# DOMAIN/IX User's Guide

**Order No. 005803**
**Revision 00**
**Software Release 9.0**

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

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

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

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

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

THIS SOFTWARE AND DOCUMENTATION ARE BASED IN PART ON THE FOURTH BERKELEY SOFTWARE DISTRIBUTION UNDER LICENSE FROM THE REGENTS OF THE UNIVERSITY OF CALIFORNIA.

© 1985 Apollo Computer Inc. All rights reserved.
Printed in U.S.A.
First Printing: July 1985

This document was formatted using the **troff** text formatter distributed with DOMAIN®/IX™ software.

---

APOLLO and DOMAIN are registered trademarks of Apollo Computer Inc.
AEGIS, DGR, DOMAIN/IX, DPSS, DSEE, D3M, GMR, and GPR are trademarks of Apollo Computer Inc.

# PREFACE

The DOMAIN®/IX™ *User's Guide* and its companion volume, the *DOMAIN/IX Text Processing Guide* consist of those papers normally included in Volumes 2A, 2B, and 2C of the UNIX† *Programmer's Manual* as supplied by Bell Telephone Labs and the University of California at Berkeley. The papers in these books have been revised where necessary to reflect the DOMAIN system environment. However, we have tried to remain aware of the history of UNIX as a multiuser system, and have included the more important references to operations conducted at terminals.

## Audience

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 introductions if you are not already familiar with UNIX.

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

This document also assumes a basic familiarity with the DOMAIN system. The best introduction to the DOMAIN system is *Getting Started With Your DOMAIN System* (Order No. 002348). 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.

## The Structure of This Document

This manual is divided into four sections and an appendix.

**Section 1** provides an introduction to DOMAIN/IX and explains how to install the SR9 DOMAIN/IX software.

**Section 2** discusses shells. Chapter 1 is an overview of the shells available to the DOMAIN/IX user. Chapter 2 is an introduction to the Bourne shell. Chapter 3 is an introduction to the C shell.

---

† UNIX is a trademark of AT&T Bell Laboratories.

**Section 3** deals with the communications programs **mail** and **uucp**.

**Section 4** deals with the following support tools.

- awk — a pattern matching tool
- sed — a stream editor
- lint — a C program checker
- make — a program for maintaining other programs
- lex — a lexical analyzer
- yacc — a compiler compiler
- sccs — a source code control system

**Appendices** includes papers on The C Programming Language, **Ratfor**, the **M4** Macro Processor, the **bc** and **dc** programs, and the **sendmail** and **curses** packages.

### Related Volumes

The *DOMAIN/IX User's Guide* (this book) is the first volume you should read. It explains how DOMAIN/IX works, and contains extensive material on the Bourne shell, C shell, and the communications utilities **mail** and **uucp**.

The *DOMAIN/IX Text Processing Guide* (Order No. 005802) describes the UNIX text editors (ed, ex, and vi) supported by DOMAIN/IX. 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 DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005799) describes all the UNIX System V system calls and library functions supported by the *sys5* version of DOMAIN/IX.

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 DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for BSD4.2* (Order No. 005801) describes all the BSD4.2 UNIX system calls and library functions supported by the *bsd4.2* version of DOMAIN/IX.

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 two-volume *DOMAIN System Call Reference* (Volume I Order No. 007196, Volume II Order No. 007194) describes calls to operating system components that are accessible to user programs.

### Documentation Conventions

Unless otherwise noted in the text, this manual uses the following 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 Output returned by programs or commands is shown in Roman type.

[optional] Square brackets enclose optional items in formats and command descriptions.

**...** Horizontal ellipses indicate that the preceding item can be repeated one or more times.

**name**[*x*] Single numbers or numbers and letters enclosed in brackets immediately following the name of a UNIX command or library function refer to the section where you can find reference information on the command or function in the *DOMAIN/IX Command Reference* or the *DOMAIN/IX Programmer's Reference*.

↑*x* A control character, where *x* is the character.

SMALL CAPS We use small caps for acronyms and key names; e.g., ASCII and RETURN . Note that in tutorial material, we place a box around the name of a key.

*filename* We use italics to represent generic, or meta- names in example command lines, and also to represent a character that stands for another character, as in **d***x* where *x* is a digit. In text, the names of files written or read by programs are set in italics.

### 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*. Refer to the CRUCR (Create User Change Request) command. You can also get more information by typing:

**/com/help crucr**

in any UNIX or AEGIS shell. There is a Reader's Response form at the back of this manual. We'd appreciate it if you would take the time to fill it out when you're ready to comment on this document.

SECTION 1

# GETTING STARTED

CONTENTS

# Chapter 1: An Overview of DOMAIN/IX

## 1.1 INTRODUCTION

DOMAIN/IX (pronounced "domain eye ex") is an implementation of the UNIX operating system that runs on DOMAIN nodes. It supports the DOMAIN distributed file system, ring network, and 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, DOMAIN/IX offers many features that are seldom found on either time-sharing or workstation implementations of this software.

There are two versions of DOMAIN/IX. The *sys5* version is compatible with UNIX System V Release 2 from AT&T Bell Laboratories, and the *bsd4.2* version is compatible with 4.2 BSD, from the University of California at Berkeley. You may install either or both at your site.

In this chapter, we provide

- an introduction to those DOMAIN system features that are not found in other UNIX systems,
- an explanation of the way in which DOMAIN/IX allows you to use the *bsd4.2* and *sys5* UNIX versions concurrently, and
- information about other features of DOMAIN/IX (e.g., character mapping, compiler output, and other details primarily of interest to programmers).

We also provide pointers to additional reference materials on DOMAIN architecture.

Other sections of this *User's Guide*, comprised largely of papers written by the developers of UNIX System V and 4.2 BSD, deal with the various UNIX shells we support, with the more prominent implements in the UNIX tool kit (e.g., **awk**, **lex**, **make**, and so on), and with the communications utilities mail and uucp. A companion volume, the *DOMAIN/IX Text Processing Guide*, covers the text editors **ed**, **ex**, and **vi**, the **troff** and **nroff** formatters, and their related preprocessors and macro packages.

## 1.2 AN OVERVIEW OF DOMAIN ARCHITECTURE

In this section, we explain the fundamental concepts of the DOMAIN system. Readers who are already familiar with DOMAIN systems may skip this section.

### 1.2.1 The DOMAIN System

A DOMAIN system is comprised of two or more nodes connected by a high-speed (12Mbit/sec.) network. The network has a ring topology, and uses a token-passing protocol to prevent collisions between messages being sent from one node to another. 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 are 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 stand-alone computers, and can support additional users through serial communications ports. (This subject is covered in more detail in Chapter 2.)

In order to take advantage of this networked architecture, all DOMAIN/IX software includes support for our distributed file system. Data and programs on all mounted volumes in the ring 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 with no degradation in performance. All users (excluding those who access the system via a tty device) have the power of a dedicated processor, memory-management hardware, and a high-resolution bit-mapped display at their disposal.

**Note:** For more information on DOMAIN architecture, refer to *Getting Started With Your DOMAIN System*, the *DOMAIN System User's Guide*, and *Administering Your DOMAIN System*. These volumes are shipped with all DOMAIN nodes.

### 1.2.2 The User Interface

From the user's point of view, the interface to DOMAIN/IX differs from the UNIX interface described in many of the papers in this book, principally because the screen of a DOMAIN node can display "windows" into many processes (shells, programs, and so on). These windows have some unique features not found on the "dumb" terminals commonly used by the machines on which UNIX System V and 4.2 BSD were developed.

### 1.2.3 DOMAIN/IX and AEGIS

DOMAIN/IX is co-resident with the DOMAIN system's AEGIS operating system. Since they use many of the same underlying kernel functions, DOMAIN/IX and AEGIS are tightly integrated. As a result:

- the UNIX programs supplied with DOMAIN/IX have the same file format as AEGIS programs
- DOMAIN/IX UNIX shells can coexist on the same screen with AEGIS shells
- UNIX commands can be executed by an AEGIS shell
- AEGIS commands can be executed by a UNIX shell

There is normally 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. There are only a few cases where naming conflicts (UNIX and AEGIS programs that have the same name) may make it necessary for you to rename or alias a command.

## 1.3 THE DISPLAY AND THE DISPLAY MANAGER

In this section, we provide a brief introduction to your DOMAIN node's display and keyboard, placing an emphasis on those unique DOMAIN features that, properly used, help make your job easier. Readers who are already experienced users of DOMAIN nodes should feel free to skip to the next section.

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 enable you to 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.

### 1.3.1 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 tell it that 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 provides a running transcript of the standard output. On the theory that it is unwise (possibly 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 that word is used in many of the papers in this book. In addition, it has features that go far beyond what most terminals can manage.

As we mentioned, an input pad is actually special instance of an edit pad. It can't be made read-only, and it will "grow" 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. At this point, you can issue the DM command **wc** (window close, normally mapped to ↑N) to remove the window from the screen.

**Note:** You can save the information contained in a transcript pad in either of two ways.

- You can copy all or part of the pad into an edit pad, paste buffer, or file (see Section 1, Chapter 4 of the *DOMAIN/IX Text Processing Guide* for details on how this is done.)
- You can use the DM's **pn** (pad name) command to write the pad to a disk file.

Edit pads do not interact with programs at all; 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 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. The *DOMAIN System User's Guide* has more detailed information on pads, windows, and window legends.

### 1.3.2 Default Windows and Shells

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

user logs 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. Depending on the value of the environment variable *UNIXLOGIN*, this may be either the standard UNIX

login

prompt or the slightly different AEGIS prompt

Please Log In:

(We explain more about environment variables in a later section.) After you log in, the DM input pad displays the

Command:

prompt. Pressing the CMD key brings the cursor to the DM input window.

The DM output pad (which is actually 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 opens an AEGIS shell when you log in, then executes your personal login script of DM commands. You may, of course, arrange for the DM to open a UNIX shell instead.

### 1.3.3 DM Commands

We have already mentioned a few of the many DM commands. Since all DM commands are covered in detail in the *DOMAIN System Command Reference*, we will only touch lightly on the subject here. Those DM commands that you are most likely to use when editing text or examining a transcript are covered in somewhat greater depth in Section 1, Chapter 4 of the *DOMAIN/IX Text Processing Guide*. All DM commands have several things in common:

- They can be entered in the DM input window. Press the CMD (command) key to bring the cursor to the DM's

Command: prompt, then enter the command line.

- They can be placed in a command file for execution as needed (e.g., when you log in).
- They can be bound to DOMAIN keyboard keys using the DM's **kd** (key definition) command.

### 1.3.4 Regions

Some DM commands deal with the whole screen. Most, however, deal with an individual window, or even with a region within the pad a window opens on. Since the concept of a screen that is divided into regions may be new, we offer an introduction to the topic here.

Whenever you move the cursor to the DM input window (by pressing the CMD key), the DM first notes the cursor's location on the screen. That way, it can derive such information as the current working directory of a shell, the location of the cursor in an edit pad, or the current location of a window you intend to MOVE or GROW. The same is true when you press a key that has been defined to invoke a DM command sequence. For example, when you press the EDIT key, the DM first notes the current working directory of the shell window that the cursor last occupied. If you type

edit file: **foo**

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 an obscured — or partially obscured — window. By operating in this manner, the DM is able to provide services to all processes running in windows on your node.

### 1.3.5 Moving the Cursor

Probably the most fundamental DM command is "move the cursor." While there are a number of ways to get the DM to move the cursor, the arrow keys at the left of the keyboard are, for most people, the most intuitively obvious. Many keyboards are also equipped with a mouse or a touchpad, both of which are effective tools for large-scale cursor movements. In addition, the mouse has three programmable function keys. Read the *DOMAIN System User's Guide* for more information on the mouse and touchpad.

There are even explicit DM commands that 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 time to execute these commands.) In addition, the special keys CMD and NEXT WNDW move the cursor to the DM input window and the next unobscured shell input pad or read/write edit pad respectively.

**Note:** The DM considers a window to be obscured if any part of it is covered by another window. If there are no unobscured shell windows or read/write edit pads on the display, NEXT WNDW has no effect. A read-only edit pad is also not a candidate for NEXT WNDW.

Chapter 2 of the *DOMAIN System Command Reference* details all of the DM commands supported at SR9. Remember, shell commands won't work in the DM window.

### 1.3.6 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 at any time, it is usually best to begin with the default bindings, then "customize" your key definitions as needed. For more information on the DM and keyboard mapping, see Chapter 2 of the *DOMAIN System Command Reference*.

The DOMAIN system supports two types of keyboards: the 880 (high-profile) keyboard and the low-profile keyboard.

**Note:** The majority of nodes are equipped with the low-profile keyboard.

The directory */sys/dm* contains the command files that define both keyboards:

*std_keys* keyboard definitions for the 880 keyboard

*std_keys2* keyboard definitions for the low-profile keyboard

### 1.3.7 UNIX Key Definitions

Alternate versions of the standard key definitions — modified to provide necessary UNIX functions — reside in the following files in */sys/dm*.

*unix_keys* Generic UNIX keyboard definitions for the 880 keyboard

*unix_keys2* Generic UNIX keyboard definitions for the low-profile keyboard

*att_keys* System V UNIX keyboard definitions for the 880 keyboard

*att_keys2* System V UNIX keyboard definitions for the low-profile keyboard

*bsd_keys* BSD 4.2 UNIX keyboard definitions for the 880 keyboard

*bsd_keys2* BSD 4.2 UNIX keyboard definitions for the low-profile keyboard

The *bsd4.2* and *sys5* key definitions files include commands that bind various keys to certain version-specific (or shell-specific) features. They will be described in detail in Section 2 of this manual, in the chapters that deal with the Bourne Shell and the C Shell. Initially, none of these key definitions files will be automatically invoked, although you may arrange for them to be, as we shall explain. To put any key definitions file into effect, execute the DM command

Command: **cmdf** *filename*

where *filename* is one of those listed above. For example, to invoke the generic UNIX key definitions on a low-profile keyboard, type:

Command: **cmdf /sys/dm/unix_keys2**

When the keyboard is remapped by a *unix_keys* file, the following keys get new definitions:

- The SHELL key executes the DM command

  **cp /bin/start_sh**

  which invokes a Bourne Shell and runs your *.profile*. (See Section 2 of this manual for more on shells and *.profile*.) Normally, SHELL invokes an AEGIS shell (*/com/sh*).

- When shifted, the TAB key inserts a literal ASCII tab character. Normally, TAB merely moves the cursor a predetermined number of spaces to the right, and ↑ TAB moves the cursor a predetermined number of spaces to the left.

- ↑I sends a UNIX interrupt signal. Normally, ↑I has no special function.

- The READ key and the EDIT key both insert (invisible) quotation marks around any pathname you supply to their prompts. (The quotation is required in order to pass mapped characters to the DOMAIN system's naming server.) The prompts themselves are also subtly different. READ generates

  read file:

  (rather than "Read file:"), and EDIT generates

  edit file:

  (not "Edit file:").

## 1.4 DM Environment Variables

UNIX users should be familiar with the concept of environment variables. These process-wide ASCII strings 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 ones.

You will typically want to initialize these variables in one of the command files that the DM reads when the node is booted and — later — when a user logs in.

**Note:** Environment variables extant 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.

For processes that use multiple program levels, environment variables are mark-released so that, while a new program level inherits all environment variables from a previous level, a new level cannot 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. 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) or **ce** (edit file) are used, for reasons explained later in the section on variant links.

Environment variables defined in the DM startup file are inherited by all server processes created during DM startup, as well as by the first process created by a user at login.

**Note:** After the first user process is created, environment variables are inherited by the DM from the current context process (and passed to new processes) as described above.

A program interface for environment variable usage is defined in the files */sys/ins/ev.ins.?**. 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, others have special significance to system software or other special attributes. Environment variables that are defined by the system at login time are referred to as "predefined" variables. The subset of predefined variables that cannot be deleted or changed by **/com/sh** or any callers of **ev_$...** are referred to as "privileged."

In the following list, predefined variables are flagged with a † and privileged variables (which are also predefined) are flagged with a ‡.

*USER* ‡ — the user's login name

*LOGNAME* ‡ — is synonymous with *USER*. The synonyms are provided to support both versions of DOMAIN/IX.

*PROJECT* ‡ — is the project (group) ID under which the user logged in.

*ORGANIZATION* ‡ — is the organization ID under which the user logged in.

*NODEID* ‡ — is the unique node identifier for the node on which the process is running, expressed in hexadecimal.

*NODETYPE* ‡ — is the type of node on which the process is running.

