo * <RETURN>
```

echoes all filenames in the current directory not beginning with a period. This echoes all those filenames beginning with a period:

```
# echo .* <RETURN>
```

It avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory," respectively. (Notice that **ls** suppresses listing of information for the files "." and "..".)

Note: AEGIS commands perform their own wildcard expansion, with rules differing from those used by the Bourne Shell. Unquoted wildcards used in the Bourne Shell are expanded according to the Bourne Shell's rules, then passed to the command being executed. When executing an AEGIS command from a Bourne Shell, you may need to protect certain shell metacharacters with quotes so that they are passed unmodified to the AEGIS command.

3.1.7 Quotation

As we have mentioned, characters that have a special meaning to the shell are called metacharacters. A complete list of Bourne Shell metacharacters appears at the end of this chapter, but some of the more common ones are shown in Table 3–1 below.

Table 3–1. Some Common Bourne Shell Metacharacters

<	Redirects input
>	Redirects output
*	Matches any set of characters
?	Matches any single character
&	Designates a background command
|	Designates a pipe

A character preceded by a backslash (\) is said to be "quoted" and loses any special meaning it may otherwise have had. Since the backslash is elided, **echo**, used as shown, returns the following strings

```
# echo \? <RETURN>
?
# echo \\ <RETURN>
\
```

To allow long strings to be continued over more than one line, the shell ignores the sequence *newline*. The backslash is convenient for quoting single characters. When more than one character needs quoting, we recommend the easier method of enclosing the string between single quotes. For example,

```
# echo xx'****'xx <RETURN>
xx****xx
```

The quoted string may not contain the single quote character ('), but it may contain newlines, which are preserved. We recommend this simple quoting for casual use. A third quoting mechanism, which uses double quotes to prevent interpretation of some but not all metacharacters, is discussed in a later section.

3.1.8 Prompting

The shell issues a prompt when it is ready for more input. The default *sys5* Bourne Shell prompt is a pound sign (#) followed by a space. The default *bsd4.2* Bourne Shell prompt is B$ followed by a space. Either prompt may be changed. For example, to set the prompt to the string *yesdear*, type this:

```
# PS1=yesdear <RETURN>
```

If a newline is typed and further input is needed, the shell issues the secondary prompt, a greater–than symbol (>) followed by a space. If this happens unexpectedly, type an interrupt to return the main shell prompt. You may also change this prompt.

For example,

```
# PS2=nodear <RETURN>
```

sets the prompt to the string *nodear*.

3.2 Starting the Bourne Shell

When you log in to a DOMAIN node, the DM (Display Manager) looks in several places for information about what windows to open and what processes to start (see *Getting Started With Your DOMAIN/IX System* and the *DOMAIN System User's Guide* for more detailed information). It normally opens an AEGIS Shell, then looks for the file

your_home_directory/user_data/startup_dm.*display_type*

where *display_type* matches the type of display in use (e.g., 19L or color). If you include a command line such as

```
(0,200)dr; (540,600)cp /sys5/bin/start_sh -n bourne_shell
```

in your *startup_dm* file, the DM automatically opens a *sys5* Bourne Shell when you log in. Since we included the **-n** option, the process is named "bourne_shell."

Note: In the example line above, we specified

```
/sys5/bin
```

as the */bin* to use. See Chapter 1's information on multiple version support for further details.

You may also define a key or function key to open a Bourne Shell. This DM command defines the shifted L5 key (labeled <SHELL>) so that pressing <SHIFT> <SHELL> opens a Bourne Shell:

```
kd l5s cp /bin/start_sh ke
```

Here, since no */bin* is specified, */$(SYSTYPE)/bin* supplies the **start_sh**(1) command.

When you log in, the shell sets the working directory to your home directory and begins reading commands from the file named *.profile* in this directory. The shell assumes that any file called *.profile* in your home directory contains commands, and thus reads it first, before reading commands from the terminal or any other file. Every Bourne Shell you start reads from this file.

Note: If you use the DM editor to create your *.profile*, you should then use **chown**(1) to make yourself the owner of your *.profile* and ensure that the file is read. (Use the System V version of **chown**, since you must be super–user in order to use the BSD4.2 version.)

3.3 Shell Procedures

The shell may be used to read and execute commands contained in a file, e.g.,

```
# sh file [argument(s)] <RETURN>
```

calls the shell to read commands from *file*. Such a file is called a shell procedure or shell script. Arguments may be supplied with the call and are referred to in *file* using the positional parameters $1, $2, ... $9. For example, if the file *wg* contains

```
who | grep $1
```

then

```
# sh wg fred  <RETURN>
```

is equivalent to

```
# who | grep fred  <RETURN>
```

Files have three independent attributes: read, write, and execute. Use the UNIX command **chmod**(1) to make a file executable. For example,

```
# chmod +x wg  <RETURN>
```

ensures that the file *wg* has execute status. Following this, the command

```
# wg fred  <RETURN>
```

is equivalent to

```
# sh wg fred  <RETURN>
```

This allows shell procedures and programs to be used interchangeably. Besides providing names for positional parameters, the number of positional parameters in the call is available as $#. The name of the file being executed is available as $0.

A special shell parameter $* is used to substitute for all positional parameters except $0. Typically, this is used to provide some default arguments, as in the following, which simply prepends some arguments to those already given:

```
# nroff -T450 -cm $* <RETURN>
```

3.3.1 Control Flow Using "for"

Shell procedures are frequently used to loop through the arguments ($1, $2, ...) executing commands once for each argument. For example, consider the following program that searches a file of corporate phone numbers containing lines of the form

```
tony 8756
bob 9934
sherry 4368
...
richard 5335
```

If this file is called */usr/lib/telnos*, then the text of the shell procedure *tel* is

```
#! /bin/sh
for i
do grep $i /usr/lib/telnos; done
```

This command line prints those lines in */usr/lib/telnos* that contain the string *sherry*

```
# tel sherry <RETURN>
```

while this prints those lines containing *sherry* followed by those for *richard*:

```
# tel sherry richard <RETURN>
```

The **for** loop notation is recognized by the shell and has the general form

for *name* **in** *w1 w2* **...**
do *command–list*
done

A *command–list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, the shell only recognizes reserved words like **do** and **done** when they follow a newline or semicolon. The shell variable *name* is set to the words *w1 w2* ... in turn each time the *command–list* following **do** is executed. If **in** *w1 w2* ... is omitted, then the loop is executed once for each positional parameter; that is, **in** $* is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
#! /bin/sh
for i do >$i; done
```

The command line

```
# create alpha beta <RETURN>
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

3.3.2 Control Flow Using "case"

The Bourne Shell's **case** statement provides a multiway branching mechanism, e.g.,

```
#! /bin/sh
case $# in
        1) cat >>$1 ;;
        2) cat >>$2 <$1 ;;
        *) echo 'usage: append [ from ] to' ;;
esac
```

is an **append** command. When called with one argument as in

append *file*

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command. When called with two arguments as in

append *file1 file2*

the contents of *file1* are appended to *file2*. If the number of arguments supplied to **append** is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

case *word* **in**
pattern) *command–list*;;
...
esac

The shell attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, the associated *command–list* is executed and execution of the **case** is complete. Since a single asterisk (*) is the pattern that matches any string, it can be used for the default case.

Note: The shell doesn't check to see that only one pattern matches the **case** argument. The first match found by the shell defines the set of commands to be executed.

In this example, the commands following the second asterisk (*) are never executed.

```
#! /bin/sh
case $# in
        *) ... ;;
        *) ... ;;
esac
```

The **case** construction may also be used to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command:

```
#! /bin/sh
for i
do case $i in
        -[ocs]) ... ;;
        -*) echo 'unknown flag $i' ;;
        *.c) /lib/c0 $i ... ;;
        *) echo 'unexpected argument $i' ;
        esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a pipe character (|). Thus,

```
case $i in
        -x|-y) ...
esac
```

is equivalent to

```
case $i in
        -[xy]) ...
esac
```

The usual quoting conventions apply, so that

```
case $i in
        \?) ...
```

matches a question mark (?).

3.3.3 Here Documents

The shell procedure *tel,* illustrated previously, uses the file */usr/lib/telnos* to supply the data for **grep**(1). Alternatively, this data may be included within the shell procedure as a "here document." For example,

```
#! /bin/sh
for i
do grep $i <<!
        ...
        richard 5335
        sherry 4368
        ...
!
done
```

In this case, the shell takes the lines between <<! and ! as the standard input for **grep**. The exclamation point (!) is arbitrary. The here document is terminated by a line that consists of the character (or string) following the lesser-than characters (<<).

Parameters are substituted in the document before it is made available to **grep** as illustrated by the following procedure called *edg*.

```
#! /bin/sh
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
# edg string1 string2 file <RETURN>
```

is then equivalent to the **ed** commands

```
ed file <<%              <RETURN>
g/string1/s//string2/g   <RETURN>
w                        <RETURN>
%                        <RETURN>
```

and changes all occurrences of *string1* in *file* to *string2*. To prevent substitution, use a backslash (\) to quote the special dollar sign character ($) as in

```
# ed $3 <<+      <RETURN>
1,\$s/$1/$2/g    <RETURN>
w                <RETURN>
+                <RETURN>
```

This version of *edg* is equivalent to the first except that **ed**(1) prints a question mark if no occurrences of the string $1 appear. You can entirely prevent substitution within a here document by quoting the terminating string. For example,

```
grep $i <<\#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a here document, this latter form is more efficient.

3.3.4 Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores.

Note: Use the **set** command to examine all variables that are currently set.

Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

assigns values to the variables *user*, *box*, and *acct*. A variable may be set to the null string. The following line sets the variable *null* to the null string:

```
null=
```

The value of a variable is substituted by preceding its name with a dollar sign ($). For example, the following line echoes *fred*:

```
# echo $user
```

Variables may be used interactively to provide abbreviations for frequently used strings. For example, this moves the file *pgm* from the current directory to the directory */usr/fred/bin*:

```
# b=/usr/fred/bin
# mv pgm $b
```

A more general notation is available for parameter (or variable) substitution, as in

```
# echo ${user}
```

which is equivalent to

```
# echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
# tmp=/tmp/ps
# ps a >${tmp}a
```

directs the output of **ps**(1) to the file */tmp/psa*, whereas this causes the value of the variable **tmpa** to be substituted:

```
# ps a >$tmpa
```

Except for **$?**, which is set after every command, the Bourne Shell sets these variables when invoked:

$? The exit status (decimal string return code) of the most–recently–executed command. Most commands return a zero if they execute successfully, and a non–zero status otherwise. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). This is used, for example, in the **append** command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example,

```
# ps a >/tmp/ps$$
...
# rm /tmp/ps$$
```

$! The decimal process number of the last process run in the background.

$– The current shell flags, such as **–x** and **–v.**

Some variables have special meaning to the shell. Avoid using them elsewhere.

Note: Those shell variables unique to the *sys5* Bourne Shell are flagged with the indicator [*sys5*]. The *bsd4.2* version of the Bourne Shell doesn't recognize these.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since last examined, the shell prints the message "you have mail" before prompting for the next command. This variable is typically set in the file *.profile* in your home directory, e.g.,

```
$MAIL=/usr/mail/fred
```

$MAILCHECK Specifies how often (in seconds) the shell checks for mail. The default value is 600 seconds. If $MAILCHECK is set to 0, the shell checks before each prompt. [*sys5*]

$MAILPATH A colon–separated list of filenames. If this parameter is set, the shell announces the arrival of mail in any of the specified files. Each filename can be followed by a percent sign (%) and a message printed when the modification time changes. The default message is "you have mail". [*sys5*]

$CDPATH Specifies the search path for the **cd** command. [*sys5*]

$HOME The default argument for the **cd** command. The current directory is used to resolve filename references not beginning with a slash (/), and is changed using the **cd** command. For example,

```
# cd /usr/fred/bin <RETURN>
```

makes the current directory */usr/fred/bin*. The command **cd** with no argument is equivalent to

```
# cd $HOME <RETURN>
```

This variable is also typically set in the the user's *.profile*.

$PATH A list of directories that contain commands. Each time a command is executed by the shell, a list of directories is searched for an executable file. If the **$PATH** variable isn't set, the current directory, */$(SYSTYPE)/bin*, and */$(SYSTYPE)/usr/bin* are searched by default. Otherwise, **$PATH** consists of directory names separated by colons (:). For example,

```
# PATH=:/usr/fred/bin:/bin:/usr/bin <RETURN>
```

specifies that the current directory (the null string before the first colon), */usr/fred/bin*, */bin* and */usr/bin* are to be searched in that order.

Thus, individual users can have "private" commands that are accessible independently of the current directory. If the command name contains a slash (/), this directory search is not used. The shell makes a single attempt to execute the command.

$PS1 The primary shell prompt string; by default, a pound sign (#).

$PS2 The shell prompt when further input is needed; by default, a greater–than character (>).

$IFS The set of characters used by blank interpretation.

3.3.5 The "test" Command

The **test** command has a number of uses in shell programs. For example,

```
# test -f name <RETURN>
```

returns zero exit status if *name* exists and non–zero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given here. See **test**(1) for a complete specification.

test *s* true if *s* is non–null

test -f *name* true if *name* is a file that exists

test -r *name* true if *name* is a readable file

test -w *name* true if *name* is a writable file

test -d *name* true if *name* is a directory that exists

test -l *name* true if *name* is a soft link

Note: In determining whether an object is a soft link, **test -d** *name* also returns true if *name* is a soft link that points to a directory. Furthermore, **test -f** *name* returns true if *name* is a soft link that points to a file. If *name* is a soft link that points to a non-existent object, then **test -f** *name* returns false while **test -l** *name* returns true.

3.3.6 Control Flow Using "while"

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if-then-else** branch are also provided. The actions of **while, until,** and **if-then-else** are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*
do *command-list*
done

The value tested by the **while** command is the exit status of the last simple command following **while.** Each time around the loop, *command-list* is executed. If a zero exit status is returned, then *command-list* is executed; otherwise, the loop terminates. Thus,

```
#! /bin/sh
while test $1
do ...
        shift
done
```

is equivalent to

```
#! /bin/sh
for i
do ...
done
```

Shift is a shell command that renames the positional parameters $2, $3, ... as $1, $2, ... and loses $1.

You can also use the **while/until** loop to make the shell wait until an external event occurs, before running commands. An **until** loop reverses the termination condition. For example, this does a loop every five minutes until *file* exists (presumably another process creates the file):

```
#! /bin/sh
until test -f file
do sleep 300; done
commands
```

3.3.7 Control Flow Using "if"

The Bourne Shell also provides a general conditional branch of the form

```
if command–list
then command–list
else command–list
fi
```

that tests the value returned by the last simple command following **if**. The **if** command may be used along with the **test** command to test for the existence of a file as in

```
if test –f file
then process file
else do something else
fi
```

A multiple test **if** command of the form

```
if ...
then ...
else if ...
then ...
else if ...
...
fi
fi
fi
```

may be written using an extension of the **if** notation as

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following shows the **touch**(1) command, which changes the "last modified" time for a list of files. The command may be used along with **make**(1) to force recompilation of a list of files.

```
#! /bin/sh
flag=
for i
do case $i in
        -c) flag=N ;;
        *) if test -f $i
        then ln $i junk$$; rm junk$$
        elif test $flag
        then echo file \'$i\' does not exist
        else >$i
        fi
        esac
done
```

The **-c** flag in this command forces subsequent files to be created if they don't already exist. Otherwise, an error message would be printed. The shell variable **flag** is set to a non-null string if the **-c** argument is found. These commands make a link to the file and then remove it, thus causing the last modified date to be updated:

```
ln ...; rm ...
```

The sequence

```
if command1
then command2
fi
```

may be written as

command1 && *command2*

Conversely,

command1 || *command2*

executes *command2* only if *command1* fails. In each case, the value returned is that of the last simple command executed.

3.3.8 Command Grouping

Commands may be grouped in one of the following two ways:

{ *command-list* ; }
(*command-list*)

In the first example, *command-list* is simply executed; the second executes *command-list* as a separate process. For example, this command line executes **rm junk** in the directory *x* without changing the current directory of the invoking shell:

(cd x; rm junk) <RETURN>

The commands

cd x; rm junk <RETURN>

have the same effect, but they leave the invoking shell in the directory *x*.

3.3.9 Debugging Shell Procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
# set -v <RETURN>
```

(**-v** for verbose), causing lines of the procedure to be printed as they are read. This helps isolate syntax errors. Invoke it without modifying the procedure by specifying

```
# sh -v proc <RETURN>
```

where *proc* is the name of the shell procedure. This flag may be used along with the **-n** flag, which prevents execution of subsequent commands.

Note: Using **set -n** at a terminal renders the terminal useless until you type an end-of-file (EOF).