*HOME* ‡ — is the user's home directory path name, established at login time.

*TERM* † — is the device name of the "terminal" in use. We define this variable for the sake of C or UNIX programs that have a terminal dependency. Values for our displays are

| | |
|---|---|
| apollo_15P | 15 inch portrait display |
| apollo_19L | 19 inch landscape display |
| apollo_color | DN460/660 color display |
| apollo_800_color | DN550 color display |

*TZ* † — 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* — is the UNIX system version in use.

*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. (There is an extended discussion of *UNIXLOGIN* in a later section of this chapter.)

| | |
|---|---|
| *UNIXNAMES* | specifies the name mapping scheme to use. Meaningful values are SR8 and SR9. UNIX name mapping changes are described in detail in a later section of this chapter. |
| *NAMECHARS* | specifies a set of characters to which special semantics are attached during name translation. This variable is meaningful only when using SR9 name mapping. It is described in detail later in this chapter. |

## 1.5 INSTALLING DOMAIN/IX

There are a number of options available when installing DOMAIN/IX. Chapter 2 of this section covers these options in detail. It's short, and we suggest that both node administrators and system administrators read it carefully before installing DOMAIN/IX. Further information about installation is available in the UNIX release notes.

DOMAIN/IX can be installed with either single-version or multiple-version UNIX support. The type of installation affects the availability of features described in this manual and the other manuals shipped with DOMAIN/IX. Before attempting to access a program or command described in the DOMAIN/IX user documentation, find out which version(s) of UNIX are installed at your site.

## 1.6 SUPPORT FOR MULTIPLE UNIX VERSIONS

The two versions of the UNIX operating system that DOMAIN/IX supports provide a variety of similar — though seldom identical — system services through kernel and library functions. It is frequently the case that, while function *x* exists in both the *sys5* and the *bsd4.2* environments, the semantics of the function and, in some cases, even its arguments may be subtly different.

As an illustration, consider the kernel function **setpgrp**[2]. In Bell Systems 3 and 5, the function definition is

```
int setpgrp ()
```

It is defined 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." In 4.1 BSD and 4.2 BSD, there is an identically-named function with similar semantics but a different calling sequence. The Berkeley function

```
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 UNIX is written with the assumption that the run-time environment will be UNIX software of a certain lineage (Bell or Berkeley) or even a specific version (Bell System V or 4.2 BSD). 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. Here's how it works.

At compile time, you select the version of UNIX for which your program is targeted. This version selector is called the *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* that was 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.

DOMAIN/IX supports the following *SYSTYPE*s.

| | |
|---|---|
| *sys5* | Bell System 5 release 2 |
| *bsd4.2* | Berkeley 4.2bsd |
| *sys3* | Bell System 3, provided for backward compatibility. |
| *bsd4.1* | Berkeley 4.1bsd, provided for backward compatibility |
| *any* | Declares that the program is independent of a particular UNIX version (highly unlikely). |

There are several ways to express a systype to the C compiler. You may include it in the source file itself by using the #systype directive (supported by the DOMAIN C compiler) followed by one of the values listed above. If you use #systype, it must be the first non-comment statement in the source. For example;

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

The systype may also be specified on the compiler command line. For **/com/cc** (the DOMAIN C compiler), this takes the form **-systype** *value*. For **/bin/cc** (the DOMAIN/IX interface to **/com/cc**), it takes the form **-T***value*. For example, in the AEGIS shell, say

```
$ cc berkprog.c -systype bsd4.2
```

In the C Shell, it would be

```
% cc -Tsys5 bellprog.c
```

If you specify one *systype* on the command line and a different one in the file, the compiler will object. 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, if you wanted to compile all programs to run in a *sys5* (Bell System 5) environment, you would set *COMPILESYSTYPE* (in a *sys5* Bourne Shell) as follows.

```
# COMPILESYSTYPE=sys5
# export COMPILESYSTYPE
```

In a C Shell, the line would be:

```
% setenv COMPILESYSTYPE sys5
```

As long as *COMPILESYSTYPE* was thus set, all C programs would be 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* (UNIX System III).

### 1.6.1 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 with differing versions of UNIX. To support identically named Bell and Berkeley versions of these directories on the same DOMAIN file system, we have introduced "symbolic," or "variant" links. Unlike regular links, symbolic 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 allow programs to use either the *sys5* or *bsd4.2* versions of the */bin*, */etc*, and */usr* directories (*/tmp* and */dev* are, of course, common to both). Although the links to */bin*, */usr*, and */etc* are normally created by the installation script, you may at some point need to create (or re-create) such links yourself using the **/com/crl** command. For example, to create a *SYSTYPE*-dependant link for */bin*, use the command line below.

```
% /com/crl /bin '$(systype)/bin'
```

**Note:** You must use the AEGIS command **/com/crl** here. The UNIX command **ln** won't work. Note also that the single quotes around the link text are required. Otherwise the dollar sign will be 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. The top-level DOMAIN/IX directory organization is as follows:

| Name | Object Type | Major Subdirectories |
|---|---|---|
| */usr* | symbolic link | - |
| */bin* | symbolic link | - |
| */etc* | symbolic link | - |
| */dev* | normal link | - |
| */bsd4.2* | directory | */usr, /bin, /etc* |
| */bsd4.1* | directory | */usr/include* |
| */sys5* | directory | */usr, /bin, /etc* |
| */sys3* | directory | */usr/include* |
| */tmp* | link | |

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 will get the proper include files, but it means that when you invoke a *sys3* or *bsd4.1* environment for interactive use, you will not be getting the "old" versions of, for example, commands and macro packages.

### 1.6.2 Environment Switching

The object-module stamping scheme, described earlier, allows you to execute System 5 programs from a 4.2 BSD C Shell and vice versa, without any knowledge of the UNIX version for which the program was targeted. When you invoke a program that is 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 will continue to work when executed from a process running in another version. The program **/etc/systype** 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 called **ver**[1]. You can use **ver** in three ways.

[1] Typing **ver** with no arguments displays the current value of *SYSTYPE*.

[2] Typing **ver** *value* changes *SYSTYPE* to **value**, thereby changing the version of subsequently executed commands.

[3] Finally, **ver** *value command* executes the *value* version of *command* but does not change *SYSTYPE*.

For example

```
% ver                                  What version am I using?
bsd4.2
% ver sys5 id                         Execute the sys5 version of id
uid=212(kate) gid=38(unix)
% ls                                   Execute the bsd4.2 version of ls
prog.c  prog.o  testfile
% ver sys5                            set SYSTYPE to sys5
% ls do an ls
prog.c
prog.o
testfile
%
```

### 1.6.3 Getting Help

You can obtain information about available UNIX commands, system calls, and functions with the **man** command. This command allows you to select and display on-line versions of reference material from the *DOMAIN/IX Command Reference* and the *DOMAIN/IX Programmer's Reference*. For example, to display the manual page for the command **who**, type

**man who**

in any UNIX shell. The **man** command will then open a read window containing a formatted version of the manual page(s) on the **who** command. See the information on the DM editor in Section 1, Chapter 4 of the *DOMAIN/IX Text Processing Guide* for more information on how to scroll through and search for patterns 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 CTRL N to close the window.

**Note:** The **man** command, like all DOMAIN/IX UNIX commands, uses the symbolic links in effect for the *SYSTYPE* of the shell in which it is executed. When **man** is executed in a shell that has a *SYSTYPE* of *sys5*, manual pages will come from */sys5/usr/catman*. When **man** is executed in a shell that has a *SYSTYPE* of *bsd4.2*, manual pages will come from */bsd4.2/usr/man*.

In addition, the directory */usr/docs* contains the source text for the papers, tutorials, and articles included in the *DOMAIN/IX User's Guide* and the *DOMAIN/IX Text Processing Guide*. The file */usr/docs/read_me* includes brief instructions for printing these files at your site.

## 1.7 OTHER FEATURES OF DOMAIN/IX

In this section, we have collected miscellaneous facts that you may need to know, especially if you are developing applications software to run on your DOMAIN system.

### 1.7.1 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 line 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 login name can be typed without a preceding "l"
- the rejection message is changed to "login incorrect"
- an acceptance message read from */etc/motd*. is displayed on the output device. If the user is logging in to the DM, only the last line of this file will be displayed due to space limitations (one line) in the DM output window. If */etc/motd* isn't found, a standard AEGIS login acceptance message, minus the "project" and "organization" fields, is issued.

**Note:** If */etc* is a variant link (the usual case), *SYSTYPE* must be set to *bsd4.2* or *sys5* or else the DM will not be able to locate */etc/motd*.

### 1.7.2 The Process Model

Both UNIX System V and 4.2 BSD 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 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* which, when **set**, specifies in-process execution (the standard DOMAIN process model) and when **unset** specifies the traditional *bsd4.2* process model.

**Note:** The default value of *inprocess* is unset.

There are advantages and disadvantages to each process model. In the Chapter 3 of Section 2 of this manual (about the C Shell), we supply

details about the use of *inprocess*, including a summary of these advantages and disadvantages.

### 1.7.3 Filename Mapping

While the DOMAIN/IX kernel is inherently case-insensitive, the set of characters that may be used in both AEGIS and UNIX pathnames has been greatly extended. DOMAIN/IX component names may contain any ASCII character except slash and null. All ASCII printing characters are legal in AEGIS (stored) component names except as noted below.

| | |
|---|---|
| Illegal as first character | space, tilde, slash, backslash, dot, tic (grave accent) |
| Illegal as any character | space, slash, backslash |

If you need to retain the ability to access file system objects that are named under the old rules, set the *UNIXNAMES* environment variable to the value "sr8", as shown below, to use SR8 name mapping.

```
# add this line to your
# 'node_data/startup
# file to use SR8 name mapping rules
env UNIXNAMES 'sr8'
```

If *UNIXNAMES* is not present (or not set to 'sr8'), SR9 name mapping rules are used.

**Note:** To ensure compatibility with existing products, all files on the DOMAIN/IX SR9 distribution media are named under the SR8 rules. This means that every site must initially set *UNIXNAMES* to sr8.

When you're ready to convert SR8 names to the new rules, run the program **/etc/cvtumap** by typing

**/etc/cvtumap -9** *pathname(s)* **[-l]**

**Cvtumap** converts components in *pathname(s)* from the SR8 to SR9 mapping scheme (or vice versa). If the **-l** switch is included, **cvtumap** will list the names of files it has remapped. After running this program on trees containing any mapped names, you may delete *UNIXNAMES* or set it to SR9.

Under either set of naming rules, mapping is transparent to DOMAIN/IX users. You may include any legal character in a component name simply by typing it. However, uppercase alphabetics and certain other characters are stored as two-character escape sequences, and component names are limited to 32 characters, including any escape characters that may be required. A component name that consisted exclusively of uppercase alphabetics, to cite an extreme case, would be limited to 16 characters, since each character would be stored as a two-character escape sequence.

The table below shows how filename mapping works. Any characters not listed in the first column of the table are passed unchanged to the

DOMAIN naming server. Note that some characters require an escape only if they are used as the first character of a component name.

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

In addition to the mapping rules summarized above, 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 would be mapped as

:ab:#03

As we mentioned above, any time 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. For some examples, refer to the following table.

| 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 |

AEGIS shells will display 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 will have to escape it with a colon when you create the name.

By default, the SR9 mapping scheme causes all characters except slash and null to be mapped and stored in component names. You may use the *NAMECHARS* environment variable to specify that any or all of the following characters should retain special meanings when read by the naming server. The characters available are

tilde home directory (or “naming directory”)

backslash parent directory

grave accent "this node" (e.g., *'node_data*)

For example, to retain the ability to access the naming directory with a leading tilde, the parent directory with a backslash, and to reference 'node_data, you would set *NAMECHARS* to the string ~ '\ by including the following line in a DM startup file.

```
# add this line to your
# /sys/dm/startup or user_data/startup
# file to use preserve the special significance
# of tilde, grave accent, and backslash.
env NAMECHARS '~ \''
```

We recommended that, in cases where references using special characters are coded into programs, a network-wide standard be established for the value of NAMECHARS. For programs that are intended to be transported to other networks or systems, special care must be taken with respect to this feature.

### 1.7.4 Password and User Identification

Although the process of login verification and home-directory setting are always handled by the DOMAIN system's login mechanism, we provide a means of generating an */etc/passwd* file so that those UNIX programs that need to access it will find what they expect to find there. In order to be sure that new users at your site have accounts on both the DOMAIN network registry and in */etc/passwd*, a system administrator must invoke this program, called **crpasswd**[1M], each time a new user account is added to the network registry. You may use **cron**[1] to run **crpasswd** on a daily basis. It will invoke **crpasswd** with a user-id of *root*.

### 1.7.5 File Protection

The normal protection mechanism in the DOMAIN environment is the access control list (ACL). Every object (file, directory, and so on) has an ACL associated with it. At SR9, the ACL mechanism has been extended to include support for all of the UNIX operating system's access modes, 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, files, pipes, and directories created with **creat**[2], **mknod**[2], and **open**[2] are assigned an ACL that corresponds to the value of the mode specified in the call, modified by the current **umask** value.

**Note:** If an object's ACL specifies more than one "group" owner, its UNIX access mode will show 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.

### 1.7.6 Read, Write, and Execute Rights

It is a characteristic of the DOMAIN system's single-level store that file system objects must be readable if they are to be executable or writable. When a file is created with **creat**[2], it will be readable and writeable by the owner, regardless of any mode specified with the **creat**. Use **chmod**[1] to change these permissions if necessary. However, if you use **chmod** to make a file "execute only" or "write only" for owner or any group, the "read" bits will also be turned on. See the example below.

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

### 1.7.7 Output from the C Compiler

The DOMAIN/IX C compiler uses the DOMAIN common code generation mechanism. It produces a non-standard *a.out* file.

### 1.7.8 Debugging

DOMAIN/IX does not support the **adb** and **sdb** debuggers. Use the DOMAIN Language Level Debugger (DEBUG) in place of these utilities. See the *DOMAIN Language Level Debugger Reference* (Order No. 001525) for a description of DEBUG.

### 1.7.9 Library Organization

The DOMAIN/IX and DOMAIN C products are contained in two global libraries that are automatically loaded at system startup. These libraries, (*/lib/unixlib* and */lib/clib*) are global. These libraries are mapped into the node's address space at run time, and all program globals are resolved by the loader. Libraries that are not loaded at run time will require binding.

### 1.7.10 The Process Environment Flag

The DOMAIN system maintains a per-process flag that describes the environment in which the process is running. This flag is used to identify processes running in the DOMAIN/IX environment and to control how access rights are applied to any objects that are created. When the flag is set, system calls use filename mapping and protect objects according to the access mode value supplied to system calls such as **open**, **creat**, and **mknod**.

### 1.7.11 Ownership of Files

When you create a file using the DM editor (see the *DOMAIN/IX Text Processing Guide*), UNIX programs will see it as owned by "root" until you explicitly specify another owner of the file using the **chown**[1] command. In this case, ownership is assigned to "root" only because the real owner can't be determined. You will not have to log in as "root" in order to change the ownership of these files. Once ownership has been assigned, it will not be affected by further editing with the DM editor.

**Note:** It is especially important to recognize this phenomenon when using the DM editor to create *.login*, *.cshrc* and *.profile* files, since UNIX shells only read these files if they are owned by the person opening the shell.

### 1.7.12 Networking Software

The *bsd4.2* version of DOMAIN/IX includes full support for sockets and other ingredients of Berkeley networking software. With BSD4.2 you can use utilities such as **rsh**[1], **rlogin**[1], and **telnet**[1] to communicate with other nodes on the ring. To support such usage, the *bsd4.2* version of DOMAIN/IX includes a limited implementation of DOMAIN TCP/IP. Unless you also have the DOMAIN COM-ETH product, you will not be able to use these utilities to communicate with other hosts (e.g., over an Ethernet). The TCP/IP implementation provided with DOMAIN/IX TCP supports communication on the DOMAIN ring only.

We ship the *DOMAIN TCP/IP Reference Manual* (order no. 003247) with the documentation for the bsd4- 2 version of DOMAIN/IX. It includes information you will need in order to configure DOMAIN nodes to use the limited implementation of TCP/IP with DOMAIN/IX. Appendix D of that manual provides special information for DOMAIN/IX users who do not have the COM-ETH product.

## 1.8 HOW TO FORMAT ONLINE DOCUMENTS

The files in the */usr/doc* directory are the ones from which the *User's Guide* and *Text Processing Guide* were made. If you want to troff these files at your site, remember that:

- All files that end in *.mm* automatically source in the files */usr/lib/macros/mmt* (**-mm** macros for **troff**) and */usr/lib/macros/mmtx* (DOMAIN extensions to *mmt*). Files with the *.mm* extension may not **nroff** especially well, and contain numerous constructs optimized for the IMAGEN CX laser printer. The hardcopy versions of these files were prepared by piping **troff** output through the **catdvi** and **dviimp** postprocessors from IMAGEN.
- All files that end in *.ms* or *.me* were **troff**ed using the **-ms** or **-me** macros, respectively, and are in largely original form.

Subdirectories under */usr/docs* are *tg* (*Text Processing Guide*) and *ug* (*User's Guide*). The "refer" and "sendmail" documentation files are kept in their own directories; */usr/doc/tg/refer.dir* and */usr/doc/ug/sendmail.dir*.

# Chapter 2: Installing DOMAIN/IX

## 2.1 INTRODUCTION

There are two versions of DOMAIN/IX.

- DOMAIN/IX *sys5*, based on release 2 of UNIX System V from AT&T Bell Laboratories.
- DOMAIN/IX *bsd4.2*, based on 4.2BSD from the University of California at Berkeley.

You may install either or both on your node. In this chapter, we provide detailed information on how to install DOMAIN/IX using the installation script, and how to customize your installation after the script has been run.

DOMAIN/IX software may be distributed on any of three types of media.

- magtape
- cartridge tape
- 8-inch floppy disks

Once a system administrator has installed the file system on a node (or nodes) on the network, users may install DOMAIN/IX on their own nodes.

### 2.1.1 Terms

We use the following terms when describing the installation procedure.

- The "Work node" is the node on which you enter the commands that do software installation. The work node must be running standard SR9 software.
- The "Target node" or target volume is the disked node whose software you are installing or updating. The target node must be running standard SR9 software. The target node and the work node may be the same node.
- The "Source node" or source volume is a disked node on which the DOMAIN/IX file system has already been installed.

### 2.1.2 What Happens During Installation

Even though the actual installation process is handled by a script, it may be useful to know — in a general way — what happens during installation.

DOMAIN/IX makes use of several library files. These files (or links to them) will be installed in the */lib* directory of your node.

*/lib/clib* These are the C language library functions.

*/lib/unixlib* These are additional UNIX functions that may be distributed to UNIX licensees only.

In addition, installation will place the following directories (or links to these directories) in the entry directory of your node.

*/bin* This directory includes the executable (binary) files for most of the UNIX commands.

*/etc* This directory contains system administration files (e.g., */etc/passwd*, the password file) and other files that may need to be customized for your installation.

*/usr* This contains library functions, on-line documentation, and a variety of other programs that perform system services (e.g., **uucp**, the dictionary used by **spell**, and macro packages used by **troff**).

**Note:** Both the DOMAIN C Language product and DOMAIN/IX put files in the directory */usr/include* as part of the installation process. Where identically-named include files are installed by both products, they are guaranteed to have identical functionality. This means that you may install either product (DOMAIN C or DOMAIN/IX) first without running a risk of corrupting the contents of */usr/include* when the other is installed.

Copies of the */bin* and */usr* directories may exist on each node. However, installation normally creates links to such "public" areas as */usr/spool* and */usr/lib/uucp*. It also normally creates a link to a central copy of */etc*.

The installation process also creates a link to */tmp* as shown below.

```
$ crl /tmp 'node_data/tmp
```

This allows a */tmp* directory for every diskless node that is partnered with your node as well as a */tmp* of your own. While this may place added demands on disk resources, it will ensure that there are no naming conflicts in the temporary files built in */tmp* by, for example, **troff**.

When all of the required links, libraries, and files have been placed on your node, you will be prompted to shut down and restart the node. This allows the new libraries to be mapped into the node's address space.

### 2.1.3 Types of Installation

There are two types of installations: administrative and user. The administrative install procedure installs a complete DOMAIN/IX file system on a node (or nodes) in the network. This ensures that those files and directories that need specific access rights are set up correctly, and that spool directories and other "public" areas of the file system will only

exist in one place. Subsequent user installs allow individuals to mount a subset of the file system on their own nodes. Before any user can run a user install, the administrative install must be complete.

## 2.2 RUNNING THE ADMINISTRATIVE INSTALL

To run the administrative install, you must have *sys_admin* rights. That is to say, you will need to have read/write and add/delete rights to all public parts of the file system, as well as rights to add/change ACL's on all public file system objects. The actual installation is handled by an interactive AEGIS shell script that must first be copied from the distribution media onto a target node.

A system administrator should use the following procedure to install DOMAIN/IX software from the distribution media.

[1] Use the AEGIS command **wd** (WORKING_DIRECTORY) to set your work node's working directory to the entry directory of the target node.

$ **wd** *//target_node*

[2] Use the AEGIS command **rbak** (READ_BACKUP) command to get the *install_sysadmin* script from the distibution media.

$ **rbak -dev** *d* **-f 1 install_sysadmin -as install_domain_sysadmin -ms -l**

where *d* is one of **ct** for cartridge tape (DN550 only), **m** for magtape, or **f** for floppy disk.

[3] When the installation script has been copied to your disk, use the **wd** command to set your work node's working directory to the */install* directory on the target node.

$ **wd** *//target_node*/**install**

[4] Type

$ **install_sysadmin**

to run the administrative installation script. The script is interactive. It will present you with options and let you select the one(s) appropriate to the type of installation you're doing. It will also let you restart a partially-completed install.

[5] After the installation is complete, the following message will appear.

DOMAIN/IX SR9 INSTALLATION COMPLETE

** PLEASE SHUTDOWN, RESET, AND RESTART THE TARGET NODE **

Before shutting down the target node, examine the transcript pad for error messages. If any errors occurred during the installation, repeat the installation procedure.

[6] If no errors occurred, shut down the target node by pressing the CMD key and typing the DM command

Command: **shut**

The DM will exit and the message

SHUTDOWN SUCCESSFUL

will be displayed. At this point, type

**RE**

followed by two carriage returns. This will cause the Mnemonic Debugger (MD) to display its prompt. Restart the node by typing

> **EX AEGIS**

at the MD prompt.

[7] After you have restarted the target node, log in and run the **/install/fix_cache** program.

$ **/install/fix_cache**

[8] Edit the template in */install/preserve* as described in the next section.

### 2.2.1 The User Install Template

The administrative install sets up templates to which subsequent user installs refer when creating links to file system objects. These templates will be on the target node's disk in the files */preserve/install/domainix_template_sys5* and */preserve/install/domainix_template_bsd4.2*. A typical template for a site at which both versions have been installed — on a target node named //ice — is shown below.

- *//ice/preserve/install/domainix_template_bsd4.2*:

```
NETWORK CONFIGURATION:  both
BIN: //ice
ETC: //ice
USR/ADM: //ice
USR/BIN: //ice
USR/DICT: //ice
USR/DOC: //ice
USR/INCLUDE: //ice
USR/LIB: //ice
USR/LIB/UUCP: //ice
USR/MAN: //ice
USR/SPOOL: //ice
USR/UCB: //ice
SR8: //ice
#
```

- *//ice/preserve/install/domainix_template_sys5*:

  NETWORK CONFIGURATION: both
  BIN: //ice
  ETC: //ice
  USR/BIN: //ice
  USR/CATMAN: //ice
  USR/DOC: //ice
  USR/INCLUDE: //ice
  USR/LIB: //ice
  USR/LIB/UUCP: //ice
  USR/NEWS: //ice
  USR/PUB: //ice
  USR/SPOOL: //ice
  USR/MAIL: //ice
  SR8: //ice
  #

If you (as an administrator) need to move parts of the file system to other nodes, you'll have to edit these templates so that they reflect the changes you have made. The edited versions below show that the man pages for both versions have been moved to node //doc, and that /usr/spool has been moved to node //beanbag.

- *//ice/preserve/install/domainix_template_bsd4.2*:

  NETWORK CONFIGURATION: both
  BIN: //ice
  ETC: //ice
  USR/ADM: //ice
  USR/BIN: //ice
  USR/DICT: //ice
  USR/DOC: //doc
  USR/INCLUDE: //ice
  USR/LIB: //ice
  USR/LIB/UUCP: //ice
  USR/MAN: //doc
  USR/SPOOL: //beanbag
  USR/UCB: //ice
  SR8: //ice
  #

- *//ice/preserve/install/domainix_template_sys5*:

```
NETWORK CONFIGURATION:  both
BIN: //ice
ETC: //ice
USR/BIN: //ice
USR/CATMAN: //doc
USR/DOC: //doc
USR/INCLUDE: //ice
USR/LIB: //ice
USR/LIB/UUCP: //ice
USR/NEWS: //ice
USR/PUB:  //ice
USR/SPOOL: //beanbag
USR/MAIL:  //beanbag
SR8: //ice
#
```

**Note:** When you copy a system directory of file from one node to another, remember to use the **-sacl** option to **cpt** (or **cpf**) so that the object's acl's are retained during the copying process.

## 2.3 THE USER INSTALL

Once the administrative install is complete, the system administrator should inform the other users at the site that DOMAIN/IX is available for general use. Individual users can install those parts of the file system that they need on their own nodes by running the **install** script and selecting the DOMAIN/IX option.

This script is highly interactive. It allows you to select from a menu of available choices, gives you the option of creating links to (rather than copies of) various file system objects, and guarantees that individual nodes will have the necessary links to **/usr/spool/uucp** and other files and directories that must be shared.

## 2.4 FILES, LINKS, AND DIRECTORIES

The following two tables are derived from the installation script. They summarize the size of major file system objects that are associated with each version of DOMAIN/IX. In addition, they include our recommendation as to what form the object should take on the average user's disk.

| *sys5* | | |
|---|---|---|
| Name | Size (blocks) | Recommended installation |
| *bin* | 1490 | as local directory |
| *usr/bin* | 3060 | as local directory |
| *usr/catman* | 1730 | as link |
| *usr/doc* | 2120 | as link |
| *usr/include* | 170 | as local directory |
| *usr/lib* | 2200 | as local directory |
| *usr/pub* | 3 | as local directory |

| *bsd4.2* | | |
|---|---|---|
| Name | Size (blocks) | Recommended installation |
| *bin* | 1220 | as local directory |
| *usr/adm* | 2 | as local directory |
| *usr/bin* | 2430 | as local directory |
| *usr/dict* | 350 | as link |
| *usr/doc* | 2120 | as link |
| *usr/include* | 170 | as local directory |
| *usr/lib* | 1700 | as local directory |
| *usr/man* | 2800 | as link |
| *usr/ucb* | 2150 | as local directory |

Installation always makes links to the following file system objects.

- */usr/spool/uucp*
- */usr/spool/uucppublic*
- */usr/spool/mail*
- */etc*

Where disk space must be conserved, it is possible to link to any (or — if necessary — all) of the major file system objects associated with DOMAIN/IX. There are no specific performance penalties attached to, for example, having /bin as a link rather than a resident object on your disk.

### 2.4.1 SYSTYPE and Symbolic Links

A key ingredient in the DOMAIN/IX formula for supporting multiple versions of UNIX is the *variant link* or *symbolic link*. A symbolic link is a link that resolves differently for different values of the environment variable SYSTYPE. The prototype command line for creating a symbolic link is

**/com/crl** *name '/$(systype)/name'*

When the DOMAIN naming server resolves this link, it substitutes the current value of *systype* into the space occupied by *$(systype)*. To take a specific case, most installations would use symbolic links of the form

$(systype)/bin
$(systype)/etc
$(systype)/usr

in place of "real" */bin*, */etc*, and *usr* directories. References to any of these objects can then be resolved to links (or, if desired, actual directories) named */bsd4.2*, */sys5*, or any other legal *SYSTYPE*, in the entry directory of the node.

**Note:** You cannot use the UNIX command **ln**[1] to create symbolic links. You may, however, use **/com/crl** UNIX and AEGIS shells. The installation process will create the necessary symbolic links based on the options you choose when running the install script.

The installation process will create the necessary symbolic links based on the options you choose when running the *install* scripts.

# Index

## SECTION 2

# SHELLS

CONTENTS

# Chapter 1: An Overview of Shell Types

## 1.1 INTRODUCTION

The DOMAIN/IX user has three types of shells available: two UNIX Shells and a third shell, called the AEGIS Shell, that is the standard shell used by the DOMAIN system's AEGIS operating system. (DOMAIN/IX users who have both the *sys5* and *bsd4.2* versions installed will actually have two versions of the Bourne Shell available.) These shells all offer such features as I/O redirection, pipes, shell procedures (scripts), and metacharacters (wildcards). And while most of these similar features have the same underlying function, the details of implementation differ — markedly, in some cases.

In this chapter, we will point out the subtle differences between these shells, our aim being to alert you to those things that, while similar on the surface, work a little differently in each shell. We will also provide a very brief discussion of the AEGIS Shell and tell you where to look for additional information. Chapters 2 and 3 of this section are detailed explanations of the features of the Bourne and C Shells, respectively. The AEGIS Shell is covered in detail in the *DOMAIN System User's Guide*.

**Note:** Before reading any further in this section, please read Section 1 of this manual. It contains important information about installing and using DOMAIN/IX. Unless you read and understand this information, you may not be able to take full advantage of the features of all our UNIX Shells.

## 1.2 UNIX SHELLS

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

### 1.2.1 Opening a Default UNIX Shell

Most DOMAIN/IX users will want to have a UNIX Shell created automatically whenever they log in. You may arrange to have the DM (Display Manager) open a UNIX shell:

- whenever any user logs in to the node
- whenever you log in to a node

If you want every user of a node 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, the file is */sys/node_data/startup_login.type* (where *type* is the type of display the node has.)
- For a diskless node the file (on the partner node) is */sys/node_data.xxxx_startup*, where *xxxx* is the node ID of your node.

Individual users who want to get a UNIX shell when they log in should edit their own *user_data/startup_login.type*.

The file below is for a node that has a 19-inch landscape display (e.g., a DN 320). It will be executed whenever anyone logs in to this node. We have added lines to it that create a process running */bin/start_sh* and one that runs */bin/start_csh*. 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.)

**Note:** In this file, we make the important assumption that the environment variable *SYSTYPE* has already been set (in the DM command file *'node_data/startup.type*. We provide instructions for doing this in Section 1, Chapter 2 of this manual.

```
# 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
#
# Now 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 that 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 (hash mark) 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.

### 1.2.2 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. There are several ways to do this. If you have invoked one of the key definitions files discussed in

Section 1, you may simply press the (shifted) SHELL key. The *unix_keys* and *sys5_keys* key definitions 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 the SHELL key. The *unix_keys* file normally includes the line

kd l5s cp /bin/start_sh ke

which opens a login Bourne Shell (*/bin/start_sh*). If you would prefer to have SHELL open a C Shell instead, change this line to

kd l5s cp /bsd4.2/bin/start_csh ke

which specifies that */bsd4.2/bin* be used, and that the **/start_csh** program be invoked.

As an alternative to using the SHELL key, 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 the CMD key and enter the DM command

Command: **cp /bin/start_sh**

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

Command: **cp /bin/start_csh**

When you do this, the display manager creates the specified shell process in a window with a transcript pad and input pad. As we described in the previous section, the *SYSTYPE* inherited from the most recent cursor position will determine the */bin* that is used. You may also specify */sys5/bin* or */bsd4.2/bin* to force creation of a shell with a given systype.

### 1.2.3 Using a Terminal

If you need to access a DOMAIN node via a tty device (an ASCII terminal), use the following procedure to create 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/sio***n*

where n is the number of the SIO (Serial Input/Output) line to which the terminal is connected.

You may achieve the same effect by going to the DM input window and typing

Command: **cpo /bin/start_sh /dev/sio***n*

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 the following SIO line characteristics:

| **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 will cause hangup fault |

The last close of the SIO line will cause 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, refer to the **TCTL** command in the *DOMAIN System Command Reference*.

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

In the C Shell, the following definitions will be in effect.