The command

```
# set -x <RETURN>
```

produces an execution trace. Following parameter substitution, each command is printed as it is executed. Both flags may be turned off by typing

```
# set - <RETURN>
```

and the current setting of the shell flags is available as

```
$-
```

3.4 Keyword Parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell isn't affected. For example, this executes *command* with **user** set to *fred*:

```
user=fred command
```

The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as the $1, $2, ... positional parameters.

You may also use the **set** command to set positional parameters from within a procedure. For example,

```
# set - *
```

sets $1 to the first filename in the current directory, $2 to the next, and so on. The dash (-) ensures correct treatment when the first filename begins with a dash.

3.4.1 Parameter Transmission

When a shell procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. Thus,

```
# export user box <RETURN>
```

marks the variables **user** and **box** for export. When a shell procedure is invoked, all exportable variables are copied for use within the invoked procedure. Modification of such variables within the procedure doesn't affect the values in the invoking shell. A shell procedure may not usually modify the state of its caller without an explicit request on the part of the caller. Shared file descriptors are an exception to this rule.

Note: Any new process created that takes its context from the process in which a variable was defined and exported will recognize the new variable. Processes already created (or those created later) that don't take their context from the process where the variable was defined won't apply the variable.

Names whose values are intended to remain constant may be declared **readonly**. The form of this command is the same as that of the **export** command:

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.4.2 Parameter Substitution (*bsd4.2*)

In the *bsd4.2* version of **/bin/sh**, the null string replaces any unset shell parameter. For example, if the variable **d** is not set,

```
B$ echo $d <RETURN>
```

or

```
B$ echo ${d} <RETURN>
```

echoes nothing. A default string may be given as in

```
B$ echo ${d-.} <RETURN>
```

which echoes the value of the variable **d** if it is set and a period (.) otherwise. The default string is evaluated using the usual quoting conventions so that

```
B$ echo ${d-'*'} <RETURN>
```

echoes an asterisk (*) if the variable **d** is not set. Similarly,

```
B$ echo ${d-$1} <RETURN>
```

echoes the value of **d** if it is set and the value (if any) of $1 otherwise. A variable may be assigned a default value using the notation

```
B$ echo ${d=.} <RETURN>
```

which substitutes the same string as

```
B$ echo ${d-.} <RETURN>
```

and if **d** were not previously set then it is set to the string ".". (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default, then the notation

```
B$ echo ${d?message} <RETURN>
```

echoes the value of the variable **d** if it has one; otherwise, the shell prints *message* and abandons the shell procedure. If *message* is absent, then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows:

```
#! /bin/sh
: ${user?} ${acct?} ${bin?}
...
```

The colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct**, or **bin** are not set, the shell abandons execution of the procedure.

3.4.3 Parameter Substitution *(sys5)*

The *sys5* version of */bin/sh* uses two types of parameters: positional and keyword. If *parameter* is a digit, it is a positional parameter. Use the **set** command to assign a value to a positional parameter. Keyword parameters (also called variables) may be assigned values as follows:

name = *value* [*name* = *value*] ...

No pattern–matching is performed on *value*. A function and a variable can't have the same name.

${parameter} Substitute the value, if any, of the parameter. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If parameter is * or @, all the positional parameters, starting with $1, are substituted (separated by spaces). Parameter $0 is set from argument zero when the shell is invoked.

${parameter:-word} If *parameter* is set and is non–null, substitute its value; otherwise, substitute *word*.

${parameter:=word} If *parameter* is not set or is null, set it to *word*; substitute the value of the *parameter*. Positional parameters can't be assigned this way.

${parameter:?word} If *parameter* is set and is non–null, substitute its value; otherwise, print *word* and exit from the shell. If word is omitted, print the message "parameter null or not set".

${parameter:+word} If *parameter* is set and is non–null, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

```
# echo ${d:-'pwd'} <RETURN>
```

If you omit the colon from the above expressions, the shell only checks to see whether or not *parameter* is set.

The *sys5* Bourne Shell automatically sets the following parameters:

The number of positional parameters in decimal.

- Flags supplied to the shell on invocation or by the **set** command.

? The decimal value returned by the last synchronously executed command.

$ The process number of this shell.

! The process number of the last command invoked in background.

3.4.4 Command Substitution

The standard output from a command can be substituted in a way similar to that allowed for parameters. The command **pwd** prints on its standard output the name of the current directory. For example, if the current directory is *usr/fred/bin*, then the command

```
# d='pwd' <RETURN>
```

is equivalent to

```
# d=/usr/fred/bin <RETURN>
```

The shell takes the entire string between opening single quotes (grave accents,'...') as the command to be executed and replaces it with the output from the command. The command is written using the usual quoting conventions except that a grave accent (') must be escaped with a backslash (\). For example,

```
# ls 'echo "$1"' <RETURN>
```

is equivalent to

```
# ls $1 <RETURN>
```

Command substitution occurs in all contexts where parameter substitution occurs (including here documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is **basename**(1), which removes a specified suffix from a string. For example,

```
# basename main.c .c <RETURN>
```

prints the string *main*. Its use is illustrated by the following fragment from a **cc**(1) command that sets **B** to the part of **$A** with the suffix **.c** stripped:

```
case $A in
        ...
        *.c) B='basename $A .c'
        ...
esac
```

Here are some composite examples:

for i in 'ls –t'; do ...	Sets the variable **i** to the names of files in time order, most recent first.
set 'date'; echo $6 $2 $3, $4	Prints the date. For example, `1984 Dec 14, 23:59:59`

3.4.5 Evaluation and Quoting

The shell is a macro processor that provides parameter substitution, command substitution and filename generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in the summary of Bourne Shell grammar in the next section. Before a command is executed, the following substitutions occur:

- parameter substitution (e.g., **$user**).
- command substitution (e.g., **`pwd`**). Only one evaluation occurs, so that if, for example, the value of the variable *X* is the string *$y*, then the following echoes *$y*:

  # **echo $X**

Following these substitutions, the resulting characters are broken into non–blank words. Thus, "blanks" are the characters of the string **$IFS**. By default, this string consists of blank, tab and newline. The null string isn't regarded as a word unless quoted, e.g.,

echo '' <RETURN>

passes on the null string as the first argument to **echo**, whereas

echo $null <RETURN>

calls **echo** with no arguments if the variable **null** is not set or set to the null string.

Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of filenames is generated to replace the word. Each such filename is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

In addition to the quoting mechanisms described earlier using backslash (\) and the '...' string, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitutions occur but filename generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using a backslash (\):

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters **$ ` " **

For example, this passes the value of the variable **x** as a single argument to **echo**:

echo "$x" <RETURN>

Similarly,

echo "$*" <RETURN>

passes the positional parameters as a single argument and is equivalent to

```
# echo "$1 $2 ..." <RETURN>
```

The notation $@ is the same as $* except when it is quoted.

```
# echo "$@" <RETURN>
```

passes the positional parameters, unevaluated, to **echo** and is equivalent to

```
# echo "$1" "$2" ... <RETURN>
```

Table 3–2 below gives, for each quoting mechanism, the shell metacharacters that are evaluated. In this table,

- t indicates a sequence used as a terminator,
- y indicates a sequence in which the metacharacter is interpreted,
- n indicates a sequence in which the metacharacter is not interpreted.

Table 3–2. Evaluation of Bourne Shell Metacharacters by Quoting Mechanisms

Quote	Metacharacter					
'	\	$	*	`	"	'
`	n	n	n	n	n	t
"	y	y	n	y	t	n

Among other things, this table shows that the sequence \$ is not interpreted (is passed as a literal $), the sequence \' can be used to terminate a string, and the sequence "$ preserves the meta–meaning of the dollar sign ($). Where more than one evaluation of a string is required, the built–in command **eval** may be used. For example, if the variable **X** has the value *$y*, and if **y** has the value *pqr*, then this echoes the string *pqr*:

```
# eval echo $X <RETURN>
```

In general, **eval** evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. Thus,

```
# wg='eval who|grep' <RETURN>
# $wg fred           <RETURN>
```

is equivalent to

```
# who|grep fred <RETURN>
```

Here, **eval** is required since there is no interpretation of metacharacters, such as a pipe character (|), following substitution.

3.4.6 Error Handling

How errors detected by the shell are treated depends on the type of error and whether the shell is being used interactively. An interactive shell is one whose input and output

are connected to a terminal as determined by **gtty**(2). A shell invoked with the **-i** flag is also interactive.

Execution of a command may fail for any of the following reasons:

- Input/output redirection won't work (e.g., a file doesn't exist or can't be created).
- The command itself doesn't exist or cannot be executed.
- The command terminates abnormally.
- The command terminates normally but returns a non-zero exit status.

In every case, the shell goes on to execute the next command. Except in the last case, the shell prints an error message. All remaining errors cause the shell to exit from a command procedure. An interactive shell returns to read another command from the terminal. Such errors include the following:

- Syntax errors (e.g., if ... then ... done).
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**.

The shell flag **-e** causes the shell to terminate if any error is detected.

Many of the UNIX signals used by DOMAIN/IX software are described in Table 3-3. The signals in this list of potential interest to shell programs are 1, 2, 3, 14, and 15. For a complete list, see **signal**(2) or **signal**(3C).

Table 3-3. UNIX Signals Commonly Used by DOMAIN/IX Software

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from **kill**(1))
19	DOMAIN system fault with no UNIX equivalent

3.4.7 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The **trap** command is required for necessary clean-up activity (e.g., removal of temporary files). For example, this line sets a trap for signal 2 (terminal interrupt):

```
trap 'rm /tmp/ps$$; exit' 2
```

If this signal is received, it executes the commands

```
rm /tmp/ps$$; exit
```

Exit is another built–in command that terminates execution of a shell procedure. It is required to keep the shell from resuming execution of the procedure at the place where it was interrupted, once the trap has been taken.

UNIX signals can be ignored (never sent to the process); they can be caught, allowing the process to decide what action to take; or they can be left to cause process termination with no further action. If a signal is ignored on entry to a shell procedure, for example, by being invoked in the background, then **trap** commands (and the signal) are ignored. The following modified version of **touch**(1) shows the use of **trap** in removing the *junk$$* file:

```
#! /bin/sh
flag=
trap 'rm –f junk$$; exit' 1 2 3 15
for i
do case $i in
   –c) flag=N ;;
   *) if test –f $i
    then ln $i junk$$; rm junk$$
    elif test $flag
    then echo file \'$i\' does not exist
    else >$i
    fi
    esac
done
```

The **trap** command appears before the creation of the temporary file; otherwise, the process could die without removing the file. Since there is no signal 0, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment, taken from the **nohup** command,

```
trap '' 1 2 3 15
```

causes *hangup, interrupt, quit*, and *kill* to be ignored by the procedure and by invoked commands. Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The shell procedure called **scan** (below) illustrates **trap** usage where there is no exit in the **trap** command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands, but cause termination when **scan** is waiting for input.

```
#! /bin/sh
d='pwd'
for i in *
do if test -d $d/$i
   then cd $d/$i
   while echo "$i:"
        trap exit 2
        read x
        do trap : 2; eval $x; done
   fi
done
```

Read x is a built–in command that reads one line from the standard input and places the result in the variable **x**. The command returns a non–zero exit status if an end–of–file is read or an interrupt is received.

3.4.8 Command Execution

To run a command other than a built–in, the shell first creates a new program level in the shell process. The execution environment for the command includes input, output, and the states of signals, and is established before the command is executed. A built–in command **exec** creates a new program level in the shell process. For example, a simple version of the **nohup** command looks like this:

```
trap '' 1 2 3 15
exec $*
```

Trap turns off the signals specified so they are ignored by subsequently created commands; **exec** runs the specified command as a new program level in the shell process.

Most forms of input/output redirection have already been described. In the following examples, *word* is only subject to parameter and command substitution. No filename generation or blank interpretation takes place; thus, for example,

```
echo ... >*.c
```

writes its output into a file whose name is *.c. Input output specifications are evaluated left to right as they appear in the command.

> *file* — The standard output (file descriptor 1) is sent to *file*, which is created if it doesn't already exist.

>> *file* — The standard output is sent to *file*. If the file exists, output is appended (by seeking to the end); otherwise, the file is created.

< *file* — The standard input (file descriptor 0) is taken from the *file*.

<< *file* — The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *file*. If *file* is quoted, no interpretation of the document occurs. If *file* isn't quoted, parameter and command substitution occur and a backslash (\) is used to quote the characters \ $ ‘ and the first character of *word*. In the latter case, *newline* is ignored (c.f. quoted strings).

>& *digit* The file descriptor *digit* is duplicated using the system call **dup**(2), and the result is used as the standard output.

<& *digit* The standard input is duplicated from file descriptor *digit*.

<&- The standard input is closed.

>&- The standard output is closed.

If any of the above are preceded by a digit, the file descriptor created is that specified by the digit instead of the default 0 or 1. For example, this runs *command* with message output (file descriptor 2) redirected to *file*:

command ... 2>file

and this runs *command* with its standard output and message output merged:

command ... 2>&1

File descriptor 2 is created by duplicating file descriptor 1, but usually results in a merge of the two streams. The environment for a command run in the background such as

 # list *.c | lpr & <RETURN>

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two parallel processes (the shell and the command) from trying to read the same input (a rather chaotic situation). For example,

 # ed file & <RETURN>

allows both the editor and the shell to read from the same input at the same time. The environment of a background command is further modified by turning off the quit and interrupt signals so that they are ignored by the command. Thus, by convention, a signal set to 1 (ignored) is never changed, even for a short time. Note also that the shell command **trap** has no effect on an ignored signal.

3.5 Summary of Bourne Shell Grammar

item: word
input-output
name = value

simple-command: item
simple-command item

command: simple-command
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* ... **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* ... **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: command
pipeline | *command*

andor: pipeline
andor **&&** *pipeline*
andor || *pipeline*

command–list: andor
command–list ;
command–list **&**
command–list ; *andor*
command–list **&** *andor*

input–output: > *file*
< *file*
>> *word*
<< *word*

file: word
& *digit*
& –

case–part: pattern) *command–list* ;;

pattern: word
pattern | *word*

else–part: **elif** *command–list* **then** *command–list else–part*
else *command–list*
empty

empty:

word: a sequence of non–blank characters

name: a sequence of letters, digits, or underscores starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

3.6 Summary of Shell Metacharacters & Reserved Words

3.6.1 Syntactic

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping

<	input redirection
<<	input from a here document
>	output creation
>>	output append

3.6.2 Patterns

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

3.6.3 Substitution

${...}	substitute shell variable
`...`	substitute command output

3.6.4 Quoting

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for $ ` \ "

3.6.5 Reserved Words

- **if**
- **then**
- **else**
- **elif**
- **fi**
- **case**
- **in**
- **esac**
- **for**
- **while**
- **until**
- **do**
- **done**
- **{ }**

Chapter 4

Using the C Shell

The primary purpose of any shell is to translate command lines typed at a terminal into useful work, something the shell usually accomplishes by invoking another program. The C Shell (*/bin/csh*) is one of several shells available to users of the DOMAIN/IX system. We provide the C Shell in both *bsd4.2* and *sys5* DOMAIN/IX versions.

This chapter introduces the more commonly-used features of the C Shell. The **csh**(1) documentation in the *DOMAIN/IX Command Reference* provides a full description of all features of this shell.

Note: Chapter 1 describes how to invoke shells in either the *sys5* or the *bsd4.2* environment. The process is nearly transparent to the user (requiring only that you set the SYSTYPE environment variable). However, it has implications for those developing new software and, to a somewhat lesser extent, people (and shells) running programs. Make sure that you have read Chapter 1 before you begin.

4.1 Introduction

This chapter includes several examples. We recommend that you try them all, to develop a variety of experiences with the C Shell.

4.1.1 Special Key Definitions

The DOMAIN/IX system provides special files that bind various keys to functions used by the C Shell. These key definitions files are introduced in Chapter 1. To invoke a particular set of key definitions, press <CMD> and enter this DM command:

```
Command: cmdf /sys/dm/file <RETURN>
```

where *file* is one of the following:

- *bsd4.2_keys2* (*sys5_keys2*) if you have a Low-Profile Model I keyboard
- *bsd4.2_keys3* (*sys5_keys3*) if you have a Low-Profile Model II keyboard
- *bsd4.2_keys* (*sys5_keys*) if you have an 880 (high-profile) keyboard.