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

In the Bourne Shell, the definitions are:

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

**Note:** No suspend character is enabled for the Bourne Shell.

When you log in via the siologin process, the shell you get 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
```

## 1.3 DIFFERENCES BETWEEN UNIX AND AEGIS SHELLS

There are several differences between the AEGIS and UNIX Shells that the DOMAIN/IX user should keep in mind. There are differences in the following areas:

- command search rules

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

A program is considered to be running in the AEGIS environment if it has been invoked in an AEGIS Shell and in a UNIX environment if it is invoked in a UNIX Shell, Keep this in mind during the discussions of metacharacters and command search rules that follow.

**Note:** Almost all AEGIS commands reside in the */com* directory.

### 1.3.1 Command Search Rules

Please read the material on *SYSTYPE* and multiple version support in Section 1, 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. Any UNIX Shell will start by looking in:

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

You can change the default search rules in any of our UNIX Shells by setting the shell variable called PATH. This topic is covered in detail in the chapters about the individual shells.

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

1. Working directory.
2. Personal command directory, ~*com*
3. AEGIS command directory, */com*

AEGIS Shells do not 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 cannot execute it from a UNIX shell.

DOMAIN/IX users 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.

### 1.3.2 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 below.

*#!/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* to specify a *bsd4.2* Bourne Shell.

*#!/sys5/bin/sh* to specify 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
```

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 will be interpreted — with unpredictable results — by the shell in which it was invoked.

### 1.3.3 Wildcards

Every shell has its own metacharacters (wildcards). Chapters 2 and 3 of this section detail the wildcard-handling mechanisms of the Bourne and C Shells. The *DOMAIN System Command Reference* has complete information on AEGIS Shell wildcards. We will not elaborate on the differences here, except to state that the differences are important — even among the various UNIX Shells.

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 2 and 3 of this section for more.
- If you're executing a UNIX command in an AEGIS Shell, do not use any wildcards.

### 1.4 Differences in Valid Pathnames

As we mentioned in Section 1, Chapter 1, there are some differences 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 (**open**[2], **creat**[2], **chdir**[2] and so on) so that you can specify pathnames which contain a greater variety of characters than those allowed in the AEGIS environment.

All DOMAIN/IX UNIX commands 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.

### 1.5 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 will 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. In order to support job control in the C Shell, we have added a shell variable that determines the process model used by that shell.

#### 1.5.1 The inprocess Variable

A C Shell 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. The default value of *inprocess* is **unset**, which means that the C Shell will always fork a new process to run a new command. For more on *inprocess*, see Chapter 3 of this section.

#### 1.5.2 Changes of Working Directory

Changes in the current working directory made with the AEGIS **wd** command are not effective across program invocations. 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 2: An Introduction to the Bourne Shell

## 2.1 INTRODUCTION

The UNIX Shell (often referred to as the *Bourne Shell*, in honor of its inventor, S. R. Bourne) is a language that provides a programmable interface to DOMAIN/IX. 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 file names or flags, may be passed to a command. Commands set a return code that may be used to determine control-flow. The standard output from a command may be used 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 sections of this chapter cover most of the everyday requirements of shell users. Later sections describe those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell.

### 2.1.1 Special Key Definitions

Read the material in Section 1 of this manual about key binding and key definitions files. It explains how the keys on DOMAIN node keyboards are bound to the functions they execute.

In this chapter, we assume that you have invoked one of the *sys5* key definitions files. These files bind various keys to functions that the Bourne Shell provides. To invoke these definitions, press the CMD key and enter the following DM command.

Command: **cmdf /sys/dm/***file*

where file is either

- *sys5_keys* if you have an 880 (high-profile) keyboard,
- *sys5_keys2* if you have a low-profile keyboard.

Special key definitions (in addition to those provided in *unix_keys*) included in this file are:

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

The *sys5_keys2* has the above definitions along with one other:

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

### 2.1.2 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
```

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

```
# ls -l
```

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

### 2.1.3 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 command and accept additional input before the command finishes. For example,

```
# cc pgm.c&
```

calls the C compiler to compile the file *pgm.c*. The trailing **&** 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. A list of currently active processes may be obtained using the **ps**[1] command.

### 2.1.4 Input/Output Redirection

Most commands produce output on the standard output — normally the screen.

**Note:** People using DOMAIN nodes should note that in this chapter, the term "terminal" is used 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*

The notation >*file* is interpreted by the shell and is not passed as an argument to **ls**. If *file* does not exist, the shell creates it; otherwise, the original contents of *file* are replaced with the output from **ls**. Output may also be appended to a file using the notation

**ls -l** >>*file*

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*

The command **wc** 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

**wc -l** <*file*

could be used.

### 2.1.5 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.

\# **ls -l | wc**

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

**ls -l** >*file***; wc** <*file*

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

Many UNIX commands are referred to as "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**, selects from its input those lines that contain some specified string. For example,

\# **ls | grep old**

prints those lines, if any, of the output from **ls** that contain the string

*old*. Another useful filter is sort. For example,

```
# who | sort
```

will print an alphabetically sorted list of logged-in users.

A pipeline may consist of more than two commands, for example,

```
# ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string old.

### 2.1.6 Generating Filenames

Many commands accept filenames as arguments. For example,

```
# ls -l main.c
```

prints information relating to the file *main.c* .

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
# ls -l *.c
```

generates, as arguments to **ls**, all filenames in the current directory that end in **.c**. When used in this context, the asterisk is a metacharacter "pattern" that will match 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 minus will match any character lexically between the pair.

For example,

| | |
|---|---|
| **[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 file name is found that 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
```

finds and prints the names of all files of the form *filename.bin* in sub-directories of **/usr/fred**. (The **echo** 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.

```
# echo *
```

will therefore echo all filenames in the current directory not beginning with "**.**".

```
# echo .*
```

will echo all those file names that begin with '**.**' . This 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:** As we have mentioned, AEGIS commands perform their own wildcard expansion, using rules different from those used by the Bourne Shell. Unquoted wildcards used in the Bourne Shell will be expanded according to the Bourne Shell's rules, then passed to whatever command is being executed. If you are 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 commands.

### 2.1.7 Quotation

As we have mentioned, characters that have a special meaning to the Shell are called metacharacters. Some of the more common Bourne Shell metacharacters are listed below.

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

There is a complete list of Bourne Shell metacharacters at the end of this chapter. Any character preceded by a \ is said to be "quoted" and loses any special meaning it may otherwise have had. Since the \ is elided, **echo**, used as shown, would return the following strings

```
# echo \?
?
# echo \\
\
```

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

quotes. For example,

```
# echo xx´****´xx
xx****xx
```

The quoted string may not contain the single quote character (') but 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, will be discussed in a later section.

### 2.1.8 Prompting

The shell issues a prompt when it is ready for more input. The default *sys5* Bourne Shell prompt is the sequence # (# followed by a space). The default *bsd4.2* Bourne Shell prompt is the sequence **B$** (B$ followed by a space). Either prompt may be changed by saying, for example,

```
# PS1=yesdear
```

that sets the prompt to be the string *yesdear*.

If a newline is typed and further input is needed, then the shell will issue the secondary prompt "> ". If this happens unexpectedly, type an interrupt to return the main shell prompt. You may change this prompt as well. The line

```
# PS2=nodear
```

would have the expected effect.

### 2.1.9 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 System* and the *DOMAIN System User's Guide* for more detailed information). It will normally open an AEGIS Shell, then look 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 like

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

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

**Note:** In the example line above, we specified

```
/sys5/bin
```

as the */bin* to use. See the information on multiple version support (Section 1, Chapter 1) for more on this.

You may also define a key or function key to open a Bourne Shell. The following DM command defines the shifted L5 key — L5 is labelled SHELL — so that when you press SHIFT SHELL a Bourne Shell will be opened.

**kd l5s cp /bin/start_sh ke**

In this case, since no */bin* is specified, the **start_sh** command is obtained from */$(SYSTYPE)/bin*.

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. Any file called *.profile* in your home directory is assumed to contain commands and is read by the shell first, before reading commands from the terminal or any other file. Every Bourne Shell you start will read from this file.

**Note:** If you use the DM editor to create your *.profile*, you must also use the UNIX command **chown**[1] to make yourself the owner of your *.profile*. Otherwise, it will not be read.

## 2.2 SHELL PROCEDURES

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

**sh file** *[argument(s)]*

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**

is equivalent to

\# **who | grep fred**

Files have three independent attributes, *read*, *write*, and *execute*. The UNIX command **chmod**[1] may be used to make a file executable. For example,

\# **chmod +x wg**

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

\# **wg fred**

is equivalent to

```
# sh wg fred
```

This allows shell procedures and programs to be used interchangeably.

As well as providing names for the 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**. A typical use of this is to provide some default arguments, as in,

```
# nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

### 2.2.1 Control Flow Using for

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

```
tony 5890
bob 3303
sherry 4368
        ...
        ...
richard 5335
```

If this file is called **/usr/lib/telnos**, then the text of the shell procedure, which we'll call **tel**, is

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

The command line

```
# tel sherry
```

prints those lines in */usr/lib/telnos* that contain the string sherry.

```
tel sherry richard
```

prints those lines containing *sherry* followed by those for *richard*.

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

```
# create alpha beta
```

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

### 2.2.2 Control Flow Using case

The Bourne Shell's **case** statement provides a multiway branching mechanism. For example,

```
#! /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

**append** *file*

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

**append** *file1 file2*

appends the contents of *file1* 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 * is the pattern that matches any string, it can be used for the default case.

**Note:** No check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed.

In the example below, the commands following the second * will never be executed.

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

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc** 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 | . For example,

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

is equivalent to

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

The usual quoting conventions apply, so that

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

will match the character **?**.

### 2.2.3 Here Documents

The shell procedure **tel**, illustrated in the previous section, uses the file **/usr/lib/telnos** to supply the data for **grep**. An alternative is to include this data within the shell procedure as a "here document."

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

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

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*

is then equivalent to the **ed** commands

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

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character **$** as in

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

(This version of **edg** is equivalent to the first except that **ed** will print a **?** if there are no occurrences of the string **$1**.) Substitution within a here document may be prevented entirely 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.

### 2.2.4 Shell Variables

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

**Note:** You may examine all variables that are currently set by typing the **set** command.

Variables may be given values by writing, for example,

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

which 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 **$**; for example,

```
# echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

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

will move the file *pgm* from the current directory to the directory */usr/fred/bin*. 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
```

will direct the output of **ps** to the file */tmp/psa*, whereas,

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

would cause the value of the variable **tmpa** to be substituted.

Except for **$?**, which is set after every command, the Bourne Shell sets the variables below when it is 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 file names. 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 a special meaning to the shell and should be avoided for other uses.

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

**$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 it was 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. For example,

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

**$MAILCHECK** [*sys5*]

Specifies how often (in seconds) the shell will check to see if you have mail. The default value is 600 seconds. If $MAILCHECK is set to 0, the shell will check before each prompt.

**$MAILPATH** [*sys5*]

A colon-separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail*.

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

**$HOME** The default argument for the **cd** command. The current directory is used to resolve file name references that do not begin with a /, and is changed using the **cd** command. For example,

# **cd /usr/fred/bin**

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

# **cd $HOME**

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

**$PATH** A list of directories that contain commands. Each time a command is executed by the shell, a list of directories is searched for an executable file. If **$PATH** is not 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**

specifies that the current directory (the null string before the first **:** ), **/usr/fred/bin**, **/bin** and **/usr/bin** are to be searched in that order. In this way, individual users can have their own "private" commands that are accessible independently of the current directory. If the command name contains a /, then this directory search is not used; a single attempt is made to execute the command.

**$PS1** The primary shell prompt string, by default, '# '.

**$PS2** The shell prompt when further input is needed, by default, "> ".

**$IFS** The set of characters used by blank interpretation.

### 2.2.5 The test Command

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

**test -f** *file*

returns zero exit status if *file* 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 *file* true if *file* exists

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

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

test -d *file* true if *file* is a directory

### 2.2.6 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 round the loop *command-list* is executed; if a zero exit status is returned, then *command-list* is executed; otherwise, the loop terminates. For example,

```
#! /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 some external event occurs, then run some commands. In an **until** loop, the termination condition is reversed. For example,

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

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually

create the file.)

### 2.2.7 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 in conjunction 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 example is the **touch** command which changes the "last modified" time for a list of files. The command may be used in conjunction 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 is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable **flag** is set to some non-null string if the **-c** argument is encountered. The commands

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

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then command2
fi
```

may be written

*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.

### 2.2.8 Command Grouping

Commands may be grouped in two ways,

**{** *command-list***; }**

and

**(** *command-list* **)**

In the first example, *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
# (cd x; rm junk )
```

executes **rm junk** in the directory **x** without changing the current

directory of the invoking shell.

The commands

```
# cd x; rm junk
```

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

### 2.2.9 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
```

(**v** for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

**sh -v** *proc*

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

**Note:** Saying **set -n** at a terminal will render the terminal useless until an end-of-file is typed.

The command

```
# set -x
```

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

```
# set -
```

and the current setting of the shell flags is available as **$-**.

### 2.2.10 The man Command

The **man** command is used to print sections of the *DOMAIN/IX Command Reference* and *DOMAIN/IX Programmer's Reference* manuals. When you type

```
# man sh
```

the *Command Reference* page for **sh**[1] is displayed in a DM read window. Since no section is specified, section 1 is used. Typing

```
# man 2 fork
```

displays the **fork**[2] manual page (from Chapter 2 of the *DOMAIN/IX Programmer's Reference*). When the **man** command was originally written, it was a shell procedure like the one below.

```
#! /bin/sh
cd /usr/man
: ´colon is the comment command´
: ´default is nroff ($N), section 1 ($s)´
N=n s=1
for i
do case $i in
      [1-9]*) s=$i ;;
      -t) N=t ;;
      -n) N=n ;;
      -*) echo unknown flag \´$i\´ ;;
      *) if test -f man$s/$i.$s
       then ${N}roff man0/${N}aa man$s/$i.$s
       else : ´look through all manual sections´
       found=no
       for j in 1 2 3 4 5 6 7 8 9
       do if test -f man$j/$i.$j
            then man $j $i
                 found=yes
            fi
       done
       case $found in
            no) echo ´$i: manual page not found´
       esac
       fi
      esac
done
```

**Note:** When the shell script above is executed on a DOMAIN/IX system, */usr/man* normally resolves to */$(SYSTYPE)/usr/man*, allowing **man** to select the manual page for the appropriate systype.

## 2.3 KEYWORD PARAMETERS

**Note:** The *sys5* and *bsd4.2* Bourne Shells differ somewhat in their method of handling parameter substitution. These differences will be noted as they apply.

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 is not affected. For example,

**user=fred** *command*

will execute *command* with **user** set to *fred*. 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 positional parameters **$1**, **$2**, **...** .

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
# set - *
```

will set **$1** to the first file name in the current directory, **$2** to the next, and so on. Note that the first argument, **-**, ensures correct treatment when the first file name begins with a - .

### 2.3.1 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. For example,

```
# export user box
```

marks the variables **user** and **box** for export. When a shell procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not 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.)

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.

### 2.3.2 Parameter Substitution (*bsd4.2*)

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

```
# echo $d
```

or

```
# echo ${d}
```

will echo nothing. A default string may be given as in

```
# echo ${d-.}
```

which will echo the value of the variable **d** if it is set and "**.**" otherwise. The default string is evaluated using the usual quoting conventions so

that

# **echo${d-´*´}**

will echo ***** if the variable **d** is not set. Similarly

# **echo ${d-$1}**

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

# **echo ${d=.}**

which substitutes the same string as

# **echo ${d-.}**

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

If there is no sensible default, then the notation

# **echo ${d?message}**

will echo the value of the variable **d** if it has one; otherwise, *message* is printed by the shell and execution of the shell procedure is abandoned. 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?}
```

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 then the shell will abandon execution of the procedure.

### 2.3.3 Parameter Substitution (*sys5*)

In the *sys5* version of */bin/sh*, there are 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 known as variables) may be assigned values as follows.

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

No pattern-matching is performed on *value*. There cannot be a function and a variable with the same *name*.

| | |
|---|---|
| ${*parameter*} | The value, if any, of the parameter is substituted. 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*; the value of the parameter is substituted. Positional parameters may not be assigned in 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, the message "parameter null or not set" is printed.

${*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`}
```

If you omit the colon from the above expressions, the shell only checks 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.

### 2.3.4 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`
```

is equivalent to

```
# d=/usr/fred/bin
```

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

```
# ls `echo "$1" `
```

is equivalent to

```
# ls $1
```

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

```
# basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

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

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

| | |
|---|---|
| for i in `ls -t`; do ... | The variable **i** is set to the names of files in time order, most recent first. |
| set `date`; echo $6 $2 $3, $4 | will print the date, for example<br>1984 Dec 14, 23:59:59 |

### 2.3.5 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 "A 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

  ```
  # echo $X
  ```

  will echo **$y**.
- Blank interpretation. Following the above substitutions, the resulting characters are broken into non-blank words. For this purpose, "blanks" are the characters of the string **$IFS**. By default, this string consists of blank, tab and newline. The null string is not regarded as

a word unless it is quoted. For example,

```
# echo ´´
```

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

```
# echo $null
```

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

- Filename generation. Each word is then scanned for the file pattern characters *****, **?** and **[...]** and an alphabetical list of file names is generated to replace the word. Each such file name 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.

As well as the quoting mechanisms described earlier using \ and **´...´**, 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 \.

| | |
|---|---|
| $ | parameter substitution |
| ` | command substitution |
| ” | ends the quoted string |
| \ | quotes the special characters $ ` ” \ |

For example,

```
# echo ”$x”
```

will pass the value of the variable **x** as a single argument to **echo**. Similarly,

```
# echo ”$*”
```

will pass the positional parameters as a single argument and is equivalent to

```
# echo ”$1 $2 ...”
```

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

```
# echo ”$@”
```

will pass the positional parameters, unevaluated, to echo and is equivalent to

```
echo ”$1” ”$2” ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated. In the table below:

- 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.

| Quote | Metacharacter | | | | | |
|---|---|---|---|---|---|---|
| | \ | $ | * | ` | " | ´ |
| ´ | n | n | n | n | n | t |
| ` | y | n | n | t | n | n |
| " | y | y | n | y | t | n |

Among other things, the table above shows that the sequence \$ is not interpreted (is passed as a literal $), the sequence \ can be used to terminate a string, and the sequence "$ will preserve the meta-meaning of $. In cases 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

```
# eval echo $X
```

will echo the string *pqr*.

In general, the eval command 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. For example,

```
# wg=´eval who|grep´
# $wg fred
```

is equivalent to

```
# who|grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as | , following substitution.

### 2.3.6 Error Handling

The treatment of errors detected by the shell depends on the type of error and on 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 may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally.
- The command terminates normally but returns a non-zero exit status.

In all of these cases, the shell will go on to execute the next command. Except for the last case, an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return 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.

The following list covers many of the UNIX signals used by DOMAIN/IX. For a complete list, see **signal**[2].

**1** hangup
**2** interrupt
**3*** quit
**4*** illegal instruction
**5*** trace trap
**6*** IOT instruction
**7*** EMT instruction
**8*** floating point exception
**9** kill (cannot be caught or ignored)
**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])

The signals in this list of potential interest to shell programs are 1,2, 3, 14, and 15.

### 2.3.7 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap ´rm /tmp/ps$$; exit´ 2
```

sets a trap for signal 2 (terminal interrupt). If this signal is received, it will execute the commands

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

**Exit** is another built-in command that terminates execution of a shell procedure. The **exit** command is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background, then **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command. The cleanup action is to remove the file *junk*$$ .

```
#! /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, it would be possible for the process to 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 is taken from the **nohup** command.

```
trap ´ ´ 1 2 3 15
```

which causes *hangup*, *interrupt*, *quit*, and *kill* to be ignored both 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) is an example of the use of **trap** where there is no exit in the **trap** command, **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**. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

### 2.3.8 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. The built-in command **exec** creates a new program level in the shell process. For example, a simple version of the **nohup** command looks like

```
trap ´ ´ 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands and **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 so that, for example,

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

will write 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 the file *file* which is created if it does not already exist.

>> *file* The standard output is sent to file *file*. If the file exists, then 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*.

<< *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, then no interpretation of the document occurs. If *file* is not quoted, then parameter and command substitution occur and \ 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.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

*command* ... 2>*file*

runs *command* with message output (file descriptor 2) redirected to *file*.

*command* ... 2>&1

runs *command* with its standard output and message output merged. (File descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

# **list *.c | lpr &**

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

# **ed file &**

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UNIX convention for a signal is that if it is set to 1 (ignored), then it is never changed even for a short time. Note that the shell command

**trap** has no effect for an ignored signal.

## 2.4 A 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**

## 2.5 SUMMARY OF BOURNE SHELL METACHARACTERS AND RESERVED WORDS

### 2.5.1 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 |

### 2.5.2 Patterns

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

### 2.5.3 Substitution

| | |
|---|---|
| **${...}** | substitute shell variable |
| **`...`** | substitute command output |

### 2.5.4 Quoting

| | |
|---|---|
| **** | quote the next character |

´...´ quote the enclosed characters except for `

“...” quote the enclosed characters except for $ ` \ “

### 2.5.5 Reserved Words

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

# Chapter 3: Using the C Shell

## 3.1 INTRODUCTION

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 (*/bsd4.2/bin/csh*) is one of several shells available to DOMAIN/IX/*bsd4.2* users.

**Note:** This chapter is an introduction to the more commonly-used features of the C Shell. The csh documentation in the *DOMAIN/IX Command Reference for BSD4.2* provides a full description of all features of this shell.

Before you read any further in this chapter, please read the material on multiple version support in Section 1, Chapter 1. It explains how DOMAIN/IX allows you to invoke shells in either the *sys5* or the *bsd4.2* environment. Although the process is nearly transparent to the user (requiring only that you set the *SYSTYPE* environment variable), it has implications for software developers and, to a lesser extent, for people (and shells) running programs.

## 3.2 FIRST STEPS

This chapter includes several examples. We recommend that you try them all. The best way to learn about the C Shell is to work with it for a while.

### 3.2.1 Special Key Definitions

Read the material in Section 1 of this manual about key binding and key definitions files. It explains how the keys on DOMAIN node keyboards are bound to the functions they execute.

In this chapter, we assume that you have invoked one of the *bsd4.2* key definitions files. These files bind various keys to functions that the C Shell provides. To invoke these definitions, press the CMD key and enter the following DM command.

Command: **cmdf /sys/dm/***file*

where file is either

- *bsd4.2_keys* if you have an 880 (high-profile) keyboard,
- *bsd4.2_keys2* if you have a low-profile keyboard.

Special key definitions (in addition to those provided in *unix_keys*) included in this file are:

```
#  part of /sys/dm/bsd4.2_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 (useful when you ^z a non-csh and want to wake it
up)
ke ^j dq -c 12002b ke
```

In addition to the lines above, the *bsd4.2_keys2* file includes the line:

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

### 3.2.2 Starting the Shell

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

Command: **cp /bin/start_csh**

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

**Note:** If *SYSTYPE* is not *bsd4.2* or *bsd4.1*, process creation will fail, since there is no C Shell in *sys5* or *sys3*.

The DM will open a window and run the C Shell in it. When giving the **start_csh** command, you may supply the coordinates at which you want the DM to locate the upper left and lower right corners of the window. You may even give the process a name, as in the line below.

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

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

**Note:** In this chapter, there are several examples of user interaction with the C Shell. In these examples, we will use bold text to denote what the user types, and standard roman text to denote prompts and messages returned to the screen. For example, the following display

```
% pwd
% //ice/kate
```

means we typed **pwd**, followed by an (implied) RETURN. The Shell then printed the current working directory, //ice/kate.

### 3.2.3 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**

consists of a command name, **mail**, followed by an argument, **kate**. The shell looks in a number of directories for a file with the name **mail**. When it finds something called **mail**, the shell assumes it is an executable file and attempts to have the system execute it.

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
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.

**Note:** In this chapter (and throughout this document), the notation ↑*x* should be read "control-*x*." It means "press the key named *x* while holding the CTRL (Control) key down."

**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 the characters

%

(a percent symbol followed by a space) to indicate that it was ready for further orders.

This is the essential pattern of all interactions with DOMAIN/IX via the C Shell. A complete command is typed at the terminal; the shell executes the command, and when this execution completes, the shell prompts for a new command. If you run, for example, the **vi** editor for an hour, the shell will patiently wait for you to finish editing, then prompt you for

further orders.

### 3.2.4 Flag Arguments

While many arguments to commands specify objects like file names or user names, some arguments invoke optional capabilities of the command. By convention, such arguments begin with the character "**-**" (hyphen). Thus, the command

**ls**

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

**ls -s**

it causes **ls** to list the size of each file in blocks of (normally) 1024 characters. The manual section for each command in the *DOMAIN/IX Command Reference* gives the available options for each command.

### 3.2.5 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*. The command

**date**

prints the current date on the transcript pad of the shell into which **date** was typed. This is because the screen (the 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 metacharacter > and the name of the file where output is to be placed. Thus, the command

**date > now**

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 on the file *now*. No output appears on the screen. It is important to know that **date** was unaware that its output was going to a file rather than to the screen. The shell performed this redirection before the command began executing.

We must also point out 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 a file that exists, that file will be overwritten unless the shell variable **noclobber** has

been **set**. See the discussion of **noclobber** in the next section.

### 3.2.6 Metacharacters in The C Shell

The C Shell makes use of a number of characters (like the recently discussed ">") which perform special functions. In general, most characters that are neither letters nor digits have special meaning to the shell. Since these special characters also have their normal uses, the shell provides a means of quotation that allows you to strip these metacharacters of special meaning.

Metacharacters normally have effect only when the shell is reading input. You need not 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:** As we have mentioned, AEGIS commands perform their own wildcard expansion, using rules different from those used by the C Shell. Unquoted wildcards used in the C Shell will be expanded according to the C Shell's rules, then passed to whatever command is being executed. If you are 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 the AEGIS commands.

### 3.2.7 Input From Files; Pipelines

We have already discussed how to redirect the standard output of a command to a file. It is also possible to redirect the standard input of a command so that it is taken from a file, rather than from the keyboard (the default standard input). This is not often necessary since most commands will read from a file whose name is given as an argument. You can give the command

% **sort < data**

to run the **sort** command with standard input, where the command normally reads its input, from the file *data*. It would be easier (and equally legal) for you to type

% **sort data**

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

**Note:** If you merely typed

% **sort**

then the **sort** program would sort lines from its standard input, the keyboard. Since you did not redirect the standard input, **sort** would sort lines as you typed them on the

terminal, until you typed 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**

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

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

**ls -s | sort -n**

tells the C Shell to run the **ls** command with the **-s** option, 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 command in combination with the previous command doing

**ls -s | sort -n -r | head -5**

This sequence takes a list of files sorted alphabetically, each with the size in blocks, and pipes this list to the standard input of **sort**. **Sort**, in turn, sorts the list numerically in reverse order (largest first). The sorted list is piped into the command **head** which, in this case, displays the first 5 lines of the list, giving you names and sizes of the 5 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 will normally take its standard input from the keyboard. The rightmost will place its standard output on the screen.

### 3.2.8 Filenames

Many commands need the names of files as arguments. In 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, better known as "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 "//?" 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** (change directory) 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 at all, the shell assumes that the pathname is the name of a file contained in the current working directory. Absolute pathnames, by contrast, have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and periods. While all printing characters except '/' (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 character "." (period, or dot), while not a C Shell metacharacter, is often used to separate the extension of a file name 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 "." and following characters which are not "." 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.

If you wished to refer to all four of these files in a command, you could use the notation

prog.*

The shell expands *prog.**, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character "*" here matches any sequence (including the empty sequence)

of characters in a filename. The names which match are alphabetically sorted and placed in the argument list of the command. Thus, the command

% **echo prog.***

will echo 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 listed them above. The **echo** command receives four words as arguments, even though only one argument was supplied to the Shell. The Shell generated the four words by filename expansion of the one input word.

There are other characters that the C Shell will expand. The character "?" matches any single character in a filename. Thus,

**echo ? ?? ???**

will echo 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 will be sorted independently — that is, the output to the screen will be a list of one-character filenames, followed by a list of two-character filenames, followed by a list of three-character filenames.

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

prog.[co]

will match both *prog.c* and *prog.o*. You can also place two characters around a "-" in this notation to denote a range. Thus, if you wanted to **troff** five chapters of a book that existed in the files *chap.1*, *chap.2* and so on, you could type the command line

% **troff chap.[1-5]**

which would pass the names

**chap.1 chap.2 chap.3 chap.4 chap.5**

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

**chap.[12345]**

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

No match.

and does not execute the command.

Files that begin with the character "." are treated specially. Neither "*" or "?" or the "[^]" mechanism will match it. This prevents accidental matching of the filenames "." and ".." in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible in a directory listing. We will 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 the character ~ (the tilde) followed by another user's login name. For instance, the word **~kate** would map to the absolute pathname of user kate's home directory, for example,

% **cd ~kate**
% **pwd**
% //ice/kate

A special case of this notation consists of a "~" 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 ~**

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

% **cp thatfile ~**

will be expanded to

cp thatfile //*your_home_directory*/thatfile

You may change this character by setting the shell variable *homedirchar* to some other character. For example, to change the home directory character to "#", say

% **set homedirchar** = #

To revert to the default, unset homedirchar.

**Note:** If you have used the DM environment variable *NAMECHARS* (see Section 1, Chapter 1) to assign DOMAIN naming server meta-meanings to the tilde, the naming server will expand 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 the left and right brace characters { and } for abbreviating a set of words that have common parts but cannot be abbreviated by the above mechanisms because they are not files (or are files that, while they will be created by the program that is being invoked, do not exist yet). This mechanism will be described in a

later section.

### 3.2.9 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 ***

will not echo the character '*'. It will either echo a sorted list of all filenames in the current working directory, or print the message "No match" if there are no files in the working directory.

The recommended mechanism for placing a character that is neither a number, a digit, "/", "." nor "-" in an argument word to a command is to enclose it in single quotes "´", i.e.,

% **echo ´*´**

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

% **echo \´\!**

prints

´!

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

**echo \´´*´**

which prints

´*

since the first "\" escaped the first "´" and the "*" was enclosed between "´" characters.

**Note:** If you have used the DM environment variable *NAMECHARS* (see Section 1, Chapter 1) to assign DOMAIN naming server meta-meanings to any of the characters tilde, grave accent, and backslash and you use one of those characters — escaped — in a pathname component, the character will be interpreted not as a literal, but according to the naming server's rules.

### 3.2.10 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. For instance if

you type the command

% **cat /etc/passwd**

the system will print 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** command by typing ↑I.

**Note:** The DM command files *unix_keys* and *bsd_keys* define ↑I as the UNIX interrupt key. You must execute one of these command files in order for this definition to be in effect.

Since **cat** does not take any precautions to avoid or otherwise handle this signal, the INTERRUPT will cause **cat** to terminate. The shell notices that **cat** has terminated and prompts you again. If you hit INTERRUPT again, the shell will just repeat its prompt since it is designed to effectively ignore INTERRUPT signals.

Many programs will terminate when they get an end-of-file from their standard input. Thus, the **mail** program in our earlier example terminated when it received a ↑D (which *bsd4.2_keys* defines as generating an end-of-file) from the standard input. The C Shell normally terminates when it receives an end-of-file. When this happens, the message

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

will be left on the transcript pad and the window will be 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 will be discussed in the next section.

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

% **mail kate < prepared.text**

the mail command will terminate 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 would be to type

% **cat prepared.text | mail kate**

since the **cat** command would then have written the text through the pipe to the standard input of the mail command. When the **cat** command completed, it would have terminated, closing down the pipeline and the **mail** command would have received an end-of-file from **cat** and terminated. These commands could also have been stopped by typing ↑I.

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

Quit

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

Commands running in the background will ignore INTERRUPT and QUIT signals. To stop them you must use the **kill** command. Kill and background commands are covered in a later section.

## 3.3 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.

### 3.3.1 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 System* and the *DOMAIN System Command Reference* for more detailed information). It will normally open an AEGIS shell, then look 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 like

(0,200)dr; (540,600)cp /bsd4.2/bin/start_csh -n c_shell

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

**Note:** In the example line above, we specified

/bsd4.2/bin

as the */bin* to use. See the information on multiple version support (Section 1, Chapter 1) for more on this.

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 labelled SHELL — so that when you press SHIFT SHELL a C Shell will be opened.

kd l5s cp /bin/start_csh ke

In this case, since no */bin* is specified, the **start_csh** command is obtained from */$(SYSTYPE)/bin*.

### 3.3.2 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 will read (after it reads **/bin/start_csh** command.

Neither of these files is required. If either doesn't exist, the shell will use its own defaults.

**Note:** If you use the DM editor to create your *.cshrc*, or *.login*, you must also use the UNIX command **chown**[1] to make yourself the owner of these files. Otherwise, they will not be read.