Special key definitions (beyond those provided in *unix_keys*) included in these files are:

```
# part of /sys/dm/bsd4.2_keys (/sys/dm/sys5_keys)
# ^z is changed from unix_keys eef to suspend
kd ^z dq -c 120028 ke (for job control in the csh)
# ^d is mapped to eef
kd ^d eef ke
# ^\ is mapped to quit
kd '^\' dq ke
# ^j is mapped to unsuspend (if you ^z a non-csh and want to wake it);
ke ^j dq -c 12002b ke
```

In addition to the lines above, those files used for the Low-Profile keyboards include the line:

```
# ^c is changed from cut to interrupt
kd ^c dq -i ke
```

4.1.2 Starting the Shell

To start a C Shell on a DOMAIN node, log in and type the DM command

```
Command: cp /bin/start_csh <RETURN>
```

In the case of the line above, */bin* resolves to */$(SYSTYPE)/bin*, as shown in Chapter 1.

The DM opens a window and runs the C Shell in it. With the **start_csh** command, you may supply the coordinates where the DM will locate the upper left and lower right corners of the window. You may even give the process a name, as in this line:

```
Command: (0,200)dr; (540,600)cp /bin/start_csh -n c_shell <RETURN>
```

This command line opens up a small window near the left side of the screen and displays the name *c_shell* in the window legend.

4.1.3 The Basic Notion of Commands

A shell acts primarily as a medium through which you invoke other programs. While the shell has a set of built-in functions that it performs directly, most commands to the shell cause execution of programs that reside elsewhere (are not part of the shell).

A command consists of a word or words that the shell interprets as a command name followed by optional arguments. Thus, the command

```
% mail kate <RETURN>
```

consists of a command name (**mail**), followed by an argument (*kate*). The shell looks in every directory for a file named **mail**. Upon finding something called **mail**, the shell assumes it to be an executable file, and so requests that the system execute the file.

The rest of the words on the command line are assumed to be arguments and are passed to the command when it is executed. In this case, we specified the argument **kate** which **mail** interprets as the name of a user to whom mail is to be sent. In normal usage, we might invoke **mail** as follows:

```
% mail kate <RETURN>
Is there a meeting today?  And is it at 1:00?
bob
*** EOF ***
EOT
%
```

Here we typed a message to send to *kate* and ended this message with a ↑D, which sent an end-of-file (EOF) to the **Mail** program.

Mail, in turn, echoed "EOT" (end-of-transmission), transmitted the message to *kate*, and exited. The shell, noticing that **Mail** was finished, prompted for input by displaying a percent sign (%), which indicated its readiness for further orders.

This is the essential pattern of all interactions with DOMAIN/IX software via the C Shell. You type a complete command, and the shell executes it. When command execution completes, the shell prompts for a new command. If you run, for example, the **vi**(1) editor for an hour, the shell waits for you to finish editing, and then prompts you for further orders.

4.1.4 Flag Arguments

While many arguments to commands specify objects such as filenames, some arguments invoke optional capabilities of the command. By convention, such arguments begin with a dash (-). Thus, the command

```
% ls <RETURN>
```

produces a list of the files in the current working directory. The **ls**(1) command has many options, including **-s**, the size option. If you include **-s** on a **ls** command line,

```
% ls -s <RETURN>
```

ls lists the size of each file in blocks of (normally) 1024 characters. Consult the *DOMAIN/IX Command Reference* to determine available options for each command.

4.1.5 Output to Files

Commands that normally read input or write output on the screen can optionally be told to get their input from a file or to send their output to a file. Suppose you wish to save the current date in a file called *now*. This command

```
% date <RETURN>
```

prints the current date on the transcript pad of the shell into which **date**(1) was typed, because the screen (transcript pad) is the default standard output, and **date** always prints the date on the standard output.

The shell lets you redirect the standard output of a command through a notation using the greater-than (>) metacharacter and the name of the file where output is to be placed.

Thus, the command

```
% date > now  <RETURN>
```

runs the **date** command and redirects the standard output to a file called *now* rather than to the default standard output (the screen). The current date and time are written to the file *now*. No output appears on the screen. It is important to know that **date** is unaware that its output is going to a file rather than to the screen. The shell performs this redirection before the command begins executing.

The file *now* need not have existed before the **date** command above was executed; the shell would have created the file (in the current working directory) if it did not exist.

Note: If you redirect standard output into an existing file, that file is overwritten unless the shell variable **noclobber** has been set. See the discussion of **noclobber** in the next section.

4.1.6 Metacharacters in The C Shell

The C Shell uses a number of characters to perform special functions. In general, most characters that are neither letters nor digits have special meaning to the shell. Since these special characters may also be used literally, the shell provides a means of quoting that lets you strip these metacharacters of any special meaning.

Metacharacters normally have effect only when the shell is reading input. You needn't worry about placing shell metacharacters in a letter you are sending via **mail,** or when supplying text or data to some other program. Note that the shell is only reading input when it is displaying its prompt.

Note: AEGIS commands perform their own wildcard expansion, with rules that differ from those used by the C Shell. Unquoted wildcards used in the C Shell are expanded according to the C Shell's rules, then passed to the command being executed. If executing an AEGIS command from a C Shell, you may need to protect certain shell metacharacters with quotes so that they are passed unmodified to AEGIS commands.

4.1.7 Input From Files; Pipelines

The standard input of a command can be redirected so that it is taken from a file, instead of the keyboard (default standard input). This is often unnecessary, since most commands read from a file whose name is given as an argument. You could use this

```
% sort < data  <RETURN>
```

to run to run the **sort**(1) command with standard input, where the command normally reads its input, from the file *data*. But, it is easier and just as legal to type this

```
% sort data  <RETURN>
```

letting the **sort** command open the file *data* and sort it.

Note: If you merely type

```
% sort  <RETURN>
```

then the **sort** program sorts lines from its standard input, the keyboard. Since you are not redirecting the standard input, the program sorts lines as you type them on the terminal, until you type a ↑D to indicate an end-of-file.

Another useful feature of the C Shell is its ability to connect the standard output of one command to the standard input of another using a mechanism known as a pipeline. For instance, the command

```
% ls -s  <RETURN>
```

normally produces a list of the files in the current directory and lists the size of each file in blocks of 1024 characters. To help determine which of your files is largest, you may want to have the list sorted by size rather than by name. Although **ls** has no such option, you can pipe the output of **ls** to the **sort** command and use some of **sort**'s options to get a list of files sorted in size order.

The **-n** option of **sort** specifies a numeric sort rather than an alphabetic sort. Thus,

```
% ls -s | sort -n  <RETURN>
```

tells the C Shell to run the **ls** command with the **-s** option, and then pipe the resulting output to the **sort** command run with the **-n** (numeric sort) option. The output of this combination of commands is a list of files sorted by size, with the smallest file first. You could then use the **-r** reverse sort option and the **head**(1) command in combination with the previous command, as shown here:

```
% ls -s | sort -n -r | head -5  <RETURN>
```

This sequence takes a list of files sorted alphabetically, each with the size in blocks, and pipes this list to the standard input of **sort**. **Sort**, in turn, sorts the list numerically in reverse order (largest first). The sorted list is piped to the command **head** which then displays the first five lines of the list, giving you names and sizes of the five largest files in the current directory.

Commands separated by pipe (|) characters are connected together by the shell. The standard output of the command to the left of the pipe is connected to the standard input of the command to the right of the pipe. The leftmost command in a pipeline normally takes its standard input from the keyboard. The rightmost places its standard output on the screen.

4.1.8 Filenames

Many commands need the names of files as arguments. Both DOMAIN/IX and AEGIS pathnames consist of a number of components separated from each other by the slash (/). Each component except the last names a directory in which the next component resides, in effect specifying the path of directories to follow to reach the file.

Thus, the pathname */etc/systype* specifies a file in the directory *etc*, which is a subdirectory of the node's entry directory, or "slash" (/). Within this directory the file named is *systype*, a program that returns the value of the SYSTYPE environment variable. A pathname that begins with a slash is said to be an absolute pathname, since it is specified from the absolute top of the node's directory hierarchy.

Note: A node's directory hierarchy begins one level below the network root, or "double slash" (//) directory, so a truly "absolute" pathname must always begin with two slashes followed by the name of the node's entry directory as in

```
//ice/tmp
```

When the shell sees a pathname that does not begin with a slash, it assumes that it should start looking in the current working directory. When you log in, the working directory is set to your home directory. From there, you can move to (and work in) other directories by using the **cd**(1) command. Pathnames not beginning with a slash are said to be relative to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname contains no slashes, the shell assumes that the pathname is the name of a file contained in the current working directory. Absolute pathnames, by contrast, are unrelated to the working directory.

Most filenames consist of a number of alphanumeric characters and periods. While all printing characters except a slash (/) may appear in UNIX filenames, it is inconvenient to have most non-alphabetic characters in filenames, since many of them have special meaning to the shell. The period or dot (.), while not a C Shell metacharacter, is often used to separate the extension of a filename from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. Their names share a common base portion (that part of the name which is left when a trailing period and following characters that are not periods are stripped off). The file *prog.c* might be the source for a C program, the file *prog.o* the corresponding object file, the file *prog.errs* the errors resulting from a compilation of the program and the file *prog.output* the output of the program itself.

To refer to all four of these files in a command, use the notation

```
prog.*
```

The shell expands *prog.** into a list of names that begin with *prog.* before the command to which it is an argument is executed. The asterisk (*) here matches any sequence (including the empty sequence) of characters in a filename. The names that match are alphabetically sorted and placed in the argument list of the command. Thus,

```
% echo prog.* <RETURN>
```

echoes the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we list them above. The **echo**(1) command receives four words as arguments, even though only one argument is supplied to the shell. The shell generates the four words by filename expansion of the one input word.

The C Shell also expands other characters. The question mark (?) matches any single character in a filename. Thus,

```
echo ? ?? ???
```

echoes a line of filenames; first those with one-character names, then those with two-character names, and finally those with three-character names. The filenames of each

length are sorted independently (i.e., the output to the screen is a list of one-character filenames, followed by a list of two-character filenames, followed by a list of three-character filenames).

The shell also matches any single character from a sequence of characters delimited by brackets. Thus,

```
prog.[co]
```

matches both *prog.c* and *prog.o*. You can also place two characters around a dash (-) in this notation to denote a range. Thus, to **troff**(1) five chapters of a book that exists in the files *chap.1*, *chap.2* and so on, type the command line

```
% troff chap.[1-5] <RETURN>
```

which would pass the names

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

to **troff** for processing. The above notation is equivalent to

```
chap.[12345]
```

Note: If a list of argument words to a command (an argument list) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing filenames, then the shell considers this to be an error and prints the diagnostic message

```
No match.
```

and does not execute the command.

Files beginning with a period (.) are treated specially. Neither an asterisk (*), nor a question mark (?), nor square brackets ([ ]) match it. This prevents accidental matching of the filenames "." and ".." in the working directory, where they have special meaning to the system. It also prevents matching of other files such as *.cshrc* which are not normally visible in a directory listing. (We discuss *.cshrc* in a later section.)

Another filename expansion mechanism gives access to the pathname of the *home directory* of other users. Normally, this notation consists of a tilde (~) followed by another user's login name. For instance, the word **~kate** maps to the absolute pathname of user *kate*'s home directory, as shown here:

```
% cd ~kate  <RETURN>
% pwd       <RETURN>
% //ice/kate
```

A special case of this notation consists of a tilde alone. The tilde is the default home directory character. The shell expands this notation into the pathname of your home directory. For example, the command

```
% ls -a ~ <RETURN>
```

lists all the files in your home directory. Likewise, the command

```
% cp thatfile ~ <RETURN>
```

expands to

```
% cp thatfile //your_home_directory/thatfile
```

You may change this character by **set**ting the shell variable *homedirchar* to some other character. For example, to change the home directory character to a pound sign (#):

```
% set homedirchar = #  <RETURN>
```

To revert to the default, **unset** homedirchar.

Note: If you use the DM environment variable NAMECHARS (see Chapter 1) to assign DOMAIN naming server metameanings to the tilde, the naming server expands the tilde into the pathname of your home directory, followed by a slash. It does not, however, expand the input *~name* into user *name*'s home directory. To pass the tilde to the naming server (rather than the C Shell), escape it.

The shell also has a mechanism that uses left and right brace characters ({ }) for abbreviating a set of words that have common parts but can't be abbreviated by other mechanisms because they are not files (or are files that, while created by the program being invoked, do not exist yet). This mechanism is described in a later section.

4.1.9 Quotation

We have already described a number of the metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus, the command

```
% echo *  <RETURN>
```

does not echo the asterisk (*). It either echoes a sorted list of all filenames in the current working directory, or prints the message "No match" if no files exist in the working directory.

The recommended mechanism for placing a character that is neither a number, a digit, slash, period, nor dash in an argument word to a command is to enclose it in single quotes ('), as in the following example:

```
% echo '*'  <RETURN>
```

One special character, the exclamation point (!), is used by the history mechanism of the shell and cannot be escaped by the normal means of placing it within single quotes. The exclamation point and the single quote should be preceded by a single backslash (\) to escape their special meaning. Thus,

```
% echo \'\!  <RETURN>
```

prints

```
' !
```

These two mechanisms let you include any printing character in an argument to a shell command. They can be combined, as in

```
% echo \''*'  <RETURN>
```

which prints

```
' *
```

since the first backslash escaped the first single quote and the asterisk was enclosed in single quotes.

Note: The DM environment variable NAMECHARS (see Chapter 1) may be used to assign DOMAIN naming server metameanings to the tilde, grave accent, and backslash. When used in a pathname component, then, these characters are interpreted not as a literal, but according to the naming server's rules.

4.1.10 Terminating Commands

When you are executing a command and the shell is waiting for it to complete, there are several ways you can force it to stop executing. For instance, if you type

% **cat /etc/passwd** <RETURN>

the system prints a list of all users of the system. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT signal to the **cat**(1) command by typing †I.

Note: The DM command files for UNIX key definitions define †I as the UNIX interrupt key. You must execute one of these command files for this definition to be effective.

Since **cat** doesn't try to avoid or to otherwise handle this signal, the INTERRUPT terminates **cat**. The shell notices its termination and prompts you again. If you hit INTERRUPT again, the shell repeats its prompt since it is designed to effectively ignore INTERRUPT signals.

Many programs terminate when they get an end-of-file from their standard input. The **mail** program in an earlier example terminated when it received a †D (which generates an end-of-file) from the standard input. The C Shell normally terminates when it receives an end-of-file. When this happens, the messages

```
% *** EOF ***
logout
*** Pad Closed ***
```

are left on the transcript pad and the window is closed. Since this means that typing †D one too many times can accidentally log you out of a window, the shell has a mechanism for preventing this. This **ignoreeof** option is discussed in the next section.

If a command has its standard input redirected to come from a file, it normally terminates when it reaches the end of this file. If you execute

% **mail kate < prepared.text** <RETURN>

the **mail** command terminates when it sees the EOF at the end of the file *prepared.text* from which it is getting input. Another way to accomplish the same thing is to type

% **cat prepared.text | mail kate** <RETURN>

since the **cat** command then writes the text through the pipe to the standard input of the **mail** command. When the **cat** command completes, it terminates, closing down the pipeline, and the **mail** command receives an end-of-file from **cat** and terminates. You can also stop these commands by typing †I.

If you write or run programs that are not fully debugged, it may be necessary to stop them somewhat ungracefully. This can be done by typing †Q, which sends a QUIT signal. The shell displays the message

```
Quit
```

and the number (if any) of the job that quit.

Commands running in the background ignore INTERRUPT and QUIT signals. To stop them, use the **kill**(1) command (covered in a later section).

4.2 Starting, Stopping, and Modifying the C Shell

This section includes information on starting the C Shell and arranging for it to set certain variables to convenient values every time you log in.

4.2.1 Opening a C Shell When You Log In

When you log in to a DOMAIN node, the DM looks in several places for information about what windows to open and what processes to start (see *Getting Started With Your DOMAIN/IX System* and the *DOMAIN System Command Reference* for more detailed information). It normally opens an AEGIS shell, then looks for the file

your_home_directory/user_data/startup_dm.*display_type*

where *display_type* matches the type of display in use (e.g., 19l or color). If you include a command line such as this

```
(0,200)dr; (540,600) cp /bin/start_csh -n c_shell
```

in your *startup_dm* file, the DM automatically opens a C Shell when you log in.

You may also define a key or function key to open a C Shell. The following DM command defines the shifted L5 key — L5 is labeled <SHELL> — so that when you press <SHIFT> <SHELL>, a C Shell is opened:

```
kd l5s cp /bin/start_csh ke
```

Note: Since no */bin* is named, the **start_csh** command comes from */$(systype)/bin*.

4.2.2 Login and Logout Scripts

When you log in, the C Shell sets the working directory to your *home directory* and begins reading commands from a file *.cshrc* in this directory. Every C Shell started with the command **/bin/csh** reads from this file. In addition, you may create a file called *.login* in your home directory that the C Shell reads (after it reads *.cshrc*) if it is started with the **/bin/start_csh**(1) command. Neither of these files is required. If neither exists, the shell uses its own defaults.