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

```
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
- when a command is typed, the shell will look for it in the following places, in the order listed
  1. the current directory (.)
  2. the directory *home_ directory/ com*
  3. the */usr/ucb* directory
  4. the */bin* directory
  5. the */usr/bin* directory
  6. the */com* directory
- 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 will not terminate (e.g., 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**.)
- the variable *homedirchar* is **set** to make the home directory character a % rather than the default ~ .

The next two commands are **alias** commands that, in effect, rename command sequences. In this case, 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**, providing a way to close up the shell window with a minimum of typing.

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

**>!**

For example, if you really wanted to overwrite the contents of a file named *now* with the current date, you could do so even if **noclobber** had been set. The command line

date >! now

would do it. The ">!" is a special metasyntax indicating that clobbering the file is allowed. Note that the space between the "!" and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, and have a totally different effect.

### 3.3.3 Shell Variables

The shell maintains a number of variables. In the *.cshrc* file shown above, 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 was given above and is

**set** *name=value*

Shell variables provide you with a way to 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 the variable **path**. This variable contains a sequence of directory names where the shell searches for commands. If you execute the **set** command with no arguments, the shell will display 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 will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

% **rehash**

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory "." on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable home which shows your home directory, and the variable **cwd** which contains your current working directory. Note that variable **ignoreeof** is one of several variables that have no value other than off or on — more correctly **unset** or **set**. Thus, to set this variable you simply

**set ignoreeof**

and to unset it,

**unset ignoreeof**

The variable **noclobber** is another one of these "boolean" variables. It can only assume two states.

### 3.3.4 History

The shell can maintain a history list into which it places the words of previous commands. The C Shell's history mechanism makes it possible to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following example is a session involving typical usage of the C Shell's history mechanism.

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

```
num bug.c | ssp | prf
%
```

In this example there is a very simple C program which has a bug (or two) in it. To begin with, the file *bug.c* is **cat**ted onto the screen. An attempt is made to run the C compiler on it, referring to the file again as "!$", which is an invocation of the history mechanism that means "use the last argument to the previous command." The **!** is the metacharacter that invokes the history mechanism and the **$** stands for the last (most recent) argument read by the shell. The shell echoes the command, as it would have been typed without use of the history mechanism, and then executes it. Since the compilation yielded error diagnostics, the editor had to be invoked in order to fix the bug, then the file had to be compiled again, 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 it's necessary to be more specific, (for example, if other commands starting with "c" had been used recently), the history mechanism could have been invoked by typing **!cc**. If further caution was needed, the form **!cc:p** just prints the last command that started with "cc," without appending the RETURN that would execute it.

After this recompilation, a run of the resulting *a.out* file revealed that there was still a bug, so the editor was invoked again, then the C compiler. This time, the **-o bug** switch was added 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 ran the **size** 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 ran the program **bug** to see that its output was indeed correct.

To make a numbered listing of the program we ran the **num** command on the file *bug.c*. In order to remove blank lines in the output of **num** we ran the output through the filter **ssp**, but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between A characters. This is similar to the substitute command in the editor. Then we repeated the same command with **!!**, but sent its output to the line printer.

**Note:** On DOMAIN nodes, the AGAIN key 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 prove more effective than the history mechanism in certain situations. See the *DOMAIN System Command Reference*, as well as Section 1, Chapter 4 of the *DOMAIN/IX Text Processing Guide* manual for more

information on the DM editor's cut-and-paste facility.

There are other ways to repeat a command from the history list. The **history** command prints out a number of previous commands accompanied by the numbers with which they can be referenced. There is also a way to refer to a previous command by searching for a string that appeared in it. A complete description of all these mechanisms is given in the C Shell manual pages in the *DOMAIN/IX Command Reference*.

### 3.3.5 Aliases

The shell has an alias mechanism that is very useful for transforming input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to do transformations on commands and their arguments. The alias mechanism is similar to a macro facility. Some of the features obtained by aliasing can also be obtained using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment or involve commands, such as **cd**, which must be done in the current shell. As an example, suppose you wish the command **ls** to always show sizes of files, that is to always do **-s**. The following alias would do the trick.

% **alias ls ls -s**

You could even create a "new" command called 'dir' that does an 'ls -s'.

% **alias dir ls -s**

Thus, the alias mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to 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 the *.cshrc* file above:

% **alias cd 'cd \!* ;ls'**

causes the shell to automatically do an **ls** after every **cd**. It was necessary to enclose the entire alias definition in "´" characters to prevent most substitutions from occurring and to prevent the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' 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 ';' is used here to indicate that one command is to be done first, followed by the next. Similarly the definition

% **alias whois ´grep \!^ /etc/passwd´**

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

**Note:** The C Shell reads the *.cshrc* file each time it is invoked. If you place a large number of commands there, shells will tend to start slowly. We recommend that you limit the number of aliases you place in this file. Ten aliases will cause no perceived delay. Fifty aliases will 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.

### 3.3.6 More Redirection; >> and >&

There are a few more useful C Shell notations that need to be introduced. 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:** The notation

*command* >**&!** *file*

can be used when **noclobber** is set and *file* already exists.

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 will result if *file* does not exist; otherwise the shell will create *file* if it doesn't exist. A form

*command* >>**!** *file*

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

### 3.3.7 Jobs; Background, Foreground, or Suspended

Whenever one or more commands are connected via pipes or as a sequence of commands separated by semicolons, a single job is created by

the shell 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 metacharacter **&** at the end of a command line, then the job generated by that command line is started as a background job. This means that 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 &**

would run the **du** program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file *usage* and return immediately with a prompt for the next command without waiting for **du** to finish. The **du** program would continue executing in the background until it finished, and the shell would continue accepting input from you. When a background job terminates, the shell types a message before the next prompt, telling you that the job has completed. In the following example, the **du** job finishes sometime during the execution of the **mail** command. Its completion is reported just before the prompt after the **mail** job is finished.

```
% du > usage &
[1] 503
% mail kate
How can I tell when a background job is finished?

bob
*** EOF ***
EOT
[1] - Done du > usage
%
```

If the job did not terminate normally the "Done" message might say something else like "Killed". If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the **notify** variable. In the previous example, this would mean that the "Done" message might have appeared in the middle of the message to Kate. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

**Note:** On DOMAIN systems, you can invoke a C Shell with or without the ability to suspend/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 will

not be able to use the **fg**, **bg**, and **stop** commands unless *inprocess* is **unset**. When the *inprocess* is **set** (the default condition), the following limitations — all of which concern AEGIS commands, apply.

- The /**com**/**tb** command will always return the message "no traceback available."
- Libraries loaded with the **inlib** command will not be available to programs running in an environment where *inprocess* is **unset**.
- The /**com**/**las** command will only list the address space occupied by itself.
- The /**com**/**lopstr** command shows only those streams that the C-Shell has open.
- The /**com**/**wd** command will not work.

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 command names, arguments, and the 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 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 &
[2] 65 66
%
```

runs the **ls** program with the **-s** options, pipes this output into the **sort** program with the **-n** option which puts its output into the file "usage". Since there was an **&** at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number (**[2]** in this case) in brackets followed by the job's process numbers, then prompts for a new command.

To suspend a foreground job, you need to send a STOP signal to the shell process. If you invoke the *bsd4.2_keys* key definitions, suspend will be 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 described below. When jobs are suspended, they merely stop

any further progress until started again, either in the foreground or the background. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs, this looks like

```
% du > usage
↑Z Stopped
%
```

The shell displays the "Stopped" message when it notices that a job (in this case, the **du** program) has stopped. When a background job is stopped with the **stop** command, the shell prints a slightly different message.

```
% sort usage &
[1]23
% stop %1
[1] + Stopped (signal) sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended, then continued as background jobs using the **bg** command, allowing you to continue other work and stop waiting for the foreground job to finish. In this sequence,

```
% du > usage
↑Z
Stopped
% bg
[1] du > usage &
%
```

we start **du** in the foreground, stop it before it finishes, then continue it in the background.

All job control commands can take an argument that identifies a particular job. All job name arguments must begin with the character **%**, since some of the job control commands also accept process numbers.

**Note:** To obtain the numbers of all running or suspended processes, use the **ps** (process statistics) command.

The default job (when no argument is given) is called the "current job" and is identified by a "+" 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 **-** in the output of **jobs**. When the current job terminates, the previous job becomes the

current job. When given, the argument to **jobs** is one of the following.

| | |
|---|---|
| **%-** | the previous job |
| **%***n* | where *n* is the job number |
| **%***pref* | where *pref* is some unique prefix of the command name and arguments of one of the jobs |
| **%?***string* | where *string* is a string found in only one of the command lines that set up a job. |

The **jobs** command lists the table of jobs, giving the job number, commands, and status ("Stopped" or "Running") of each background or suspended job. With the **-l** option, the process numbers are also given.

```
% du > usage &
[1] 33
% ls -s | sort -n > myfile &
[2] 34
% mail ers
↑Z
Stopped
% jobs
[1] - Running du > usage
[2] Running ls -s | sort -n > myfile
[3] + Stopped mail ers
% fg %ls
ls -s | sort -n > myfile
% more myfile
```

The **fg** moves a job into the foreground. If the job is suspended, it will be restarted. If the job is already running in the background, it will continue to run but will become the foreground job and, as a consequence, will be able to accept signals or input from the terminal. In the above example, we used **fg** to change the **ls** job from the background to the foreground since we wanted 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 into 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 (printed by **ps**) as arguments. Thus, in the example above, the running **du** command could have been terminated by the command

```
% kill %1
[1] Terminated du > usage
```

The **notify** command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes, instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal, it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the "s" command in the text editor might take a long time.

```
% ed bigfile
120000
1,$s/thisword/thatword/
↑Z
Stopped
% bg
[1] ed bigfile &
%
some foreground commands
[1] Stopped (tty input) ed bigfile
% fg
ed bigfile
w
120000
q
%
```

So after the "s" command was issued, the **ed** job was stopped with ↑Z, and then put in the background using **bg**. Some time later when the "s" command was finished, **ed** tried to read another command and was stopped because jobs in the background cannot read from the terminal. The **fg** command returned the **ed** job to the foreground where it could once again accept commands from the terminal.

**Note:** The **jobs** command only prints jobs started in the currently executing shell. It knows nothing about background jobs started in other shells. Use **ps** to find out about background jobs not started in the current shell.

### 3.3.8 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 wish to work on, then place all files related to that project in that directory. The "make directory" command, **mkdir**, creates a new directory. The **pwd** ("print working directory") command reports the absolute pathname of the working directory of the shell, that is, the directory in which you are located. Thus, in the example below;

```
% pwd
//ice/kate
% mkdir newdocs
% cd newdocs
% pwd
//ice/kate/newdocs
%
```

we have created and moved to the directory *newdocs*.

No matter where you have moved to in a directory hierarchy, you can return to your "home" directory by doing just

**cd**

with no arguments. The name **..** ("dot dot") always means the directory above the current one. Thus,

**cd ..**

changes the shell's working directory to the parent of (the directory immediately above) the current directory. The name **..** can be used in any pathname, thus,

**cd ../programs**

means change to the directory *programs* contained in the directory above the current one. If you have several directories for different projects under your home directory, this shorthand notation makes it easier to switch between them.

The shell always remembers the pathname of its current working directory in the variable **cwd**. The shell can also be requested to remember the previous directory when you change to a new working directory. If the "push directory" command, **pushd**, is used in place of the **cd** command, the shell saves the name of the current working directory on a directory stack before changing to the new one. You can see this list at any time by typing the "directories" command **dirs**.

```
% pushd newpaper/references
%/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~ /newpaper/references ~
% dirs
/usr/lib/tmac ~ /newpaper/references &
% popd
%/newpaper/references ~
% popd
%
%
```

The list is printed in a horizontal line, reading left to right, with a tilde as shorthand for your home directory. The directory stack is printed whenever there is more than one entry 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) by **pushd** command. There are other options to **pushd** and **popd** to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the **csh** manual pages in the *DOMAIN/IX Command Reference for BSD4.2* 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, then change the shell's working directory, then bring a background job into the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs -l
//ice/kate
% cd my project
% dirs
%/myproject
% ed prog.c
1143
↑Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c                    working dir is: ~ /myproject
```

This way the shell warns you when there is an implied change of working directory, even though no **cd** command was issued. In the above example the **ed** job was still in */ice/kate/myproject* even though the shell had changed to */ice/kate/*. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell implies a change of working directory.

```
% fg
ed prog.c                working dir is: ~ /myproject
q
working dir is now: ~
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell assumes that a job stays in the same directory where it started. The **-l** option of **jobs** will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

### 3.3.9 Useful Built-in Commands

This section describes a few of the useful built-in shell commands and explains how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments, it prints a list of the current aliases. If given a single argument, such as

```
% alias ls
```

**alias** will show the current alias for that argument (e.g., **ls**).

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will produce.

The **history** command shows the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called **prompt** which tells the C Shell to use a specific character or string as the prompt. Thus, if you

```
% set prompt=´\! %´
```

the shell will prepend the number of the current command in the history list to the % sign. Note that the **!** character had to be escaped here even within "´" characters.

The **logout** command can be used to terminate a login shell which has **ignoreeof** set.

The **rehash** command causes the shell to recompute a table of command locations. You'll need to do a **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 5 copies of the file *one* in the *file* five you could do

% **repeat 5 cat one >> five**

The **setenv** command can be used to set variables in the C Shell environment. Thus,

**setenv TERM vt100**

will set the value of the environment variable TERM to "vt100". There is a program called **printenv** that will print out the environment. It might then show:

```
% printenv
USER=kate
LOGNAME=kate
PROJECT=none
ORGANIZATION=doc
NODEID=1054
PATH=:~ com:/usr/ucb:/bin:/com:/usr/bin
TERM=apollo_19L
NODETYPE=DN300
TZ=EST5EDT
HOME=//ice/kate
```

The **source** command can be used to force the current shell to read commands from a file. Thus,

**source .cshrc**

can be used after making a change to the **.cshrc** file if you wish the change to take effect immediately. The **unalias** command cancels aliases. **Unset** removes shell variables, and **unsetenv** removes environment variables.

## 3.4 SHELL CONTROL STRUCTURES AND SHELL SCRIPTS

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called "shell scripts." In this section, we detail those features of the shell useful to the writers of such scripts.

### 3.4.1 Invocation and the argv Variable

To run a C Shell script, you may type

% **csh** *scriptname args*

where *scriptname* is the name of the file containing a group of **csh** commands and *args* denotes 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 doing

**chmod 755** *scriptname*

and place the line

#!/bin/csh

as the first line of the file *scriptname*, a C Shell will automatically be invoked to execute *scriptname* when you type

*scriptname*

In general, you should always start a shell script with a line of 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 #, the shell in which you invoked the script will try to execute it, with unpredictable results.

### 3.4.2 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 $ character, 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 would cause 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*. The sequence below should make this more clear.

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus,

```
$argv[1]
```

gives the first component of argv or in the example above "a". Similarly

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give "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. Since this process number is unique in the system, it can be used in generation of unique temporary file names. 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=($<)
```

would write out the prompt "yes or no?" without a newline and then read the answer into the variable "a". In this case "$#a" would be "0" if either a blank line or end-of-file (↑D) was typed.

For compatibility with the way older shells handled parameters, the form "$argv[*n*]" will yield an error if *n* is not in the range "1-$#argv" while "$n" will never yield an out-of-range subscript error.

It is never an error to give a subrange of the form "n-"; if there are less than **n** components of the given variable then 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.

### 3.5 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 (like *, ? 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 **?** is replace 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.

It is possible to 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 variable **$status**.

**Note:** Since **$status** is set by every command, you will have to save a particular command's **$status** if you can't examine it

immediately after the command is executed.

### 3.5.1 A Sample Shell Script

The following shell script, called *copyc*, makes use of 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 makes use of 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 **(** and **)** with the named **variable**, in this case **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.

We set the variable **noglob** here to prevent filename expansion of the members of **argv**. This is generally a good idea if the arguments to a shell script are filenames that have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible, though tedious, to quote each use of a $ variable expansion.

The other control construct used here is a statement of the form

```
if ( expression ) then
command
...
endif
```

**Note:** The placement of the keywords here is not flexible. The following two formats are not acceptable 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 **|**, **&** or **;** and must not be another control command. In the second form, the final "\" 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 **:** modifier. We can use the modifier **:r** here to extract a root of a filename or **:e** to extract the **extension**. Thus, if the variable **i** has the value **/mnt/foo.bar** then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the **:r** modifier strips off the trailing **.bar** and the **:e** modifier leaves only the bar. Other modifiers will take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the *DOMAIN/IX Command Reference for BSD4.2*. It is also possible to use the command substitution mechanism, described in the next major section, to perform modifications on strings.

**Note:** The C Shell allows only one **:** modifier on a $ substitution. Thus,

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would otherwise expect.

Finally, we note that the character # lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a # are discarded by the shell. This character can be quoted using "" or "\" to place it in an argument word.

### 3.5.2 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

For details, see the material on the C Shell in the *DOMAIN/IX Command Reference for BSD4.2*. C programmers should note that 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 looking like they do in C, i.e.,

loop:
commands
goto loop

### 3.5.3 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 which 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 which 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 notation "<< ´EOF´" means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly "´EOF´". The fact that the "EOF" is enclosed in "´" characters (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 will not be performed. In this case since we used the form "1,$" in our editor script, we needed to ensure that this $ did not trigger a substitution. We could also have ensured this by preceding the "$" here with a "\", i.e.,

```
1,\$s/^[ ]*//
```

but quoting the "EOF" terminator is a more reliable way of achieving the same result.

### 3.5.4 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 will do a "goto label." You can then remove the temporary files and do an **exit** command (built in to the shell) to exit from the shell script. If you wish to exit with a non-zero status, do

```
exit(1)
```

to exit with status "1".

### 3.5.5 Additional Options

There are other features of the shell useful to writers of shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the shell. The **-n** option causes the shell to read commands but not execute them, something which may be of use when debugging.

There is also another quotation mechanism using " (the double quote) which allows only some of the expansion mechanisms we have so far

discussed to occur on the quoted string and serves to make this string into a single word as ‘´’ does.

## 3.6 OTHER SHELL FEATURES

The C Shell features discussed in this section are less commonly used and will be touched lightly here. Further details on these features can be obtained from the C Shell section of the *DOMAIN/IX Command Reference for BSD4.2*.

### 3.6.1 Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if you needed 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
3
% grep -c '^\.TS|.EQ' dir2
5
% grep -c '^\.TS|.EQ' dir3
6
```

or you could use foreach to do this more easily.

```
% foreach i (´dir1´ ´dir2´ ´dir3´)
? grep -c '^\.TS|.EQ'  $i
? end
3
5
6
%
```

Note here that the shell prompts for input with "? " when reading the body of the loop.

Variables that contain lists of filenames or other words are also useful in loops. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The **set** command here gave the variable **a** a list of all the filenames in the current directory as value. You can then iterate over these names to

perform any chosen function.

The output of a command within

`

characters is converted by the shell to a list of words. You can also place the "`" quoted string within

"

characters 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" exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

### 3.6.2 Braces { ... } in Argument Expansion

Another form of filename expansion involves the brace characters

{}

These characters specify that the delimited strings, separated by "," 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 (i.e., nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

% **mkdir ~/{hdrs,retrofit,csh}**

to make subdirectories *hdrs*, *retrofit*, and *csh* in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.,

chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}

### 3.6.3 Command Substitution

A command enclosed in ` characters is replaced, just before filenames are expanded, by the output from that command. Thus, you may

% **set pwd=`pwd`**

to save the current directory in the variable *pwd* or to do

% **ex `grep -l TRACE *.c`**

to run the editor **ex** supplying as arguments those files whose names end in ".c" which have the string "TRACE" in them.

**Note:** Command expansion also occurs in input redirected with "<<" and within "" quotations. Refer to the C Shell pages in the *DOMAIN/IX Command Reference for BSD4.2* for full details.

### 3.6.4 Other Details Not Covered Here

In particular circumstances, it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed in the *DOMAIN/IX Command Reference for BSD4.2*.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts.

## 3.7 A SUMMARY OF C-SHELL METACHARACTERS

This section lists the metacharacters recognized by the C Shell. A number of these characters also have special meaning in expressions. See the information on **csh** in the *DOMAIN/IX Command Reference for BSD4.2* for a complete list.

**Note:** If you have used the DM environment variable *NAMECHARS* (see Section 1, Chapter 1) to assign DOMAIN naming server meta-meanings to any of the characters tilde, grave accent, and backslash and you use one of those characters — escaped — in a pathname component, the character will be interpreted not as a literal, but according to the naming server's rules.

### 3.7.1 Syntactic Metacharacters

**;** separates commands to be executed sequentially

**|** separates commands in a pipeline

**( )** brackets expressions and variable values

**&** follows commands to be executed in background

### 3.7.2 Filename Metacharacters

**/** 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

**{ }** used to specify groups of arguments with common parts

### 3.7.3 Quotation Metacharacters

\ prevents meta-meaning of following single character

´ prevents meta-meaning of a group of characters

” like ´, but allows variable and command expansion

### 3.7.4 Input/Output Metacharacters

< indicates redirected input

> indicates redirected output

### 3.7.5 Expansion/substitution Metacharacters

**$** indicates variable substitution

**!** indicates history substitution

**:** precedes substitution modifiers

used in special forms of history substitution

` indicates command substitution

### 3.7.6 Other Metacharacters

# begins shell comment

**-** prefixes option (flag) arguments to commands

% prefixes job name specifications

# Index

**T**



**W**

## SECTION 3

# COMMUNICATIONS

CONTENTS

# Chapter 1: UNIX-TO-UNIX COPY (UUCP)

## 1.1 INTRODUCTION

**Uucp** is the name of the central program in a group of programs that, together, permit communication between UNIX systems using either dial-up or hardwired connections. The first version of the system was designed and implemented at Bell Labs. Today, it is commonly used for file transfers and remote command execution. This chapter explains how **uucp** works and includes information about installing **uucp** on your DOMAIN system.

**Uucp** is a batch-type operation. Files are created in a spool directory for processing by the **uucp** daemons. There are three types of files used for the execution of work.

| | |
|---|---|
| Data files | contain data for transfer to remote systems. |
| Work files | contain directions for file transfers between systems. |
| Execution files | contain directions for executing UNIX commands that involve the resources of one or more systems. |

The **uucp** system consists of four primary and programs and several secondary programs. The primary programs are:

| | |
|---|---|
| **uucp** | This program creates work files and gathers data files in the **uucp** spool directory. |
| **uux** | This program creates work files and execute files. It also gathers data files for the remote execution of UNIX commands. |
| **uucico** | This program executes the work files for data transmission. |
| **uuxqt** | This program executes UNIX commands from command files. |

**Note:** On DOMAIN systems, **uucico** runs on all nodes that have DOMAIN/IX software. Another program, called **uucico.real**, runs only on the node(s) from which the actual connection (to another system) is made.

Secondary uucp programs common to both *sys5* and *bsd4.2* are:

| | |
|---|---|
| **uulog** | This program updates the log file with new entries and reports on the status of uucp requests. |

**uuclean** This program removes old files from the spool directory.

The *bsd4.2* version of **uucp** includes these additional programs.

**uusend** sends a file to a given location on a remote system.

**uusnap** displays a tabular synopsis of the current uucp activity at your site.

The *sys5* version of **uucp** includes these additional programs.

**uuto** sends selected files to another **uucp** site.

**uupick** allows you to accept or reject files sent to you with **uuto**

**uusub** allows you to define and monitor activity on a **uucp** subnetwork.

**uuname** lists the names of other **uucp** sites known to your site.

**uustat** lets you examine or change the status of submitted (or running) **uucp** jobs.

In the remainder of this chapter, we describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

## 1.2 DIFFERENCES BETWEEN sys5 AND bsd4.2 UUCP

While the **uucp** programs provided with the *sys5* and *bsd4.2* versions of DOMAIN/IX perform essentially the same functions, there are differences between them which need to be pointed out. In the rest this chapter, we will flag those features that are unique to the *sys5* version of **uucp** with a †. Features that are unique to the *bsd4.2* version of **uucp** will be flagged with a ‡. Unflagged items are common to both.

There is an additional difference in implementation of which you should be aware. The *bsd4.2* version of **uucp** creates a number of subdirectories in the directory */usr/spool/uucp*; one for each type of uucp file. "D." files are stored in */usr/spool/uucp/D.*, "C." files are stored in */usr/spool/uucp/C.*, and so on. The *sys5* version of **uucp** does not make these subdirectories.

## 1.3 UUCP—UNIX-TO-UNIX FILE COPY

The **uucp** command looks to the user much like the UNIX command **cp**. The syntax is

**uucp** [ *option(s)* ] *source ...destination*

where *source* and *destination* may contain the prefix *system-name*! that indicates the system on which the file(s) can be found (or the one to which the files will be copied).

**Note:** C shell users should be sure to escape (through quotation or use of \) any shell metacharacters in **uucp** command lines. See Section 2, Chapter 3 for details.

Uucp interprets the following options.

**-d** Make directories when necessary for copying the file.

**-c** Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

**-g***letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

**-m** Send mail on completion of the work.

**-n***user* † Notify *user* on the remote system that a job was sent.

**-C** † Copy the source file to the spool directory.

**-f** † Do not make intermediate directories for the file copy.

**-e***sys* † Execute **uucp** on system *sys*.

**-j** † Control writing of the **uucp** job number to standard output.

The following options are used primarily for debugging:

**-r** Queue the job but do not start the **uucico** program.

**-s***dir* Use directory *dir* for the spool directory.

**-x***num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain shell metacharacters. If a source argument has a *system-name*! prefix for a remote system, the filename expansion will be done on the remote system.

The command

**uucp *.c usg!/usr/dan**

will set up the transfer of all files whose names end with a ".c" to the "/usr/dan" directory on the "usg" machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory on the specified system.

**Note:** (*bsd4.2* only) The "home" character, normally tilde (~), can be redefined in the C Shell. In this chapter, we assume that it has not be so redefined.

For names with partial pathnames, the current directory is prepended to

the file name. File names with ../ are not permitted.

The command

**uucp usg!~dan/*.h ~dan**

will set up a transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types.

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains shell metacharacters.

After the work has been set up in the spool directory, **uucp** calls upon **uucico** to contact the other machine and execute the work (unless the -r option was specified).

### 1.3.1 Copying on the Local System

If the source and destination are both on the local system, **uucp** simply calls **cp** and copies the file from source to destination. The **-d** and the **-m** options are not honored in this case.

### 1.3.2 Receiving Files from Other Systems

If the source is on a remote system, a "work file" is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All work files and execute files use a blank as the field separator.)

- R
- The full path-name of the source or a ~*user*/*path-name*. The ~*user* part will be expanded on the remote system.
- The full path-name of the destination file. If the ~*user* notation is used, it will be immediately expanded to be the login directory for the user.
- The user's login name.
- A "-" followed by an option list. (Only the -m and -d options will appear in this list.)

### 1.3.3 Sending Files to a Remote System

If the destination file is on a remote system, a work file is created for each source file, then the source file is copied into a "data file" in the

spool directory. (A **-c** option on the **uucp** command will prevent the data file from being made. In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

1. S
2. The full-path name of the source file.
3. The full-path name of the destination or ~user/file-name.
4. The user's login name.
5. A "-" followed by an option list.
6. The name of the data file in the spool directory.
7. The file mode bits of the source file in octal print format (e.g. 0666).

### 1.3.4 Transfers from one Remote System to Another

If both source and destination files are on remote systems, **uucp** generates files that are subsequently executed by the **uucico** program running on a remote machine.

## 1.4 UUX - UNIX TO UNIX EXECUTION

The **uux** command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of **uux** is

**uux** [ *option(s)* ] *command string*

where *command string* is made up of one or more arguments. All shell metacharacters must be protected either by quoting the entire *command-string* or quoting the character as a separate argument. Within the *command-string*, the command and file names may contain a *system-name*! prefix. Arguments that do not contain a "!" will not be treated as files (i.e., they will not be copied to the execution machine.) The **-** is used to indicate that the standard input for *command-string* should be inherited from the standard input of the **uux** command. The options, mostly useful for debugging, are:

**-r** Don't start **uucico** or **uuxqt** after queuing the job;

**-x***num* *Num* is the level of debugging output desired.

The command

**pr abc | uux - usg!lpr**

will set up the output of "pr abc" as standard input to an **lpr** command to be executed on system "usg".

**Uux** generates an "execute file" that contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed.

This file is either put in the spool directory for local execution or sent to the remote system by **uucp**.

For required files that are not on the execution machine, **uux** will generate receive command files. These command-files will be put on the execution machine and executed by the **uucico** program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.) The execute file will be processed by the **uuxqt** program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

### 1.4.1 User Line

This line has the form

U *user system*

where the *user* and *system* are the requester's login name and system.

### 1.4.2 Required File Line This line has the form

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the execute file. The **uuxqt** program will check for the existence of all required files before the command is executed.

### 1.4.3 Standard Input Line

This line has the form

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "-" option is used. If a standard input is not specified, "/dev/null" is used.

### 1.4.4 Standard Output Line

This line has the form

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note that the use of ">>" is not implemented.)

### 1.4.5 Command Line

This line has the form

C *command* [ *argument(s)* ]

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All required files will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command will be executed using the Shell specified in the **uucp.h** header file. In addition, the "PATH" field from the *L.cmds* file is prepended to the command line as specified in **uuxqt**.

After execution, the standard output is copied or set up to be sent to the proper place.

## 1.5 UUCICO—COPY IN, COPY OUT

The **uucico** program performs the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a. by a system daemon (**cron**[1]),

b. by one of the **uucp**, **uux**, **uuxqt** or **uucico** programs;

c. directly — usually only for testing — by the user;

d. by a remote system.

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If the **-s***sys* option is not specified, the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

Uucico recognizes the following options.

**-r1** Start the program in *MASTER* mode. This is used when uucico is started by a program or **cron** shell.

**-s***sys* Do work only for system *sys*. If **-s** is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

**-d***dir* Use directory *dir* for the spool directory (primarily for debugging.)

**-x***num* *Num* is the level of debugging output desired.

### 1.5.1 Uucico and uucico.real

In a distributed system, it's important to distinguish between those nodes that are able to make a connection to a remote host and those that are not. Typically, one node in a DOMAIN system will be designated the "uucp server." Requests for **uucp** services entered at other nodes in the network will be handled by this server node, since attempts to handle them by other nodes would invariably fail due to lack of the appropriate connect hardware.

As has been noted, DOMAIN systems supply two versions of **uucico**: **uucico**, which, when called, simply exits, leaving your work in the queue, and **uucico.real**, which, as its name implies, is the "real" **uucico** program. Normally, **uucico.real** is invoked locally by **cron**, or when a remote host initiates a **uucp** connection with the DOMAIN system in slave mode. Work queued by you will be processed when **uucico.real** is invoked by either method. The only way for a user to directly invoke **uucico.real** is to run it on the node on which it is installed.

The directory */usr/spool/uucppublic/user_data* should include a file named *startup_sh* that contains the following lines.