Note: When you use the DM editor to create files, you aren't automatically named as file owner. In fact, "<none>" is listed as owner. Thus, if you create your *.cshrc* or *.login* using the DM editor, you must also make yourself the owner of these files. Otherwise, they are not read. Although normally used to change file per-

missions, executing the **chmod**(1) command on the file (taking care to not really change file permissions) also provides the DM with your name as file owner. Actually, the command that is normally used to change ownership is **chown**(8), but since its use is restricted to super–user, **chmod** is a reasonable substitute if used correctly. If you insist on using **chown** to change file ownership, you may use the *sys5* version taking care to supply the pathname **/sys5/bin/chown**, e.g.:

```
% /sys5/bin/chown user .cshrc <RETURN>
```

where *user* is the name of the user to whom you are assigning ownership.

As an example of a *.cshrc* file, consider the following listing:

```
set history=10
set prompt='% '
set path = (.%/com /usr/ucb /bin /usr/bin /com )
set noclobber
set ignoreeof
set inprocess
set homedirchar='%'
alias cd 'cd \!* ;ls'
alias lo logout
```

This file begins with a series of **set** commands that the shell interprets directly. These particular **set** commands establish the following conditions in the C Shell:

- The shell maintains a "history list" of the last 10 commands.
- The prompt is a percent sign followed by a space.
- The shell searches for a command in the following places, in this order:

  current directory (.)

  home_directory/com

  /usr/ucb

  /bin

  /usr/bin

  /com

  Note: The *sys5* version of the C Shell does not search */usr/ucb*, because it is not included in the structure of the */sys5* directory. Thus, if you're using the *sys5* C Shell, and you've included */usr/ucb* in the path set in your *.cshrc*, this subdirectory will be completely ignored by the shell.

- The variable **noclobber** is set, forcing the shell to notify you whenever you redirect output into a file that already exists.
- The variable **ignoreeof** is set. The shell does not terminate (close the window or, if you are using a terminal, log you off) when it receives an end–of–file from standard input.

- The variable **inprocess** is set, forcing in–process (rather than forked) execution of commands. (The default value of *inprocess* is **unset**.) See more on *inprocess* below.
- The variable **homedirchar** is set to make the home directory character a percent (%) rather than the default, tilde (~).

The next two commands are **alias** commands that, in effect, rename command sequences. Here, the command **cd** is aliased to change to the specified directory, then list its contents. And, since the variable **ignoreeof** is set, the string "lo" is defined as having the alias **logout**, allowing the closing up of the shell window with a minimum of typing.

Note: You may override **noclobber** if it is set by using the syntax

```
>!
```

For example, to overwrite the contents of a file named *now* with the current date, you can do so even if **noclobber** is set. The command line

```
date >! now
```

does it. The ">!" is a special metasyntax indicating that clobbering the file is allowed. Note that the space between the exclamation point the the *now* is critical here, as *!now* is an invocation of the history mechanism, and has a totally different effect.

You can set **inprocess** as a DM environment variable. In fact, we recommend that you do this if you plan to access DSEE–managed objects from the C Shell. In order to set **inprocess** in the DM, put the following line in any DM command file read before the C Shell is started (e.g., *'node_data/startup*):

```
env INPROCESS 'true'
```

If **inprocess** is set in this way, the C Shell runs as if you had "set inprocess" in your *.cshrc* file. If an "env INPROCESS" line is not found (or is not set to 'true' or 'TRUE') in a DM start–up file, the process model used by a shell is determined by the shell variable **inprocess**. The C Shell will, by default, have **inprocess** unset. Note that the C Shell does not export **inprocess** to the DM if you set it in your *.cshrc* or *.login* file(s).

4.2.3 Shell Variables

The shell maintains a number of variables. In the *.cshrc* file just shown, the variable **history** had a value of 10. In fact, each shell variable has as its value an array of zero or more strings. The **set** command assigns values to variables. **Set** has several forms, the most useful of which is

set *name=value*

Shell variables let you store values that can then be made available, via the substitution mechanism, to commands. The shell variables most commonly referenced are, however, those to which the shell itself refers. By changing the values of these variables, you can directly affect the behavior of the shell.

One of the most important variables is **path**. It contains a sequence of directory names where the shell searches for commands. If you execute the **set** command with no arguments, the shell displays the values of all variables currently set.

Note: The shell examines each directory in the specified **path** and determines what commands are contained there. Except for the current directory, which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they are not necessarily found by the shell. To use a command that has been added in this way, type

```
% rehash <RETURN>
```

This command causes the shell to recompute its internal table of command locations, so that it finds the newly added command. Since the shell has to look in the current directory for each command, placing **rehash** at the end of the path specification works equally well and reduces overhead.

Other useful built–in variables are **home**, which shows your home directory, and **cwd**, which contains your current working directory. **Ignoreeof** is one of several variables capable of having no value other than **unset** or **set**. Thus, to set this variable, type

```
set ignoreeof
```

To unset it, type this:

```
unset ignoreeof
```

The variable **noclobber** is another "boolean" variable. It can only assume two states.

Note: Any newly–created process that takes its context from the process in which a variable was defined and exported will recognize the variable. Existing processes (or those created later) that don't take their context from the process in which a variable was defined won't apply the variable.

4.2.4 History

The shell can maintain a history list into which it places the words of previous commands. This history mechanism lets you reuse commands or words from them in forming new ones. Use this mechanism to repeat commands or to correct minor typing mistakes in them. Here's how the C Shell's history mechanism is typically used:

```
% cat bug.c          <RETURN>
main()
{
printf("hello);
}
% cc !$             <RETURN>
cc bug.c
"bug.c", line 4:newline in string or char constant
"bug.c", line 5:syntax error
% ed !$             <RETURN>
ed bug.c
29
4s/);/"&/p
printf("hello");
w
30
q
% !c                <RETURN>
cc bug.c
% a.out             <RETURN>
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
printf("hello\n");
w
32
q
% !c -o bug         <RETURN>
cc bug.c -o bug
% size a.out bug    <RETURN>
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*          <RETURN>
ls -l a.out bug
-rwxr-xr-x 1 kate 3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 kate 3932 Dec 19 09:42 bug
% bug               <RETURN>
hello
% num bug.c | spp <RETURN>
spp: Command not found.
% ^spp^ssp          <RETURN>
num bug.c | ssp
1 main()
3 {
4 printf("hello\n");
5 }
% !! | prf          <RETURN>
num bug.c | ssp | prf
```

This example shows a very simple C program with some bugs. To begin, we use **cat**(1) to print the file *bug.c* onto the screen. Then, we attempt to run the C compiler, **cc**(1), referring to the file again as *!$*, which is an invocation of the **history** mechanism that means "use the last argument to the previous command." The exclamation point is the metacharacter that invokes the history mechanism and the dollar sign stands for the last (most recent) argument read by the shell. The shell echoes the command, as it would have been typed without using the **history** mechanism, and then executes it.

Since the compilation yielded error diagnostics, we invoke the line editor, **ed**(1) to fix the bug. Then the file is recompiled, this time referring to the **cc** command simply as **!c**. The notation **!***x* tells the shell to repeat the most recently submitted command that begins with character *x*. If specificity is necessary (for example, if other commands starting with *c* had been used recently), we can invoke the **history** mechanism by typing *!cc*. If further caution is needed, the form *!cc:p* prints the last command that started with "cc," without appending the <RETURN> that executes it.

After this recompilation, a run of the resulting *a.out* file reveals that a bug still exists, so we reinvoke the editor, and then the C compiler. This time, we add the **-o bug** switch to the **cc** command line, telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, the **history** mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then run the **size**(1) command to see how large the object files were, and then an **ls -l** command with the same argument list, denoting the argument list *\!**. Finally, we run the *bug* program to see that its output was indeed correct.

To make a numbered listing of the program we run the *num* program on the file *bug.c*. To remove blank lines in the output, we run it through the filter *ssp*, but misspell it as *spp*. To correct this, we use a shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. We then repeat the same command with *!!*, but send its output to the line printer.

Note: On DOMAIN nodes, <AGAIN> is often defined to copy all text between the cursor position and the next EOL into the "next input window." In fact, the DM's cut–and–paste facilities may be more effective than the **history** mechanism in certain situations. See *Getting Started With Your DOMAIN/IX System* and the *DOMAIN/IX Text Processing Guide* for more on the DM's cut–and–paste facility.

You can repeat a command from the history list by other means. The **history** command prints out a number of previous commands accompanied by the numbers with which they can be referenced. You can also refer to a previous command by searching for a string that appeared in it. See **csh**(1) in the *DOMAIN/IX Command Reference* for a complete description of these mechanisms.

4.2.5 Aliases

The shell has an alias mechanism that helps in transforming input commands. It can be used to simplify the commands you type, to supply default arguments to commands, or to do transformations on commands and their arguments. The alias mecha-

nism is similar to a macro facility. Some of the features obtained by aliasing can also be obtained using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment or involve commands such as **cd**, which must be done in the current shell. For example, if you'd like the command **ls** to always show sizes of files (i.e, do **-s**), use the following alias:

```
% alias ls ls -s <RETURN>
```

Or, you can create a "new" command called *dir* that does the same thing, by typing

```
% alias dir ls -s <RETURN>
```

Thus, the alias mechanism can be used to provide short names for commands, to supply default arguments, and to define new short commands in terms of other commands. You can also define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. For example, the alias for **cd** in our *.cshrc* example

```
% alias cd 'cd \!* ;ls' <RETURN>
```

causes the shell to automatically do an **ls** after every **cd**. We enclose the entire alias definition in single quotes (') to prevent most substitutions from occurring and to prevent the semi-colon (;) from being recognized as a metacharacter. The exclamation point (!) here is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing **cd** command, without giving an error message if no arguments are supplied. The semi-colon is used to indicate that one command is to be done first, followed by the next. Similarly, the definition

```
% alias whois 'grep \!^ /etc/passwd' <RETURN>
```

defines a command which looks up its first argument in the password file.

Note: The C Shell reads the *.cshrc* file each time it is invoked. If you put many commands there, shells tend to start slowly. We recommend that you limit the number of aliases in this file. Ten aliases cause no perceived delay. Fifty aliases cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

4.2.6 More Redirection; >> and >&

In addition to the standard output, commands also have a diagnostic output (or "error output") that is normally directed to the screen even when the standard output is redirected to a file or a pipe. If you need to redirect the diagnostic output to the same place as you redirect standard output (e.g., if you want to redirect the output of a long-running command into a file and need to have a record of any error diagnostics produced while the command was running), use the notation

```
command >& file
```

The **>&** here tells the shell to route both the diagnostic output and the standard output into *file*. Similarly, you can give the command

```
command |& prf
```

to route both standard and diagnostic output through the pipe to the **/com/prf** print spooler.

Note: This notation can be used when **noclobber** is set and *file* already exists:

command >&! *file*

Finally, it is possible to use the form

command >> *file*

to place output at the end of an existing file.

Note: If **noclobber** is set, an error results if *file* does not exist; otherwise, the shell creates *file* if it doesn't exist. A form

command >>! *file*

can be used if it's necessary to override **noclobber**'s error message.

4.2.7 Background, Foreground, and Suspended Jobs

When one or more commands are connected via pipes or as a sequence of commands separated by semicolons, the shell creates a single job consisting of all commands so connected. A single command without pipes or semicolons is, of course, the simplest job. Usually, every line typed to the shell creates a job.

If you type the ampersand (&) metacharacter at the end of a command line, the job generated by that command line is started as a background job. Thus, the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs "in the background" at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time. Thus,

% **du > usage &** <RETURN>

runs the **du**(1) program, which reports on the disk usage of your working directory (as well as any directories below it), puts the output into the file *usage*, and returns immediately with a prompt for the next command without waiting for **du** to finish. The **du** program continues executing in the background until finished, and the shell continues accepting input from you. When a background job terminates, the shell types a message before the next prompt, telling you that the job is completed. In the following example, the **du** job finishes sometime during the execution of the **mail**(1) command. Its completion is reported just before the prompt after the **mail** job is finished.

```
% du > usage &     <RETURN>
[1] 503
% mail kate        <RETURN>
How can I tell when a background job is finished?
bob
*** EOF ***
EOT
[1] - Done du > usage
%
```

If the job hadn't terminated normally, you might have gotten a message such as "Killed". To have terminations of background jobs reported at the time they occur

(possibly interrupting the output of other foreground jobs), set the **notify** variable. If you had done this for the previous example, the "Done" message might have appeared in the middle of the message to *kate*. Background jobs are unaffected by any signals from the keyboard (e.g., STOP, INTERRUPT, QUIT).

On DOMAIN systems, you can invoke a C Shell with or without the ability to suspend and then restart a process, or move it into or out of the foreground. The C Shell's ability to handle this kind of job control is determined by the state of the shell variable **inprocess**, which may be set or unset. (You may also invoke a C Shell with **inprocess** unset by including the **-j** switch on the **csh** or **start_csh** command line.) You can only use the **fg**, **bg**, and **stop** commands if **inprocess** is unset.

Note: If you create a C Shell as a remote process by using the DOMAIN system **crp** (create process) command, you won't have access to any of the job control features, regardless of the setting of **inprocess**.

When **inprocess** is set (default condition), these limitations apply:

- **/com/tb** always returns the message "no traceback available."
- Libraries loaded with the **inlib** command (built-in to the AEGIS Shell) are unavailable to programs running in an environment where **inprocess** is unset.
- **/com/las** only lists the address space occupied by itself.
- **/com/lopstr** shows only those streams that the C Shell has open.

Whether or not **inprocess** is set, information about all running jobs is recorded in a table maintained by the C Shell. In this table, the shell stores the names, arguments, and process numbers of all commands in the job. It also notes the working directory in which the job was started. Each job in the table is either running in the foreground with the shell waiting for it to terminate, running in the background, or suspended.

Only one job can be running in the foreground. Simultaneously, several jobs can be either running in the background or suspended. As each job is started, it is given a "job number." This number is used in conjunction with the commands below to suspend or kill the job. The job number assigned to a job remains the same until the job terminates, at which time the job number is available for reuse.

When a job is started in the background, the shell displays the job's number, as well as the process numbers of all its (top level) commands. This job, for example,

```
% ls -s | sort -n > usage &  <RETURN>
[2] 65 66
%
```

runs the **ls** program with the **-s** option, and pipes this output into the **sort** program with the **-n** option which puts its output into the file *usage*. Since an ampersand appears at the end of the line, these two programs start together as a background job. After starting the job, the shell prints the job number (e.g., 2) in brackets followed by the job's process numbers, then prompts for a new command.

To suspend a foreground job, send a STOP signal to the shell process. If you invoke the *bsd4.2_keys* key definitions, suspend is mapped to ↑Z. This sends a STOP signal to

the job that's currently running in the foreground. To suspend a background job, use the **stop** command. When jobs are suspended, they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job stops and reports this fact, much like it reports the termination of background jobs. For foreground jobs, this looks like

```
% du > usage  <RETURN>
<↑Z>
Stopped
%
```

The shell displays the "Stopped" message when it notices that a job (in this case, the **du** program) has stopped. When you use the **stop** command on a background job, the shell prints a slightly different message:

```
% sort usage &     <RETURN>
[1] 23
% stop %1          <RETURN>
[1] + Stopped (signal) sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended, then continued as background jobs using the **bg** command, allowing you to continue other work and stop waiting for the foreground job to finish. In this sequence, we start **du** in the foreground, stop it before it finishes, then continue it in the background:

```
% du > usage              <RETURN>
<↑Z>
Stopped
% bg
[1] du > usage &          <RETURN>
%
```

All job control commands can take an argument that identifies a particular job. All job name arguments must begin with a percent (%), since some of the job control commands also accept process numbers. To get the numbers of all running or suspended processes, use **ps**(1).

The default job (when no argument is given) is called the "current job" and is identified by a plus sign (+) in the output of the **jobs** command. When only one job is stopped or running in the background, it is always the current job. No argument is needed in this case. If you stop a job running in the foreground, it becomes the current job and the existing current job becomes the previous job, identified by a dash (-) in the output of **jobs**. When the current job terminates, the previous job becomes the current job.

When given, the argument to **jobs** is one of the following:

%-	the previous job
%n	where *n* is the job number

%pref where *pref* is some unique prefix of the command name and arguments of one of the jobs

%?string where *string* is a string found in only one of the command lines that set up a job.

The **jobs** command lists the table of jobs, giving the job number, commands, and status ("Stopped" or "Running") of each background or suspended job. With the **-l** option, the process numbers are also given.

```
% du > usage &                    <RETURN>
[1] 33
% ls -s | sort -n > myfile &     <RETURN>
[2] 34
% mail ers                       <RETURN>
<↑Z>
Stopped
% jobs                           <RETURN>
[1] - Running du > usage
[2] Running ls -s | sort -n > myfile
[3] + Stopped mail ers
% fg %ls                         <RETURN>
ls -s | sort -n > myfile
% more myfile                    <RETURN>
```

The **fg** moves a job into the foreground. If the job is suspended, it is restarted. If the job is already running in the background, it continues to run, but becomes the foreground job; consequently, it can accept signals or input from the terminal. In the above example, we use **fg** to change the **ls** job from the background to the foreground since we want to wait for it to finish before looking at its output file.

The **bg** command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the **bg** command changes a foreground job to a background job. The **stop** command suspends a background job.

The **kill** command terminates a background or suspended job immediately. In addition to jobs, **kill** may be given process numbers as arguments. Thus, in the example above, the running **du** command can be terminated as shown here:

```
% kill %1 <RETURN>
[1] Terminated du > usage
%
```

The **notify** command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes, instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal, it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the **s** (substitute) command in the text editor might take a long time:

```
% ed bigfile                <RETURN>
120000
1,$s/thisword/thatword/     <RETURN>
<↑Z>
Stopped
% bg                        <RETURN>
[1] ed bigfile &
%
  . . . some foreground commands . . .
[1] Stopped (tty input) ed bigfile
% fg                        <RETURN>
ed bigfile
w                           <RETURN>
120000
q                           <RETURN>
%
```

After we issue the **s** command, we stop the **ed** job with ↑Z, and then put it in the background using **bg**. Some time later when the **s** command is finished, **ed** tries to read another command and is stopped because jobs in the background cannot read from the terminal. The **fg** command returns the **ed** job to the foreground where it can once again accept commands from the terminal.

Note: The **jobs** command only prints jobs started in the currently executing shell. It knows nothing about background jobs started in other shells. Use **ps**(1) to find out about background jobs not started in the current shell.

4.2.8 Working Directories

The shell is always in a particular working directory. The "change directory" command, **cd**, changes the working directory of the shell. It's useful to make a directory for each project you work on, then place all files related to that project in that directory. The "make directory" command, **mkdir**(1), creates a new directory. The "print working directory" command, **pwd**(1), reports the absolute pathname of the working directory of the shell, i.e., the directory in which you are located. Thus, in this example, we create the directory *newdocs* and then move to it:

```
% pwd              <RETURN>
//ice/kate
% mkdir newdocs    <RETURN>
% cd newdocs       <RETURN>
% pwd              <RETURN>
//ice/kate/newdocs
%
```

No matter where you move to in a directory hierarchy, you can return to your "home" directory by typing the **cd** command with no arguments:

```
% cd  <RETURN>
```

The name .. ("dot dot") always means the directory above the current one. Thus,

```
% cd .. <RETURN>
```

changes the shell's working directory to the parent of (the directory immediately above) the current directory. The name can be used in any pathname; thus,

```
% cd ../programs <RETURN>
```

moves you to the directory *programs* contained in the directory above the current one. If you have several directories for different projects under your home directory, this shorthand notation makes it easier to switch between them.

The shell always remembers the pathname of its current working directory in the variable **cwd**. The shell can also be requested to remember the previous directory when you change to a new working directory. If the "push directory" command, **pushd**, is used in place of the **cd** command, the shell saves the name of the current working directory on a directory stack before changing to the new one. You can see this list at any time by typing the "directories" command **dirs**.

```
% pushd newpaper/references    <RETURN>
%/newpaper/references ~
% pushd /usr/lib/tmac           <RETURN>
/usr/lib/tmac ~/newpaper/references ~
% dirs                          <RETURN>
/usr/lib/tmac ~/newpaper/references &
% popd                          <RETURN>
%/newpaper/references ~
% popd                          <RETURN>
%
%
```

The list is printed in a horizontal line, reading left to right, with a tilde as shorthand for your home directory. The directory stack is printed whenever more than one entry is on it and it has changed. It is also printed by a **dirs** command. **Dirs** is usually faster and more informative than **pwd** since it shows the current working directory, as well as any other directories remembered in the stack.

The **pushd** command with no argument alternates the current directory with the first directory in the list. The "pop directory" command, **popd**, used without an argument, returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing **popd** several times in a series takes you backward through the directories you had been in (changed to) via the **pushd** command. Other options to **pushd** and **popd** manipulate the contents of the directory stack and change to directories not at the top of the stack. See **csh**(1) in the *DOMAIN/IX Command Reference* for details.

Since the shell remembers the working directory in which each job was started, it warns you when it thinks you might be restarting a foreground job that has a different working directory than the current working directory of the shell. Thus, if you start a background job, change the shell's working directory, then bring a background job into the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs -l          <RETURN>
//ice/kate
% cd myproject     <RETURN>
% dirs             <RETURN>
%/myproject
% ed prog.c        <RETURN>
1143
<↑Z>
Stopped
% cd ..            <RETURN>
% ls               <RETURN>
myproject
textfile
% fg               <RETURN>
ed prog.c               (working dir is: ~/myproject)
```

This way the shell warns you of an implied change of working directory, even though no **cd** command was issued. In our example, the **ed** job is still in */ice/kate/myproject* even though the shell changes to */ice/kate/*. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since a return to the shell implies a change of working directory.

```
% fg <RETURN>
ed prog.c                    (working dir is: ~/myproject)
. . . after some editing
q     <RETURN>
working dir is now: ~
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell assumes that a job stays in the same directory where it started. The **-l** option of **jobs** types the working directory of suspended or background jobs when it is different from the current working directory of the shell.

4.2.9 Useful Built-In Commands

We now describe some useful built-in shell commands and explain their usage.

The **alias** command is used to assign new aliases and to show existing aliases. With no arguments, it prints a list of the current aliases. With a single argument, such as

```
% alias ls <RETURN>
```

alias shows the current alias for that argument (i.e., **ls**).

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions produce.

The **history** command shows the contents of the history list. The numbers given with the history events help to reference previous events that are difficult to reference using the contextual mechanisms introduced above. Also, a shell variable called **prompt** tells the C Shell to use a specific character or string as the prompt. Thus, if you type

```
% set prompt='\! % ' <RETURN>
```

the shell prepends the number of the current command in the history list to the percent sign. Note that the exclamation point had to be escaped here even within single quotes (').

The **logout** command can be used to terminate a login shell in which **ignoreeof** is set.

The **rehash** command causes the shell to recompute a table of command locations. You must use **rehash** if you add a command to a directory in the current shell's search path. If a command isn't in the search path when the hash table is computed, the shell probably won't know that it exists.

The **repeat** command can be used to repeat a command several times. Thus, to make five copies of the file *one* in the file *five*, you could do this:

```
% repeat 5 cat one >> five <RETURN>
```

The **setenv** command can be used to set variables in the C Shell environment. Thus,

```
setenv TERM vt100
```

sets the value of the environment variable TERM to "vt100". To print out the environment, use **printenv** as shown here:

```
% printenv <RETURN>
USER=kate
LOGNAME=kate
PROJECT=none
ORGANIZATION=doc
NODEID=1054
PATH=:~com:/usr/ucb:/bin:/com:/usr/bin
SYSTYPE=bsd4.2
TERM=apollo_19L
NODETYPE=DN300
TZ=EST5EDT
HOME=//ice/kate
```

The **source** command forces the current shell to read commands from a file. Thus, use

```
source .cshrc
```

after making a change to the *.cshrc* file to have the change take effect immediately. The **unalias** command cancels aliases. **Unset** removes shell variables, and **unsetenv** removes environment variables.

4.3 Shell Control Structures and Shell Scripts

This section describes how to place commands in special files ("shell scripts") that invoke shells for reading and executing commands.

4.3.1 Invocation and the "argv" Variable

To run a C Shell script, you may type

```
% csh scriptname args <RETURN>
```

where *scriptname* is the name of the file containing a group of **csh** commands and *args* denotes a a sequence of optional arguments. The shell places these arguments in the variable **argv** and then begins to read commands from the script. These arguments placed in **argv** are made available as if they were ordinary shell variables. If you make the file *scriptname* executable by typing

```
% chmod 755 scriptname <RETURN>
```

and place the line

```
# !/bin/csh
```

as the first line of the file *scriptname*, a C Shell is automatically invoked to execute *scriptname* when you type

```
scriptname
```

In general, you should always start a shell script with a line of the the form

```
# !shell
```

where *shell* is the name of the shell that is to execute the script. Legal *shells* are:

/bin/csh the C Shell

/bin/sh the Bourne shell

/com/sh the AEGIS shell

If the file does not begin with a pound sign (#), the shell in which you invoked the script tries to execute it, with unpredictable results.

4.3.2 Variable Substitution

After each input line is broken into words and history substitutions are made, the input line is parsed into distinct commands. Before each command is executed, the shell does variable substitution on these words. Variable substitution is keyed by the dollar sign ($), and is a procedure by which the shell replaces the names of variables by their values. Thus,

```
echo $argv
```

when placed in a command script causes the current value of the variable **argv** to be echoed to the output of the shell script. It is an error for **argv** to be unset at this point.

The C Shell provides a number of notations for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if *name* is set and to 0 otherwise. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

$#*name*

expands to the number of elements in the variable *name*. To illustrate this, consider the following:

```
% set argv=(a b c) <RETURN>
% echo $?argv      <RETURN>
1
% echo $#argv      <RETURN>
3
% unset argv       <RETURN>
% echo $?argv      <RETURN>
0
% echo $argv       <RETURN>
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus, this gives the first component of argv or in the example above *a*:

$argv[1]

Similarly,

$argv[$#argv]

gives *c*, and

$argv[1-2]

gives *a b*. Other notations useful in shell scripts are

$*n*

where *n* is an integer as a shorthand for

$argv[*n*]

the *nth* parameter and

$*

which is a shorthand for

$argv

The form $$ expands to the process number of the current shell. This process number is unique in the system, and can be used in generation of unique temporary filenames. The form $< is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus, the sequence

```
#!/bin/csh
#
echo 'yes or no?\c'
set a=($<)
```

writes out the prompt *yes or no?* without a newline and then reads the answer into the variable *a*. In this case, *$#a* is 0 if either a blank line or end-of-file (↑D) is typed.

For compatibility with the way older shells handled parameters,

```
$argv[n]
```

yields an error if *n* is not in the range

```
1-$#argv
```

while *$n* never yields an out-of-range subscript error. It is never an error to give a subrange of the form

```
n-
```

If the given variable has less than *n* components, no words are substituted. A range of the form

```
m-n
```

returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is within range.

4.4 Expressions

It's important to be able to evaluate expressions in the shell based on the values of variables. All the arithmetic operations of C are available in the shell with the same precedence that they have in C. In particular, the operations == and != compare strings and the operators && and || implement the boolean and/or operations. The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (e.g., *, ?, or [ ]), and the test is whether the string on the left matches the pattern on the right.

The shell also allows file inquiries of the form

```
-? filename
```

where the question mark is replaced by a number of single characters. For instance, the expression primitive

```
-e filename
```

tells whether the file *filename* exists. Other primitives test for read, write, and execute access to the file, whether it is a directory, or has non-zero length. You can test whether a command terminates normally, by a primitive of the form

```
{ command }
```

which returns true (i.e., 1) if the command succeeds (exits normally with exit status 0), or 0 if the command terminates abnormally or with exit status non-zero. If you need more detailed information about the execution status of a command, execute it, then examine the **$status** variable.

Note: Since **$status** is set by every command, you must save a particular command's **$status** if you can't examine it immediately following the command's execution.

4.4.1 A Sample Shell Script

The following shell script, called *copyc*, uses the C Shell's expression mechanism and some of its control structures:

```
# !/bin/csh
# Copyc copies those C programs in the specified list
# to the directory ~backup if they differ from the files already in ~backup
#
set noglob
foreach i ($argv)

if ($i !~ *.c) continue # not a .c file so do nothing

if (! -r ~backup/$i:t) then
echo $i:t not in backup... not cp\'ed
continue
endif

cmp -s $i ~backup/$i:t # to set $status

if ($status != 0) then
echo new backup of $i
cp $i ~backup/$i:t
endif
end
```

This script uses the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching **end** for each of the values given between the left and right parentheses with the named variable (i.e., **i** set to successive values in the list). Within this loop, you may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (*i* in this case) has the value it was assigned at the last iteration.

Here, we set the variable **noglob** to prevent filename expansion of the members of **argv**. This is recommended if the arguments to a shell script are filenames that have already been expanded or if arguments may contain filename expansion metacharacters. You can also quote each use of a dollar sign variable expansion, though this is rather tedious.

The other control construct used here is a statement of the form

```
if ( expression ) then
command
...
endif
```

Note: The placement of the these keywords is not flexible. The following two formats are unacceptable to the C Shell:

```
#this won't work
if ( expression )
then
command
...
endif

#nor will this
if ( expression ) then command endif
```

The shell does have another form of the **if** statement:

```
if ( expression ) command
```

For the sake of appearance, this can be written with an escaped newline:

```
if ( expression ) \
command
```

The *command* must not involve a pipe (|), ampersand (&), or semi-colon (;). It must not be another control command. In the second form, the final backslash (\) must immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**, as shown here:

```
if ( expression ) then
commands
else if (expression ) then
commands
...
else
commands
endif
```

Another important mechanism used in shell scripts is the colon (:) modifier. We can use the modifier *:r* here to extract a root of a filename, or *:e* to extract the extension. Thus, if the variable *i* has the value */mnt/foo.bar*, then

```
% echo $i $i:r $i:e <RETURN>
/mnt/foo.bar /mnt/foo bar
%
```

shows how the *:r* modifier strips off the trailing *.bar* and the the *:e* modifier leaves only the *bar*. Other modifiers take off the last component of a pathname and leave the head *:h*, or all but the last component of a pathname and leave the tail *:t*. (These modifiers are fully described under **csh**(1) in the *DOMAIN/IX Command Reference*.) You can also use the command substitution mechanism, described in the next major section, to perform modifications on strings.

Note: The C Shell allows only one colon modifier on a dollar sign substitution. Thus, this doesn't produce the results that one would otherwise expect:

```
% echo $i $i:h:t <RETURN>
/a/b/c /a/b:t
%
```

Finally, we note that the pound sign (#) lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a pound sign are discarded by the shell. This character can be quoted using a single quote (') or backslash (\) to place it in an argument word.

4.4.2 Other Control Structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
commands
end
```

and

```
switch ( word )

case str1:
commands
breaksw

...

case strn:
commands
breaksw

default:
commands
breaksw

endsw
```

Note: The C Shell uses **breaksw** to exit from a **switch**, while **break** exits a **while** or **foreach** loop. A common mistake in C Shell scripts is the use of **break** rather than **breaksw** in switches.

Finally, **csh** allows a **goto** statement, with labels that look the same as they do in C:

```
loop:
commands
goto loop
```

4.4.3 Supplying Input to Commands

By default, commands run from shell scripts receive the standard input of the shell that is running the script. This allows shell scripts to participate in pipelines, but mandates extra notation for commands that are to take in–line data.

Thus, we need a metanotation for supplying in–line data to commands in shell scripts. As an example, consider this script that runs the editor to delete leading blanks from the lines in each argument file:

```
#!/bin/csh
# deblank, a script to remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/"[ ]*//
w
q
'EOF'
end
%
```

The << 'EOF' notation means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly the 'EOF' itself. The fact that the EOF is quoted causes the shell to forego variable substitution on the intervening lines. In general, if any part of the word following the "<<" that the shell uses to terminate the text to be given to the command is quoted, then variable substitutions are not performed. In this case, since we used the form "1,$" in our editor script, we needed to ensure that this dollar sign did not trigger a substitution.We could also have ensured this by preceding the dollar sign here with a backslash, i.e.,

```
1,\$s/†[ ]*//
```

but quoting the EOF terminator is a more reliable way of achieving the same result.

4.4.4 Catching Interrupts

If your shell script creates temporary files, you may wish to catch interruptions of the shell script so that you can clean up these files. To do this, use the construct

```
onintr label
```

where *label* is a label in the program. If an interrupt is received, the

shell does a "goto label." You can then remove the temporary files and use **exit** (built in to the shell) to exit the shell script. To exit with a non–zero status, type

```
exit(1)
```

to exit with status 1.

4.4.5 Additional Options

Other features of the shell are useful to writers of shell procedures. The **verbose** and **echo** options and the related **–v** and **–x** command line options can help trace the actions of the shell. The **–n** option causes the shell to read commands but not to execute them, something which may be useful during debugging.

The double quote (") mechanism allows only some of the expansion mechanisms discussed thus far to occur on the quoted string, and serves to make this string into a single word as a single quote (') does.

4.5 Other Shell Features

The C Shell features discussed in this section are less commonly used. In particular circumstances, it may be necessary to know the exact nature and order of different substitutions performed by the shell. The precise meaning of certain combinations of quotations is also important at times. Furthermore, the shell has many command line option flags mostly used in the writing of UNIX programs, and debugging of shell scripts. See **csh**(1) in the *DOMAIN/IX Command Reference* for more information.

4.5.1 Loops at the Terminal; Variables as Vectors

Occasionally, the **foreach** control structure may be used at a terminal to aid in performing a number of similar commands. For instance, to count the number of files in several directories (*dir1, dir2*, and *dir3*) that had the characters ".TS" or ".EQ" at the beginning of a line, you could use several command lines:

```
% grep -c '^\.TS|.EQ' dir1        <RETURN>
3
% grep -c '^\.TS|.EQ' dir2        <RETURN>
5
% grep -c '^\.TS|.EQ' dir3        <RETURN>
6
```

or you could use **foreach** to do this more easily:

```
% foreach i ('dir1' 'dir2' 'dir3') <RETURN>
? grep -c  '^\.TS|.EQ'  $i        <RETURN>
? end                             <RETURN>
3
5
6
%
```

Here, the shell prompts for input with a question mark (?) when reading the body of the loop. Variables containing lists of filenames or other words are also useful in loops. You can, for example, do the following:

```
% set a=(`ls`)       <RETURN>
% echo $a            <RETURN>
csh.n csh.rm
% ls                 <RETURN>
csh.n
csh.rm
% echo $#a           <RETURN>
2
%
```

The **set** command here gave the *a* variable a list of all the filenames in the current directory as value. You can then iterate these names to perform any chosen function.

The output of a command within backquotes (`) is converted by the shell to a list of words. You can also place the backquoted string within double quotes to take each

(non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier *:x* can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.5.2 Braces { ... } in Argument Expansion

Another form of filename expansion involves the brace characters ({ }), which specify that the delimited strings separated by a comma (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus,

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames need not exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments that are not filenames, but have common parts. For example,

```
% mkdir ~/{hdrs,retrofit,csh}  <RETURN>
```

makes subdirectories *hdrs, retrofit,* and *csh* in your home directory. This is most useful when the common prefix is longer than shown in this example, i.e.,

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.5.3 Command Substitution

A command enclosed in backquotes (‘) is replaced, just before filenames are expanded, by the output from that command. Thus, you may type

```
% set pwd=‘pwd‘  <RETURN>
```

to save the current directory in the variable *pwd*, or

```
% ex ‘grep -l TRACE *.c‘  <RETURN>
```

to run the editor **ex**(1), supplying as arguments filenames ending in *.c* that have the string *TRACE* in them.

Note: Command expansion also occurs in input redirected with "<<" and within double quoted (") notations. See the *DOMAIN/IX Command Reference* for details.

4.6 Summary of C Shell Metacharacters

This section lists the metacharacters recognized by the C Shell. Many of these characters also have special meaning in expressions. See the information on **csh**(1) in the *DOMAIN/IX Command Reference* for a complete list.

Note: If you use the DM environment variable NAMECHARS (see Chapter 1) to assign DOMAIN naming server metameanings to tilde, grave accent, or backslash

-- and you use one of those characters (escaped) in a pathname component -- the character is interpreted not as a literal, but according to the naming server's rules.

4.6.1 Syntactic

;	Separates commands to be executed sequentially
|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed in background

4.6.2 Filename

/	Separates components of a file's pathname
.	Separates root parts of a filename from extensions
?	Expansion character matching any single character
*	Expansion character matching any sequence of characters
[ ]	Expansion sequence matching any single character from a set
~	Used at the beginning of a filename to indicate home directories
{ }	Specifies groups of arguments with common parts

4.6.3 Quotation

\	Prevents metameaning of following single character
'	Prevents metameaning of a group of characters
"	Like a single quote ('), but allows variable and command expansion

4.6.4 Input/Output

<	Indicates redirected input
>	Indicates redirected output

4.6.5 Expansion/Substitution

$	Indicates variable substitution
!	Indicates history substitution
:	Precedes substitution modifiers
↑	Used in special forms of history substitution
`	Indicates command substitution

4.6.6 Miscellaneous

#	Begins shell comment
-	Prefixes option (flag) arguments to commands
%	Prefixes job name specifications

Chapter 5

Using the BSD4.2 Mail Program

5.1 Introduction

Mail gives DOMAIN/IX users a simple way to communicate with other users of their DOMAIN system, or with users at other sites to which their DOMAIN system can connect (e.g., via ARPANET or USENET). The **Mail** program divides incoming mail into its constituent messages and lets you deal with these messages in any order. In addition, **Mail** provides a set of editing commands for preparing messages, building mailing lists, and sending mail.

The **Mail** program we describe in this chapter is the one included with the *bsd4.2* version of DOMAIN/IX software. It resides in the */bsd4.2/usr/ucb/Mail* file. The *sys5* version has its own mail program, */bin/mail*. It is described in the *DOMAIN/IX Command Reference for System V* under **mail**(1). Before reading this chapter, you should take the time to become familiar with a UNIX shell, any of the available text editors, and some of the common UNIX commands.

The *bsd4.2* mail system accepts incoming messages for you from other people and collects them in a file, called your *system mailbox*. When you log in, the system tells you if there are any messages waiting in your system mailbox. If you are a C Shell user, you can request that the shell notify you of the arrival of new mail, but the shell must know where to find your mailbox. Your system mailbox is located in the directory */usr/spool/mail* in a file with your login name. If your login name is *sam*, you can make **csh**(1) notify you of new mail by including the following line in your *.cshrc* file:

```
set mail=/usr/spool/mail/sam
```

When you read your mail using **Mail**, it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message includes the name of the sender and the date on which it was sent.

5.1.1 Sending Mail

To send a message to a user whose login name is *root*, use the shell command:

```
% Mail root <RETURN>
```

then type your message. When you reach the end of the message, hit <RETURN> and then an EOF (End Of File), usually mapped to ↑Z. The DM (Display Manager) echoes

```
*** EOF ***
```

and sends an End-of-File signal to **Mail**. This causes **Mail** to echo

```
EOT
```

and return you to the shell.

Note: This chapter assumes that, if you are using a DOMAIN node, the sequence ↑Z is mapped to the DM command **eef** (insert End-Of-File). As noted above, when you type the key mapped to **eef**, the DM echoes the string

```
*** EOF ***
```

and sends an End-Of-File to the shell, which passes it along to **Mail**.

The next time the person to whom the message was addressed logs in, this message appears in the shell transcript pad:

```
You have mail.
```

If, while composing a message, you decide not to to send it after all, you can kill it with an INTERRUPT signal (↑I). Typing a single ↑I causes **Mail** to display the message

```
(Interrupt -- one more to kill letter)
```

Typing a second ↑I causes **Mail** to save your partial letter on the file *dead.letter* in your home directory and to abort the letter. Once you've sent mail to someone, it isn't easy to cancel the message.

The message your recipient reads consists of the message you typed preceded by a line telling them who sent the message (your login name) and the date and time sent. To send the same message to others, list their login names on the **Mail** command line. For example, this sends the reminder to users *sam*, *bob*, and *john*:

```
% Mail sam bob john <RETURN>
Tuition fees are due next Friday.  Don't forget!!
<↑Z>
*** EOF ***
EOT
%
```

5.1.2 Receiving Mail

If, when you log in, you see this message

```
You have mail.
```

you can read the mail simply by typing the following:

% Mail <RETURN>

Mail responds by displaying its version number and date and then listing the messages you have waiting. Then, it displays a prompt and awaits your command. The messages are numbered starting with 1, and you must refer to a specific message by its number. **Mail** keeps track of which messages are *new* (received since you last read your mail) and *read* (have been read by you). New messages are marked with an *N* in the header listing, and old (but unread) messages are marked with a *U*. **Mail** tracks new/old and read/unread messages by putting a header field called *Status* into your messages.

To look at a specific message, use the **type** command (you can abbreviate it to **t**). For example, if you have the following messages:

```
N 1 root     Wed Sep 21 09:21  "Tuition fees"
N 2 sam      Tue Sep 20 22:55
```

you can examine the first message by giving this command:

type 1 <RETURN>

which causes **Mail** to display:

```
Message  1:
From root  Wed Sep 21 09:21:45 1978
Subject: Tuition fees
Status: R

Tuition fees are due next Wednesday. Don't forget!!
```

Many **Mail** commands that operate on messages take a message number as an argument, in the manner of the **type** command. For these commands, there is a notion of a current message. When you enter the **Mail** program, the current message is initially the first one. Thus, you can often omit the message number and use the abbreviated form of the **type** command. For example, this types the current message:

t <RETURN>

As a further shorthand, you can type a message by simply giving its message number. Hence, this types the first message:

1 <RETURN>

Suppose you want to read the messages in your mailbox in order, one after another. You can read the next message in **Mail** by simply hitting <RETURN>. As a special case, you can type a newline as your first command to **Mail** to type the first message.

If, after typing a message, you want to immediately send a reply, use the **reply** command. **Reply**, like **type**, takes a message number as an argument. **Mail** then begins a message addressed to the user who sent you the "current message." You may type

your letter in reply, followed by a ↑Z at the beginning of a line, as before. **Mail** echoes "EOT", then types the ampersand prompt to indicate its readiness to accept another command. In our example, if, after typing the first message, you wished to reply to it, you might give the command:

reply <RETURN>

Mail responds by printing

```
To: root
Subject: Re: Tuition fees
```

and waiting for you to enter your letter. You're now in the message collection mode described earlier. **Mail** gathers up your message until you terminate the message by typing ↑Z. Note that it copies the subject header from the original message. This is useful in that correspondence about a particular matter tends to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information in them is also used. For example, if a letter has a *To:* header that lists several recipients, **Mail** arranges to send your reply to the same people as well. Similarly, if the original message contains a *Cc:* (carbon copies to) field, **Mail** sends your reply to all those users, too. **Mail** normally doesn't send the message to you, even if your name appears in the *To:* or *Cc:* field, unless you explicitly ask to be included. We cover this subject in more detail later.

After typing in your letter, the dialog with **Mail** might look like this:

```
reply <RETURN>
To: root
Subject: Tuition fees

Thanks for the reminder
*** EOF ***
EOT
&
```

The **reply** command (abbreviated to **r**) is especially useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation.

At times, you may receive a message that was sent to several people and want to reply *only* to the person who sent it. **Reply** with a capital **R** does the trick.

While reading your mail, you can send a new message (not a reply) to someone with the **mail** command (which can be abbreviated to **m**). It use the names of the intended recipients as arguments. Thus, to send a message to *frank*, specify the following:

```
mail frank <RETURN>
This is to confirm our meeting next Friday at 4.
<↑Z>
*** EOF ***
EOT
&
```

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave **Mail**. To avoid saving a message in *mbox*, use the **delete** command:

delete 1 <RETURN>

This prevents **Mail** from saving message 1 (from root) in *mbox*. Besides not saving deleted messages, **Mail** doesn't let you type them, either. Thus, deleted messages disappear altogether, along with their message numbers. The **delete** command can be abbreviated to simply **d**.

Use the **set** command to tailor many features of **Mail**. This command has two forms, depending on whether you are setting a binary option or a valued option. Binary options are either on or off. For example, the **ask** option informs **Mail** that, each time you send a message, you want it to prompt for a subject header to be included in the message. To set the **ask** option, type

set ask <RETURN>

Another useful **Mail** option is **hold.** By default, **Mail** moves the messages from your system mailbox to the file *mbox* in your home directory when you leave **Mail**. If you want **Mail** to keep your letters in the system mailbox instead, set the **hold** option.

Valued options set numeric or string values that **Mail** uses to adapt to your tastes. For example, the **SHELL** option tells **Mail** which shell you like to use, specifying

set SHELL=/bin/csh <RETURN>

for example. (Note that no spaces are allowed in the command line.) A complete list of the **Mail** options appears at the end of this chapter.

Another important valued option for terminal users is **crt.** If you use a fast video terminal, you may find that when you print long messages, they scroll by too quickly for you to read them. With the **crt** option, you can make **Mail** print any message larger than a given number of lines by sending it through the well-known file perusal filter called **more**(1). For example,

set crt=24 <RETURN>

pipes any message longer than 24 lines through **more**.

Note: The **crt** option is unnecessary when using **Mail** on a DOMAIN node, since scrolling back through the message transcript may be done at leisure.

Mail also provides "aliases", names that stand for one or more real user names. **Mail** sent to an alias is actually sent to the list of real users associated with the alias. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The **alias** command in **Mail** defines an alias. Suppose that the users in a project are named Susan, Sally, Sam, and Steve. To define an alias called **project** for them, use

alias project susan sally sam steve <RETURN>

The **alias** command can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named Cindy Walukiewicz has the login name *walukiewicz_c*, use

alias walukiewicz_c cindy <RETURN>

to avoid typing (and probably misspelling) the longer name *walukiewicz_c*.

You may create a special file of aliases and options that are placed in effect automatically every time you invoke **Mail**. When **Mail** is invoked, it first reads a system-wide file */usr/lib/Mail.rc*, and then a user-specific file, *.mailrc* (found in your home directory). The system-wide file is maintained by the system administrator and contains **set** commands that are applicable to all system users. You may create a *.mailrc* file, set options, and define individual aliases. A typical *.mailrc* file might look like this:

```
set ask nosave SHELL=/bin/csh
```

As you can see, you can set many options in the same **set** command. The **nosave** option is described in another section.

Mail aliasing is implemented at the system-wide level by the mail delivery system **sendmail**(8). (See *System Administration for DOMAIN/IX BSD4.2* for further details concerning **sendmail**.) These aliases are stored in the */usr/lib/aliases* file and are accessible to all users of the system. The lines in this file have the form:

alias: *name1, name2, name3*

where *alias* is the mailing list name and *name1*, *name2*, and *name3* are the members of the list. Continue long lists onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command **newaliases** after editing */usr/lib/aliases* since the delivery system uses an indexed file created by **newaliases**.

Specifying the **-f** flag on the command line causes **Mail** to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file *letters*, you can use **Mail** to read them:

% **Mail -f letters** <RETURN>

You can use all the **Mail** commands described in this document to examine, modify, or delete messages from the file *letters*, which is rewritten when you leave **Mail** with the **quit** command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read *mbox* in your home directory by simply using

% **Mail -f** <RETURN>

Normally, messages that you examine using the **type** command are saved in the file **mbox** in your home directory if you leave **Mail** with the **quit** command described below. To retain a message in your system mailbox, you can use the **preserve** command to tell **Mail** to leave it there. The **preserve** command accepts a list of message numbers, just like **type** and may be abbreviated to **pre**.

Messages in your system mailbox that you don't examine are normally retained in your system mailbox. To have such a message saved in *mbox* without reading it, use the **mbox** command. For example,

mbox 2 <RETURN>

causes the second message (from *sam*) to be saved in *mbox* when the **quit** command is executed. **Mbox** is also the way to direct messages to your *mbox* file if you have set the **hold** option described above. **Mbox** can be abbreviated to **mb**.

You can leave **Mail** with the **quit** command (which can be abbreviated to **q**). It saves the messages you have read (typed), but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you see them the next time you type:

```
% Mail <RETURN>
```

To leave **Mail** without altering either your system mailbox or *mbox*, type the **x** command (short for **exit**), which immediately returns you to the shell without changing anything.

To execute a shell command without leaving **Mail**, type the command preceded by an exclamation point, just as in the **vi** text editor. For example,

```
!date <RETURN>
```

displays the current date without leaving **Mail**.

Finally, the **help** command prints out a brief summary of the **Mail** commands, using only the single character command abbreviations.

5.2 Maintaining Folders

Mail includes a simple facility for maintaining groups of messages together in folders. To use it, you must tell **Mail** where you wish to keep your folders. Each folder of messages is a single file. For convenience, all of your folders are kept in a single directory of your choosing. To tell **Mail** where your folder directory is, put a line of the following form in your *.mailrc* file:

```
set folder=letters
```

If your folder directory does not begin with a slash (/), **Mail** looks for the folder directory starting from your home directory. Thus, if your home directory is */usr/joe*, the above example told **Mail** to find your folder directory in */usr/joe/letters*.

Anywhere a filename is expected, you can use a folder name, preceded with a plus sign (+). For example, to put a message into a folder with the **save** command, use:

```
save +letters <RETURN>
```

to save the current message in the *letters* folder. If the letters folder doesn't yet exist, it is created. Note that messages retained with the **save** command are automatically removed from your system mailbox.

To put a copy of a message in a folder without causing that message to be removed from your system mailbox, use the **copy** command, which is identical in all other respects to the **save** command. For example, this copies the current message into the *letters* folder and leaves a copy in your system mailbox:

```
copy +letters <RETURN>
```

You may use the **folder** command to direct **Mail** to the contents of a different folder. This, for example, directs **Mail** to read the contents of the *letters* folder:

```
folder +letters <RETURN>
```

All commands that you can use on your system mailbox, including **type**, **delete**, and **reply**, also apply to folders. To inquire which folder you are currently editing, simply use this command:

folder <RETURN>

To list your current set of folders, use the **folders** command.

To start **Mail** reading one of your folders, you can use the **–f** option described above. For example, this causes **Mail** to read your *classwork* folder without looking at your system mailbox:

% **Mail –f +classwork** <RETURN>

5.3 Tilde Escapes

While typing in a message, it helps to be able to invoke a text editor on the partially–composed message, print the message, execute a shell command, or do some other function. **Mail** provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

~p <RETURN>

This prints a line of dashes, the recipients of your message, and the text of the message so far. Since **Mail** requires two consecutive ↑I's to kill a letter, you can use a single ↑I to abort the output of **~p** or any other tilde escape without killing your letter.

If you are dissatisfied with the message as it stands, you can invoke a UNIX text editor on the message using the escape

~e <RETURN>

which causes the message to be copied into a temporary file, then starts the editor. After modifying the message to your satisfaction, write it out and quit the editor. **Mail** responds with

```
(continue)
```

after which you may continue typing text to append to your message, or type ↑Z to end the message. A standard text editor is provided by **Mail**. You can override this default by setting the valued option **EDITOR** to something else, e.g.,

set EDITOR=/bin/ex <RETURN>

To use the UNIX screen editor on your current message, you can use the escape

~v <RETURN>

which works like **~e**, except that **vi**(1) is invoked instead. A default screen editor is defined by **Mail**, but you can set the valued option **VISUAL** to the pathname of a different editor.

It's often useful to include the contents of some file in your message; the escape

~r filename <RETURN>

helps serve this purpose, and causes *filename* to be appended to your current message. **Mail** complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text. The *filename* may contain shell metacharacters such as the asterisk (*) and question mark (?) which are expanded according to the conventions of your shell.

As a special case of **~r**, the escape

~d <RETURN>

reads in the file *dead.letter* in your home directory. This may be useful, since **Mail** copies the text of your message there when you kill a message with ↑I.

To save the current text of your message on a file, use the

~w filename <RETURN>

escape. **Mail** prints out the number of lines and characters written to the file, after which you may continue appending text to your message. Shell metacharacters may be used in the filename, as in **~r**, and are expanded according to the conventions of your shell.

If you are sending mail from within **Mail's** command mode, you can read a message sent to you into the message you are constructing with the escape:

~m 4 <RETURN>

which reads message 4 into the current message. The text of the message is shifted right by one tab stop. You can name any non-deleted message or list of messages. Messages can also be forwarded without shifting by a tab stop with **~f**. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

~t name1 name2 ... <RETURN>

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list still receive the message; you cannot remove someone from the recipient list with **~t**.

If you wish, you can associate a subject with your message by using the escape

~s Arbitrary string of text <RETURN>

which replaces any previous subject with *Arbitrary string of text*. The subject, if given, is sent near the top of the message (prefixed with "Subject:"). You can see what the message will look like by using **~p**.

If you need to list certain people as recipients of "carbon" copies of a message rather than of the message itself, use the escape

~c name1 name2 ... <RETURN>

The above line adds the named people to the "Cc:" list. Again, you can execute **~p** to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subject:" field, and the carbon copies the "Cc:" field. To edit these in ways impossible with the **~t**, **~s**, and **~c** escapes, use the escape

~h <RETURN>

which prints "To:" followed by the current list of recipients and leaves the cursor at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients.

You may use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the pound (#) and at (@) symbols,

```
~h <RETURN>
To: root ers####eve
```

changes the initial recipients *root ers* to *root eve*. When you type a newline, **Mail** advances to the "Subject:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use **~p** to print the current text of the header fields and the body of the message.

Note: In the DOMAIN/IX implementation of **mail**, the **~h** escape doesn't properly carry information given on the command line over into the interactive editing session. The following is an example of what takes place when you use the **~h** escape:

```
% mail mary
~h
To:<RETURN>
Subject: useful information
Cc: <RETURN>
Bcc: <RETURN>
.
.
EOT
No recipients specified
"dead.letter" 6/104
```

To effect a temporary escape to the shell, the escape

~!command <RETURN>

is used, which executes *command* and returns you to mailing mode without altering the text of your message. To filter the body of your message through a shell command instead, use

~|command <RETURN>

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, **Mail** assumes that something is amiss and retains the old version of your message. A frequently–used filter is the command **/bin/fmt**, designed to format outgoing mail.

To effect a temporary escape to **Mail** command mode instead, you can use the

~:Mail command

escape. This is especially useful for retyping the message you are replying to, using, for example:

~:t <RETURN>

It is also useful for setting options and modifying aliases.

To send a message that contains a line beginning with a tilde, you must escape the tilde with another tilde. Thus, for example,

~~This line begins with a tilde. <RETURN>

sends the line

```
~This line begins with a tilde.
```

Finally, the escape

~? <RETURN>

prints out a brief summary of the available tilde escapes.

Mail lets you change the escape character with the **escape** option. For example,

set escape=] <RETURN>

sets the escape character to a right bracket instead of a tilde. Doing this causes everything previously said about the tilde to apply to the right bracket. Changing the escape character removes the special meaning of tilde.

5.4 Network Access

This section describes how to send mail to people on other networks. Consult your system administrator for information about off-net communications facilities available at your site.

5.4.1 ARPANET

If your site includes a node that is directly (or even indirectly) connected to the ARPANET network, you can send messages to people on the Arpanet using a name like

name@host

where *name* is the login name of the person you're trying to reach and *host* is the name of the machine on the ARPANET where *name* has a login account.

If your intended recipient logs in on a machine connected to yours via **uucp**(1C), sending mail is more complicated. You must know the list of machines through which your message must travel to arrive at its intended destination. So, if *recipient* logs in on a machine directly connected to yours, you can send mail to recipient using the syntax:

host!name

where, again, *host* is the name of the machine and *name* is recipient's login name. If your message must go through an intermediate machine first, you must use the syntax

intermediate!host!name

and so on. It is a feature of **uucp** that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Ask your system administrator about the machines connected to your site.

5.4.2 Special Recipients

As described previously, you can send mail to either user names or **alias** names. You may also send messages directly to files or to programs, using special conventions. If a recipient name has a slash (/) in it or begins with a plus sign (+), it is assumed to be the pathname of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (i.e., one for which a slash wouldn't usually be needed) precede the name with a period and a slash (./). For example, to send mail to the file *memo* in the current directory, give the command

% **Mail ./memo** <RETURN>

If the name begins with a plus sign, it is expanded into the full pathname of the folder name in your *folder* directory. You can use this ability to send mail to files for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the **alias** command for the group. Using our previous **alias** example, give the command

alias project sam sally steve susan /usr/project/mail_record

Then, all mail sent to *project* is saved on the file */usr/project/mail_record,* as well as sent to the members of the project. This file can be examined using **Mail -f.**

It is sometimes useful to send mail directly to a program. Suppose you write a project billboard program and want to access it using **Mail.** To send messages to the billboard program, you can send mail to the special name *|billboard*, for example. **Mail** treats recipient names that begin with a pipe character (|) as a program to which mail is sent. You may want to create an **alias** to reference any filename prefaced by a pipe character.

Note: The shell treats the pipe character specially, so you must quote it on the command line. You must also present the "| program" as a single argument to **Mail.** We recommend that you surround the entire name with double quotes. This also applies to usage in the **alias** command. For example, to alias *rmsgs* to *rmsgs -s* you must type:

alias rmsgs "| rmsgs -s" <RETURN>

5.4.3 Message Lists

Several **Mail** commands accept a list of messages as an argument. Along with **type** and **delete** (already described in a previous section), there is the **from** command. It

prints the message headers associated with the message list passed to it. The **from** command is particularly useful in conjunction with some of the message list features described next.

A message list consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers that directly specify messages, or one of the following special characters:

^ the first message that is not deleted

. the current message

$ the last message that is not deleted

Note: The message list is being supplied as an argument to the **undelete** command, which operates on deleted messages only, the caret (^) operates on the first deleted message, and so on.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

type 1-4

and to print all the messages from the current message to the last message, use

type .-$

A name is a username. The usernames given in the message list are collected and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every relevant (not deleted, deleted) message sent by one those users is selected. Thus, to print every message sent to you by **root**, specify the following

type root <RETURN>

As a shorthand notation, simply specify an asterisk (*) to get every relevant message:

type * <RETURN>

prints all undeleted messages,

delete * <RETURN>

deletes all undeleted messages, and

undelete * <RETURN>

undeletes all deleted messages.

You can search for the presence of a word in subject lines with a slash (/). For example, to print the headers of all messages that contain the word *PASCAL*, specify

from /pascal <RETURN>

Note that subject searching ignores uppercase and lowercase differences.

5.5 Summary of Commands

! Prefaces a command to be executed by the shell.

- Goes to the previous message and prints it. If you give a decimal number *n* as an argument, **mail** goes to the *n*th previous message and prints it.

Print Like **print**, but also prints out ignored header fields. See also **print** and **ignore**.

Reply Frames a reply to a one or more messages. (Note the capital R in the name.) The reply (or replies, if using multiple messages) is sent only to the person who sent you the message (respectively, the set of people who sent the messages to which you are replying). You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began that way. If the original message included a **reply–to** header field, the reply goes *only* to the recipient named by **reply–to.** You type in your message using the same conventions available to you through the **mail** command. The **Reply** command is especially useful for replying *only* to the originator of a message sent to an enormous distribution group.

Type Identical to the **Print** command.

alias Defines a name to represent a set of other names. Use this to send messages to a certain group of people without having to retype their names. For example, this creates an alias *project* that expands to John, Sue, Willie, and Kathryn:

alias project john sue willie kathryn

alternates If you have accounts on several machines, you may find it convenient to use the */usr/lib/aliases* on all the machines except one to direct your mail to a single account. The **alternates** command is used to inform **Mail** that each of these other addresses is really you. **Alternates** takes a list of user names and remembers that they are all actually you. When you **reply** to messages sent to one of these alternate names, **Mail** doesn't bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism). If **alternates** is given no argument, it lists the current set of alternate names. **Alternates** is usually used in the *.mailrc* file.

chdir Lets you change your current directory. **Chdir** takes a single argument – the pathname of the new working directory. Without an argument, it changes to your home directory.

copy Does the same thing that **save** does, except that it doesn't mark relevant messages for deletion when you quit.

delete Deletes a list of messages. (You can reclaim these with **undelete**.)

dt Deletes the current message and prints the next message.

edit Helps edit individual messages using the text editor. Takes a list of messages (as described under **type**) and processes each by writing it into the file "Message*x*" where *x* is the message number being edited, then invoking the text editor on it. (To make **Mail** read the message back and remove "Message*x*", edit the message and execute the editor's *"write and quit"* command.) This command may be abbreviated to **e**.

else Marks the end of the then–part of an **if** statement and the beginning of the part to take effect if the condition of the **if** statement is false.

endif Marks the end of an **if** statement.

exit Leaves **Mail** without updating the system mailbox or the file being read. Thus, if you accidentally delete messages you should have saved, use **exit** to recover.

file The same as **folder**.

folders Lists the names of the folders in your *folder* directory.

folder Switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. With an argument, it writes out changes (such as deletions) you made in the current file and reads the new file. Some special conventions are recognized for the name:

#	Previous file read
%	Your system mailbox
%name	*Name*'s system mailbox
&	Your ~/mbox file
+folder	A file in your folder directory

from Takes a list of messages and prints the header lines for each one. Thus,

```
from joe
```

is the easy way to display all the message headers from *joe*.

headers Lists the headers of all messages that you have. These headers tell you who each message is from, when they were sent, how many lines and characters each message is, and the subject of each message (if a "Subject:" header is present). In addition, **Mail** tags with a "P" the message header of each message that has been the object of the **preserve** command. Messages **saved** or **written** are flagged with an asterisk (*). Those messages **deleted** are not printed at all. To reprint the current list of message headers, use the **headers** command.

Note: If you are using a terminal, **headers** only lists the first few message headers. The number of headers listed depends on the speed of your terminal. This can be overridden by specifying the number of headers you want with the *window option*.

Mail maintains a notion of the current **window** into your messages for the purposes of printing headers. Use the **z** command to move forward and back a window. Move **Mail's** notion of the current window directly to a particular message by using, for example,

headers 40 <RETURN>

to move **Mail's** attention to the messages around message 40. The **headers** command can be abbreviated to **h**.

help Prints a brief help message.

hold Arranges to hold a list of messages in the system mailbox, instead of moving them to the file *mbox* in your home directory. If you set the binary option hold, this happens by default.

if Executes commands in your *.mailrc* file conditionally, depending on whether you are sending or receiving mail. For example, you may specify

```
if receive
commands...
endif
```

An **else** form is also available:

```
if send
commands...
else
commands...
endif
```

Note that the only allowed conditions are **receive** and **send**.

ignore Adds the list of header fields named to the *ignore list*. Does not display header fields in the ignore list when you print a message. This lets you suppress printing of certain machine–generated header fields, such as *Via* which are not usually of interest. **Type** and **Print** display messages in their entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

list Lists the valid **Mail** commands.

mail Sends mail to one or more people. With the *ask* option set, **Mail** prompts you for a subject to your message. You can type in your message, using tilde escapes to edit, print, or modify your message. To send the message, type ↑Z at the beginning of a line, or a period (.) alone on a line if you set the option *dot*. To abort the message, type two interrupt characters (↑I by default) in a row or use the **~q** escape.

mbox Indicates that a list of messages be sent to *mbox* in your home directory when you quit (default action for messages when *hold* option isn't set).

next Goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

next root <RETURN>

goes to the next message sent by *root* and types it. This command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of these magic characters: up–arrow (↑), dot (.), or dollar sign ($). So, dot prints the current message, and "4" prints message 4.

preserve Same as **hold**. Causes a list of messages to be held in your system mailbox when you quit.

quit Leaves **Mail** and updates the file, folder, or system mailbox you were reading. Messages that you have examined are marked "read" and messages that existed when you started are marked "old". If you were editing your system mailbox and if you have set the binary option *hold*, all messages that have not been **deleted**, **saved**, or **mboxed** are retained in your system mailbox. If you were editing your system mailbox and you didn't have *hold* set, all messages which have not been **deleted**, **saved**, or **preserved** are moved to the file *mbox* in your home directory.

reply Frames a reply to a single message. Sends the reply to the originator of a message, plus all the people (except you) who received the original message. You can add people using the **-t** and **-c** tilde escapes. The subject of your reply is formed by prefacing the subject in the original message with "Re:" (unless it was already prefaced with "Re:"). If the original message included a **reply-to** header field, the reply goes *only* to the recipient named by **reply-to.** You type in your message using the same conventions available to you through the **mail** command.

save Lets you save messages on related topics in a file. Takes as argument a list of message numbers, followed by the name of the file in which to save the messages. The messages are appended to the named file, allowing you to keep several messages in the file, stored in the order placed there. The **save** command can be abbreviated to **s**. An example of the **save** command relative to our running example is:

s 1 2 tuitionmail <RETURN>

Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by **next**, unless explicitly specified.

set Sets an option or gives an option a value; used to customize **Mail**. Options can be *binary*, in which case they are on or off, or *valued*. To set a binary option option on, do this:

set *option*

To give the valued option *option* the value *value*, do

set *option=value*

Several options can be specified in a single **set** command.

shell Lets you escape to the shell. Invokes an interactive shell and lets you type commands to it. When you leave the shell, you return to **Mail**. The shell used is a default assumed by **Mail**. You can override this default by setting the valued option *SHELL*, e.g.,

set SHELL=/bin/csh <RETURN>

source Reads **Mail** commands from a file. Helps when you are trying to fix your *.mailrc* file and you need to re-read it.

top Takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**. To change the number of lines to be printed, set the valued option *toplines*. On a terminal, you might prefer to use a line such as this:

set toplines=10 <RETURN>

type Prints a list of messages on your terminal. If you set the option *crt* to a number and the total number of lines in the messages you are printing exceed that specified by *crt*, the messages are piped through **more**(1).

undelete Causes a message **deleted** previously to regain its initial status. Only **deleted** messages may be **undeleted**. Abbreviate this command to **u**.

unset Reverses the action of setting a binary or valued option.

visual Invokes a display–oriented editor. Operates much like the **edit** command. Both the **edit** and **visual** commands assume some default text editors, which can be overridden by the valued options *EDITOR* and *VISUAL* for the standard and screen editors. You might want to do this:

set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi <RETURN>

write Writes only the message itself (i.e., without headers) in the file. This command has the same syntax as **save**. Thus, for example, to write the second message to *file.c*:

w 2 file.c <RETURN>

As suggested by this example, **write** is useful for such tasks as sending and receiving source program text over the message system. It can be abbreviated to **w**.

z **Mail** presents message headers in full windows as described under the **headers** command. You can move **Mail's** attention forward to the next window by giving the

z+ <RETURN>

command. You can move to the previous window with:

z- <RETURN>

5.6 Custom Options

EDITOR This valued option defines the pathname of the text editor to be used in the **edit** command and ~e. If undefined, a standard editor is used.

SHELL This valued option gives the pathname of the shell to be used for the **!** command and ~! escape. In addition, this shell expands filenames with shell metacharacters like asterisk (*) and question mark (?) in them.

VISUAL This valued option defines the pathname of the screen editor to be used in the **visual** command and ~v escape. If undefined, a standard screen editor is used.

append This binary option causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, **Mail** puts messages in *mbox* in the same order that the system puts messages in your *system mailbox*. By setting **append,** you request that new messages be put at the end of *mbox* regardless of the order in which they were received.

ask This binary option causes **Mail** to prompt you for the subject of each message you send. If you respond by typing RETURN, no subject field is sent.

askcc This binary option causes you to be prompted for additional carbon copy recipients at the end of each message. Type RETURN to use the current list.

autoprint This binary option causes the **delete** command to behave like **dp** (after deleting a message, the next one is typed).

debug This binary option causes debugging information to be displayed. It produces the same results as the **-d** command line flag.

dot This binary option, if set, causes **Mail** to interpret a period alone on a line as the terminator of a message you are sending.

escape This valued option lets you change the escape character used when sending mail. Only the first character of the **escape** option is used, and it must be doubled if it is to appear as the first character of a line of your message. Changing your escape character causes tilde (~) to lose its special meaning (and need no longer be doubled at the beginning of a line).

folder The name of the directory used for storing folders of messages. If this name begins with a slash (/), **Mail** considers it an absolute pathname; otherwise, the folder directory is found relative to your home directory.

hold This binary option causes messages that have been read but not otherwise dealt with to be held in the *system mailbox*. This prevents such messages from being automatically swept into your *mbox*.

ignore This binary option causes ↑I characters from your terminal to be ignored and echoed as at signs (@'s) while you are sending mail. All ↑I characters retain their original meaning in **Mail** command mode. Setting the **ignore** option is equivalent to supplying the **-i** flag on the command line.

ignoreeof This option, related to **dot**, causes **Mail** to refuse acceptance of a ↑Z as the end of a message. **Ignoreeof** also applies to **Mail** command mode.

keep This option causes **Mail** to truncate your *system mailbox* instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you would do with the shell command:

% **chmod 600 /usr/spool/mail/***your_login_name* <RETURN>

keepsave This option causes **Mail** to retain all **saved** messages, instead of discarding them as it usually does when you **quit.**

metoo This binary option lets you receive a copy of all messages you send to alias. Unless you set **metoo**, mail sent to an alias in which you are included is not sent to you.

noheader This binary option suppresses the printing of the version and headers when **Mail** is first invoked. Setting this option is the same as using **-N** on the command line.

nosave This binary option prevents **Mail** from copying a partial letter (aborted with two ↑I signals) to the file *dead.letter* in your home directory, which it normally does unless **nosave** is set.

quiet This binary option suppresses the printing of the version when **Mail** is first invoked, as well as the printing of the message number from the **type** command.

record This valued option can be set to the name of a file in which outgoing mail is to be saved. Each new message you send is appended to the end of the file.

screen This valued option specifies how many message headers you want printed. Unless **screen** is set, **Mail** determines the number of message headers to print by looking at the speed of your terminal interface. The faster the baud rate, the more it prints. **Screen** overrides this calculation. The number you set it to also affects scrolling with the **z** command.

sendmail This option alternates the delivery system, when set to the full pathname of an appropriate program for doing this task.

toplines This valued option defines the number of lines that the **top** command prints out instead of the default five lines.

verbose This binary option causes **Mail** to invoke **sendmail** with the **-v** flag, forcing it to go into verbose mode and announce expansion of aliases, etc. Setting this option is equivalent to invoking **Mail** with the **-v** flag.

5.7 Command Line Options

This section describes command line options for **Mail**.

-N Suppress the initial printing of headers.

-d Turn on debugging information.

-f *file* Show the messages in *file* instead of your *system mailbox*. If you omit *file*, **Mail** reads *mbox* in your home directory.

-i Ignore tty interrupt signals. Useful when connecting on noisy phone lines, which may generate spurious interrupt characters.

-n Inhibit reading of */usr/lib/Mail.rc* (not very useful; file is usually empty).

-s *string* Upon sending mail, specify *string* as the subject of the message being composed. (Note: If *string* contains blanks, enclose it in quotation marks.)

-u *name* Read *name*'s mail instead of your own. Essentially, **-u** *user* is a shorthand way of specifying **-f** */usr/spool/user*.

-v Use the **-v** flag when invoking **sendmail**. (This may also be enabled by setting the the **verbose** option.)

The following command line flags are also recognized, but are intended for use by programs (not users) invoking **Mail**:

-T *file* Arrange to print on *file* the contents of the article-id fields of all messages either read or deleted. This option is used by the **readnews** program; you should not use it for reading your mail.

-h *number* Pass on hop count information. **Mail** takes the *number*, increments it, and passes it with **-h** to the mail delivery system. A **-h** is effective only when sending mail and is used for network mail forwarding.

-r *name* When doing network mail forwarding, interpret *name* as the sender of the message. The *name* and **-r** are simply sent along to the mail delivery system. **Mail** waits for the message to be sent and returns the exit status. It also restricts message formatting.

Note that **-h** and **-r**, related to network mail forwarding, are not used in practice since mail forwarding is handled separately.

5.8 Format of Messages

This section describes the format of messages. Messages begin with a *from* line, which consists of the word "From" followed by a user name, followed by anything, followed by a date in the format returned by the **ctime**(3) library routine. A possible **ctime** format date is:

```
Tue Dec  1 10:58:23 1981
```

The **ctime** date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as PDT.

Following the *from* line are zero or more header field lines, each of the form

name: *information*

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id*, *bcc*, *cc*, *from*, *reply-to*, *sender*, *subject*, and *to*. Other header fields are also significant to other systems (see, for example, the ARPANET message standard for more on this topic). A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If headers are present, they must be followed by a blank line. The part that follows is called the body of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, we suggest that this data be encoded in a system which encodes six bits into a printable character. For example, you could use the uppercase and lowercase letters, the digits, and the characters comma and period to make up the 64 characters. Then, you can send a 16-bit binary number as three characters. These characters should be packed into lines, preferably lines about 70 characters long. Long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The **uucp**(1C) message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

Note: Some network transport protocols enforce limits to the lengths of messages.

5.9 Summary of Commands, Options, and Escapes

This section summarizes **Mail** commands, binary and valued options, and tilde escapes.

5.9.1 Commands

!	Single command escape to shell
–	Back up to previous message
Print	Type message with ignored fields
Reply	Reply to author of message only
Type	Type message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
chdir	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see **if**
edit	Edit a list of messages
else	Start of else part of conditional; see **if**
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as **file**
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of **Mail** commands
hold	Same as **preserve**
if	Conditional execution of **Mail** commands

ignore	Set/examine list of ignored header fields
list	List valid **Mail** commands
local	List other names for the local host
mail	Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
quit	Leave **Mail**; update system mailbox, *mbox* as appropriate
reply	Compose a reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive shell
top	Print first so many (5 by default) lines of list of messages
type	Print messages
undelete	Undelete list of messages
unset	Undo the operation of a **set**
visual	Invoke visual editor on a list of messages
write	Append messages to a file, but don't include headers
z	Scroll to next/previous screenful of headers

5.9.2 Options

EDITOR	[*valued*]	Pathname of editor for ~e and **edit**
SHELL	[*valued*]	Pathname of shell for **shell**, ~! and **!**
VISUAL	[*valued*]	Pathname of screen editor for ~v, **visual**
append	[*binary*]	Always append messages to end of *mbox*
ask	[*binary*]	Prompt user for "Subject:" field when sending
askcc	[*binary*]	Prompt user for additional "Cc:"s at end of message
autoprint	[*binary*]	Print next message after **delete**
crt	[*valued*]	Minimum number of lines before using *more*
debug	[*binary*]	Print out debugging information
dot	[*binary*]	Accept a period (.) alone on line to terminate message input
escape	[*valued*]	Escape character to be used instead of a tilde (~)
folder	[*valued*]	Directory in which to store folders

hold	[*binary*]	Hold messages in system mailbox by default
ignore	[*binary*]	Ignore ↑I while sending mail
ignoreeof	[*binary*]	Don't terminate letters/command input with ↑Z
keep	[*binary*]	Don't unlink system mailbox when empty
keepsave	[*binary*]	Don't delete saved messages by default
metoo	[*binary*]	Include sending user in aliases
noheader	[*binary*]	Suppress initial printing of version and headers
nosave	[*binary*]	Don't save partial letter in *dead.letter*
quiet	[*binary*]	Suppress printing of **Mail** version & message numbers
record	[*valued*]	File to save all outgoing mail in
screen	[*valued*]	Size of window of message headers for z, etc.
sendmail	[*valued*]	Choose alternate mail delivery system
toplines	[*valued*]	Number of lines to print in top
verbose	[*binary*]	Invoke **sendmail** with the **-v** flag

5.9.3 Tilde Escapes

~! *command*	Execute shell command
~c *name ...*	Add names to "Cc:" field
~*d*	Read *dead.letter* into message
~e	Invoke text editor on partial message
~f *messages*	Read named messages
~h	Edit the header fields
~m *messages*	Read named messages, right shift by tab
~p	Print message entered so far
~q	Abort entry of letter; like ↑I
~r *filename*	Read file into message
~s *string*	Set "Subject:" field to *string*
~t *name ...*	Add names to To: field
~v	Invoke screen editor on message
~w *filename*	Write message on file
~| *command*	Pipe message through *command*
~ *string*	Quote a tilde (~) in front of *string*

5.9.4 Command Line Flags

-N	Suppress the initial printing of headers
-T *file*	Article-id's of read/deleted messages to *file*
-d	Turn on debugging
-f *file*	Show messages in *file* or *~/mbox*
-h *number*	Pass on hop count for mail forwarding
-i	Ignore tty interrupt signals
-n	Inhibit reading of */usr/lib/Mail.rc*
-r *name*	Pass on *name* for mail forwarding
-s *string*	Use *string* as subject in outgoing mail
-u *name*	Read *name's* mail instead of your own
-v	Invoke **sendmail** with the **-v** flag

Note: The **-T**, **-d**, **-h**, and **-r** flags are for use only by programs that call **Mail**; they are not intended to serve the direct needs of users.

Glossary

Access Rights — These rights list the people who can use each object in the network, and specify how each person can use the object (e.g., permission to read, write, and execute the object).

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 non–interactive *process* that runs immune to quit and interrupt signals issued from your node. In this mode, the 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. (Also see *Process.)*

BSD4.2 — The version of the DOMAIN/IX system that implements 4.2BSD UNIX from the University of California at Berkeley. (Also see *SYSTYPE.)*

C Language — A general purpose low–level programming language used to write programs (e.g., numerical, text processing, and database) and operating systems.

Command — An instruction that you give a program; the name of an executable file that is a compiled program.

Command Argument — A *command option* or the name of the object upon which the command acts. Command arguments follow commands on the same line, although not all commands require an argument. (Also see *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. (Also see *Default.*)

Command Procedure — See *Shell Procedure.*

Command Search Path — The route that the shell takes in searching through various directories for command files. A default search path exists for each of the DOMAIN/IX versions. You may add other directories of executable files which the 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. (Also see *Control Key Sequence.)*

Control Key Sequence — A keystroke combination (<CTRL> followed by another key) used as a shorthand way of specifying commands. To enter a control key sequence, hold <CTRL> down while pressing another key.

Current Directory (.) — The location, within the hierarchical naming tree, of the directory that you are working in at a given time. Entering the UNIX command **pwd** prints the name of your current directory. (Also see *Working Directory.*)

Cursor — The small, blinking box initially displayed in the screen's lower left corner. The cursor marks your current typing position on the screen and indicates which program (shell or DM) receives your commands.

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. (Also see *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. (Also see *File.*)

Disk — A thin, record-shaped plate that stores data on its magnetic surfaces. 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. (Also see *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 log-in 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 System — A high-speed communications network connecting two or more 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 the shell and close the pad in which the shell was running. It is generated by pressing †D and is the same as an ASCII EOT character.

File — The basic named unit of data stored on disk. A file can contain a memo, manual, program, or picture. (Also see *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 the 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. (Also see *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.

Hard Link — A link that points directly to an object (file).

Here Document — A command procedure of the form "command << eofstring" which causes the 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
Input Window — A *pad* that accepts commands typed at your keyboard. 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. (Also see *Shell Procedure.*)

Link A special type of object that points from one place in the naming tree to another. (Also see *Hard Link* and *Symbolic Link.*)

Link Text The name of the object contained in a symbolic link to show what is being linked. When you use a link name as a pathname or as part of a pathname, UNIX Shells substitute the link text for the link name. (Also see *Symbolic Link.*)

Logging In Initially signing on to the system so that you may begin to use it. This creates your first user process.

Main Memory The node's primary storage area. It stores the instruction that the node is executing, as well as the data it is manipulating.

Memory Any device that can store information.

Metacharacter See *Shell Metacharacter.*

Mode An absolute mode is an octal number used in conjunction with the UNIX **chmod**(1) command to change permissions of files.

Name A character string associated with a file, directory, or link. A name can include various alphanumeric characters, but never a slash (/) or null character. Remember that certain characters have special meaning to the 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 DOMAIN 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. (Also see *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. (Also see *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' 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. (Also see *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. (Also see *Pathname.*)

Partner Node — A node that shares its disk with a diskless node. (Also see *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. (Also see *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. (Also see *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. (Also see *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 the 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. Regular expressions are specified by allowing certain characters special meaning to the 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). (Also see *Shell Command*.)

Secondary Prompt — A notification to the user that the command typed in response to the primary prompt is incomplete. By default, a "> " is the secondary prompt used by UNIX shells.

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. (Also see *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 the 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 the shell 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 the 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 the 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". (Also see *Script.*)

Symbolic Link — A link that points to link text or the pathname of an object (file). Sometimes also known as a "soft link". (Also see *Link.*)

System Administrator — The person responsible for system maintenance at your site.

Sys5 — The version of the DOMAIN/IX system that implements UNIX System V, Release 2, from AT&T Bell Laboratories. (Also see *SYSTYPE.)*

SYSTYPE — A DM environment variable that shows the UNIX system version currently in use. Valid SYSTYPES for DOMAIN nodes are "sys5" and "bsd4.2". (Also see *DM Environment Variable.*)

Super–User — See *System Administrator.*

Terminal — See *Node.*

Transcript Pad — A transcript pad contains a record of your interaction with a process. The process output window provides a view of its transcript pad. The term "screen" found in some of our documentation also refers to the transcript pad of the window in which a shell is running.

User Account

The system administrator defines a user account for every person authorized to use the system. Each user account contains the name the computer uses to identify the person (user ID), and the person's password. User accounts also contain project and organization names, helping the system determine who can use the system, and what resources they can use. (Also see *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. (Also see *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. (Also see *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 the shell may assign string values.

Wildcards

Special characters that you may use to represent one or more pathnames. (Also see *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. (Also see *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. (Also see *Insert Mode*.)

Working Directory

The default directory in which a process creates or searches for objects. (Also see *Current Directory*.)

Index

Primary page references are listed first. The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index.

Symbols

A

B

C

D

E

F

H

I

J

K

L

M

N

P

R

S

T

U

V

W

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: *DOMAIN/IX User's Guide*
Order No.: 005803 Revision: 01 Date of Publication: December, 1986

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other
_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the DOMAIN system?_____

What parts of the manual are especially useful for the job you are doing?

_____

What additional information would you like the manual to include?

_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)

_____

_____
Your Name Date

_____
Organization

_____
Street Address

_____
City State Zip

No postage necessary if mailed in the U.S.

cut or fo..long dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *DOMAIN/IX User's Guide*
Order No.: 005803 Revision: 01 Date of Publication: December, 1986

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the DOMAIN system?_____

What parts of the manual are especially useful for the job you are doing?

_____

What additional information would you like the manual to include?

_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)

_____

_____
Your Name Date

_____
Organization

_____
Street Address

_____
City State Zip

No postage necessary if mailed in the U.S.

cut or tear along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD

DOMAIN/IX USER'S GUIDE
ORDER NO. 005803 - REV. 01

INSTRUCTIONS FOR PLACING TABS IN BINDER

NAME	PLACE BEFORE PAGE NO.
DOMAIN/IX Overview	1-1
Introduction to Shells	2-1
Using the Bourne Shell	3-1
Using the C Shell	4-1
Using BSD4.2 Mail	5-1