```
/usr/lib/uucp/uucico.real
/com/tctl -line 1 -speed 1200 -bpc 8 -error noframing -noinsync -nosync
logout
```

This performs a function equivalent to specifying *uucico.real* in the "shell" field in the */etc/passwd* file for the "uucp" logins.

In the following subsections, which detail the operation of *uucico*, we are actually referring to **uucico.real**.

### 1.5.2 Scanning For Work

The names of the work related files in the spool directory have the format

*type.system-name grade number*

where *type* is one of the following uppercase letters.

**C** copy command file,

**D** data file,

**X** execute file

*System-name* is the remote system. *Grade* is a character. *Number* is a four digit, padded sequence number.

**Note:** In the *sys5* version of **uucp**, these files are kept in */usr/spool/uucp*. In the *bsd4.2* version, these files are kept in subdirectories */usr/spool/uucp/C.*, */usr/spool/uucp/D.*, and so on.

The file

C.res45n0031

would be a work file for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for work files (prefixed with the sequence *C.* ). A list is made of all systems to be called. **Uucico.real** will then call each system and process all work files.

### 1.5.3 Calling the Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day);

[3] device or device type to be used for call;

[4] line speed;

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU*;

[6] login information (multiple fields);

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g., mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of **uucico** will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two **uucico** programs begins with a handshake started by the called — or SLAVE — system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from

the MASTER is verified by the SLAVE and if acceptable, protocol selection begins. The SLAVE can also reply with a "call-back required" message in which case, the current conversation is terminated.

### 1.5.4 Line Protocol Selection

The remote system sends a message

**P***proto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

**U***code*

where *code* is either a one character protocol letter or *N* which means there is no common protocol.

### 1.5.5 Work Processing

The initial roles (MASTER or SLAVE) for the work processing determine the mode in which each program starts. (The MASTER has been specified by the **-r1** option of **uucico**.) The MASTER program does a work search similar to the one used when scanning for work.

There are five messages used during the work processing, each specified by the first character of the message. They are;

**S** send a file

**R** receive a file

**C** copy complete

**X** execute a **uucp** command

**H** hangup

The *MASTER* will send **R**, **S** or **X** messages until all work from the spool directory is complete, at which point an **H** message will be sent. The *SLAVE* will reply with **SY**, **SN**, **RY**, **RN**, **HY**, **HN**, **XY**, **XN**, corresponding to *Yes* or *No* for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message **CY** will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a **CN** message is sent. (In the case of **CN**, the transferred file will be in the spool directory with a name beginning with "TM".) The requests and results are logged on both systems.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote system exists in the SLAVE'S spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an **HY** response is sent.

### 1.5.6 Conversation Termination

When a **HY** message is received by the MASTER, it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final "00" message to the other. The original SLAVE program will clean up and terminate. The MASTER will proceed to call other systems and process work as long as possible or terminate if a **-s** option was specified.

## 1.6 UUXQT - UUCP COMMAND EXECUTION

The **uuxqt** program is used to execute *execute* files generated by **uux**. The **uuxqt** program may be started by either the **uucico** or **uux** programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute files* is described in the "Uux" section above.

The execution is accomplished by invoking a *sh -c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

## 1.7 UULOG - UUCP LOG INQUIRY

For each program invocation, the **uucp** programs make entries in a master log file. The **uulog** program outputs specified log entries. The output request is specified by the use of the following options:

**-s***sys* Print entries where *sys* is the remote system name;

**-u***user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

## 1.8 UUCLEAN - UUCP SPOOL DIRECTORY CLEANUP

This program is typically run once a day by **cron**. Its function is to examine the spool directory and remove files that are more than 3 days old. These are usually files for work which can not be completed.

The following options are available.

**-d***dir* Scan directory *dir*.

**-m** Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned

by the owner of the **uucp** programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the **uucp** programs.)

**-n***hours* Change the aging time from 72 hours to *hours* hours.

**-p***pre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)

**-w***file* † Write warning about, but do not delete, files older than specified with **-n** above. Warnings are written on *file* if it is specified. Otherwise, they are sent to standard output.

**-s***sys* † Only files destined for system *sys* are examined. Up to 10 **-s** arguments can be specified.

**-x***num* This is the level of debugging output desired.

## 1.9 SECURITY

The **uucp** system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the **uucp** login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the **uucp** system.

- The login for **uucp** does not get a standard shell. Instead, the **uucico** program is started. Therefore, the only work that can be done is through **uucico**.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the "Files required for execution" section for the file description.)
- The file */usr/lib/uucp/L.cmds* is a list of commands that **uucp** considers legal for remote execution. The installer may modify this file as necessary. If the *L.cmds* file is missing, **uuxqt** uses a default list of legal commands. For the **bsd4.2** version, this list consists of **rmail**, **rnews**, and **uusend**. For the **sys5** version, this list consists solely of **rmail**. **uuxqt** program.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

## 1.10 INSTALLING UUCP ON DOMAIN SYSTEMS

In this section, we explain how to install **uucp** on a DOMAIN system. The information in this section is primarily of interest to system administration personnel.

### 1.10.1 The Installation Script

We supply two scripts for installing DOMAIN/IX at your site. One — the "administrative install," called *install_sysadmin* — has a subsection which handles installation of **uucp**. Before you run this script, you should determine which version (*sys5* or *bsd4.2*) of **uucp** you want to install. After the installation is complete, you will be asked to run shell scripts that add certain accounts to */etc/passwd* and set the access modes (and ACL's) on various filesystem objects, including **uucp** files.

The *inst_registry* script creates an account for a user named "uucp" with password "uucp_password" and home directory */usr/spool/uucppublic*. The *acl_* script sets required access modes for all files and programs used by the version of **uucp** you have decided to install. Once you have run these scripts, all you will need to do is choose a name for your installation.

### 1.10.2 Selecting a Name for the Local System

**Uucp** requires you to choose a name by which your DOMAIN system (uucp site) will be known to other uucp sites. Names can be any number of lowercase letters, although **uucp** will recognize the first seven (six †) characters and ignore the rest.

When the name has been selected, record it in the file */etc/net/uname*, then verify that the name can be accessed by running the command

```
% uuname -l
```

which displays the selected name.

**Note:** You will need to log in as either "root" or "uucp" in order to edit */etc/net/uname*.

### 1.10.3 Making Subdirectories

The *bsd4.2* version of **uucp** expects to find subdirectories of the form **D.***name* and **D.***name***X** in */usr/lib/uucp*. We provide a shell script, /usr/lib/uucp/mksubdirs.sh in the *bsd4.2* version of */usr*. Run this script after you have put the site name in */etc/net/uname*.

## 1.11 REQUIRED FILES

This section details the files that **uucp** needs to access in the course of its normal operations. These files are

- */usr/lib/uucp/L.sys*

- */usr/lib/uucp/L.dialcodes*
- */usr/lib/uucp/L.devices*
- */usr/lib/uucp/L.cmds*
- */usr/lib/uucp/USERFILE*
- */etc/net/uname*

**Note:** The field separator for all files is a space unless otherwise specified.

We have provided example versions of these files in the appropriate places in the DOMAIN/IX distribution filesystem. Examine these files as you read the following descriptions.

### 1.11.1 L-devices

This file contains entries for the call-unit devices and hardwired connections to be used by **uucp**. The special device files are assumed to be in the */dev* directory. The format for each entry is

*caller line call-unit speed dialer*

where;

*caller* is the type of device that will be making the connection. For *bsd4.2* **uucp**, typical specifications for caller are ACU (for Automatic Call Unit) or DIR (for a Direct Connection). The *sys5* version recognizes only ACUVADIC (the Vadic modem) and DIR (Direct Connection).

*line* is the device for the line (e.g. sio1).

*call-unit* is an unused field in DOMAIN systems.

*speed* is the line speed (baud rate).

*dialer* ‡ is the type of ACU used (e.g., hayes, vadic, or ventel for *bsd4.2*).

The *bsd4.2* version of **uucp** supports the following modems.

- Hayes
- Vadic
- Ventel

The *sys5* version supports only the Vadic modem.

There are example *L-devices* files in the */usr/lib/uucp* directory for each version of DOMAIN/IX.

### 1.11.2 L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

*abb dialseq*

where;

*abb* is the abbreviation,

*dial-seq* is the dial sequence to call that location.

There are example *L-dialcodes* files in the */usr/lib/uucp* directory for each version of DOMAIN/IX.

### 1.11.3 uname

This is the file (*/etc/net/uname*) where the system name resides. While the login name used by a remote host should not be the same as the login name of a local user, several remote computers may employ the same login name. Each **uucp** site is given a unique system name that is transmitted at the start of each call. This name identifies the calling machine to the called machine.

### 1.11.4 USERFILE

This file contains user accessibility information. It specifies which files can be accessed by a normal user of the local machine, which files can be accessed from a remote computer, which login name is used by a particular remote computer, and whether a remote computer should be called back in order to confirm its identity. Each line in USERFILE has the following format

*login,sys* [**c**] *pathname ...*

where;

*login* is the login name for a user or the remote computer,

*sys* is the system name for a remote computer,

**c** is the optional *call-back required* flag,

*pathname* is a pathname prefix that is acceptable for *user*.

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine (MASTER mode), the pathnames allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

[2] When the program is responding to a command from a remote machine (SLAVE mode), the pathnames allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a null system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same

login name but one of them must either have the name of the remote system or must contain a *null* system name.

[4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

u,m /usr/xyz

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

dan, /usr/dan

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines

u,m /usr/xyz /usr/spool
u, /usr/spool

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines

root, /
, /usr

allow any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file.

### 1.11.5 L.sys

Each entry in this file represents a system that your **uucp** site can call. The fields are described below.

*sysname* The name of the remote system.

*time* A string which indicates the days-of-week and times-of-day when the system should be called (e.g., MoTuTh0800-1730). Times may be stated using the key-words *Su Mo Tu We Th Fr Sa* for the seven days of the week, *Wk* for "any week-day," and *Any* for "any day." The *time* should be a range of times (e.g., 0800-1230). If no *time* is specified, **uucp** assumes that it may call at any time of day.

*device* This is either ACU (Automatic Call Unit) or DIR (DIRect connection). For the hardwired case, the last part of the special file name is used (e.g., sio1).

*speed* This is the line speed for the call (e.g., 300).

*phone* The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g., mh5900, boston995-9980). For the hardwired devices, this field contains the same string as used for the *device* field.

*login* The login information is given as a series of fields and subfields in the format

expect send expect send ...

where *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received. The *expect* field may be made up of subfields of the form

expect[–send–expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

Two special names can be sent during the login sequence. The string *EOT* will send an ASCII EOT character and the string *BREAK* will try to send an ASCII BREAK character. (**Uucp** simulates the BREAK character by using line-speed changes and null characters. This may not work on all devices and/or systems.) A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The *expect* algorithm looks at the last part of the string as illustrated in the password field.

There are example *L.sys* files in the */usr/lib/uucp* directory for each version of DOMAIN/IX.

## 1.12 ADMINISTRATION

This section explains the responsibilities of the **uucp** system administrator. Some administration can be accomplished by shell scripts that can be initiated by **crontab** entries. Others will require manual intervention. Some sample shell scripts are included in the */usr/lib/uucp* directory for each version of DOMAIN/IX.

### 1.12.1 TM – Temporary Data Files

These files are created in the spool directory while files are being copied from a remote machine. Their names have the form

**TM**.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number

starting at zero for each invocation of **uucico** and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; **uuclean** is useful in this regard. The following command line is from a *bsd4.2* **uucp** cleanup script.

% **uuclean -pTM -d/usr/spool/uucp/TM**

It removes all *TM* files that are more than three days old.

### 1.12.2 STST – System Status Files

These files are created in the spool directory by the **uucico** program. They contain information about failures in the areas of login, dialup or sequence check and will contain a *TALKING* status when two machines are conversing. The form of the file name is

**STST.***sys*

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

### 1.12.3 LCK – Lock Files

Lock files are created for each device in use (e.g., automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

**LCK..**str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

### 1.12.4 Shell Scripts

The **uucp** program will spool work and attempt to start the **uucico** program, but the starting of **uucico** will sometimes fail (no devices available, login failures etc.). Therefore, the **uucico** program should be periodically started. The command to start **uucico** can be put in a shell script and started by a *crontab* entry on an hourly basis. Here's an

example file for the *bsd4.2* environment.

```
: bsd4.2 Hourly uucp script

cd /usr/lib/uucp
: poll of sites you want to connect to hourly
uucico.real -r1 -ssys1
uucico.real -r1 -ssys2

: attempt to ship remaining files
uucico.real -r1
```

Note that the "-r1" option is required to start the **uucico** program in MASTER mode.

You can write another shell script that runs daily and removes *TM*, *ST* and *LCK* files as well as *C.* or *D.* files for work that could not be accomplished (e.g. bad phone number, login change, etc.). The following example is a typical *sys5* cleanup script.

```
:sys5 cleanup script
cd /usr/lib/uucp uuclean -pTM -pC. -pD.
uuclean -pST -pLCK -nl2
```

Here's another example. This one is available in the file */bsd4.2/usr/lib/uucp/uu.daily*.

```
: bsd4.2 daily uucp script
: assumes you have subdirectories.
cd /usr/lib/uucp
uuclean -pLTMP. -n24
uuclean -d/usr/spool/uucp/TM. -pTM. -n240
uuclean -d/usr/spool/uucp/X. -pX. -n240
uuclean -d/usr/spool/uucp/C. -pC. -n240
uuclean -d/usr/spool/uucp/D. -pD. -n240
uuclean -d/usr/spool/uucp/D.`uuname -l` -pD. -n240
uuclean -d/usr/spool/uucp/D.`uuname -l`X -pD. -n240

SPOOL=/usr/spool/uucp
mv $SPOOL/LOGFILE $SPOOL/LOGFILE.old
```

This script is designed for use in the *bsd4.2* environment, since it looks in the subdirectories *D.`uuname`* and *D.`uuname`X*. Note that we use the "-n24" option to clean up *LTMP* files more than 24 hours old, and the "-n240" option on everything else. The absence of the "-n" option will use a three-day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s.

# Chapter 2: Mail

## 2.1 INTRODUCTION

**Mail** provides DOMAIN/IX users with 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 (for example, via ARPANET or USENET). The **Mail** program divides incoming mail into its constituent messages and allows you to 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.

This chapter describes how to use the **Mail** program to send and receive messages. 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.

**Note:** The **Mail** program we describe in this chapter is the one included with the *bsd4.2* version of DOMAIN/IX. It resides in */bsd4.2/usr/ucb/Mail*. The *sys5* version has its own mail program, */bin/mail*. It is described in the *DOMAIN/IX Programmer's Reference for System V*.

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 notifies you if there are any messages waiting in your system mailbox. If you are a C Shell user, you will be notified when new mail arrives if you inform the shell of the location of 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**, then you can make **csh** 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.

### 2.1.1 Sending Mail

To send a message to a user whose login name is *root*, use the shell command:

% **Mail root**

then type your message. When you reach the end of the message, type

RETURN followed by an EOF (End Of File), usually mapped to ↑Z. The DM (Display Manager) will echo

*** EOF ***

and send an End-of-File to **Mail**. This will cause **Mail** to echo

EOT

and return you to the shell.

**Note:** In this document, we assume that, if you are using a DOMAIN node, the sequence CTRL Z (or — as we usually show it — ↑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, in this case, passes it along to **Mail**.

The next time the person to whom the message was addressed logs in, the message:

You have mail.

will appear in the shell transcript pad.

If, while you are composing the message you decide that you do not wish to send it after all, you can kill the letter with an Interrupt signal (usually ↑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 abort the letter. Once you have sent mail to someone, there is no easy way to cancel the message.

The message your recipient reads will consist of the message you typed preceded by a line telling who sent the message (your login name) and the date and time it was sent.

If you want to send the same message to several other people, you can list their login names on the **mail** command line. Thus,

% **Mail sam bob john**
**Tuition fees are due next Friday. Don't forget!!**
**↑Z**
*** EOF ***
EOT
%

will send the reminder to users sam, bob, and john.

### 2.1.2 Receiving Mail

If, when you log in, you see the message,

You have mail.

You can read the mail by typing simply:

% Mail

**Mail** will respond by displaying its version number and date and then listing the messages you have waiting. Then, it will display a prompt and await your command. The messages are assigned numbers starting with 1. You must refer to a specific message using its number. **Mail** keeps track of which messages are *new* (have been sent since you last read your mail) and *read* (have been read by you). New messages have an **N** next to them in the header listing and old, but unread messages have a **U** next to them. **Mail** keeps track of 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, which may be abbreviated to simply **t**. For example, if you had the following messages:

N 1 root Wed Sep 21 09:21 "Tuition fees"
N 2 sam Tue Sep 20 22:55

you could examine the first message by giving the command:

**type 1**

which might cause **Mail** to respond with, for example,

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 like 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, for example,

**t**

to type the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

**1**

would type the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in **Mail** by simply typing 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 wish to immediately send a reply, you can do so with 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 then type your letter in reply, followed by a ↑Z at the beginning of a line, as before. **Mail** will echo "EOT", then type 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**

**Mail** responds by typing:

To: root
Subject: Re: Tuition fees

and waiting for you to enter your letter. You are now in the message collection mode described at the beginning of this section. **Mail** will gather 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 will tend to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found will also be used. For example, if the letter had a **To:** header listing several recipients, **Mail** would arrange to send your reply to the same people as well. Similarly, if the original message contained a **Cc:** (carbon copies to) field, **Mail** would send your reply to all those users, too. **Mail** will normally not send the message to you, even if your name appears in the **To:** or **Cc:** field, unless you ask to be included explicitly. We will cover this subject in more detail in a later section.

After typing in your letter, the dialog with **Mail** might look like this.

**reply**
To: root
Subject: Tuition fees

**Thanks for the reminder**
*** EOF ***
EOT
&

The **reply** command is especially useful for sustaining extended conversations over the message system, with other **listening** users receiving copies of the conversation. The **reply** command can be abbreviated to **r**.

Sometimes, you will receive a message that has been sent to several people and wish to reply *only* to the person who sent it. **Reply** with a capital **R** does the trick.

If you wish, while reading your mail, to send a message to someone, but not as a reply to one of your messages, you can send the message directly with the **mail** command, which takes as arguments the names of the

recipients you wish to send mail to. For example, to send a message to *frank*, you would do:

**mail frank**
**This is to confirm our meeting next Friday at 4.**
**↑Z**
*** EOF ***
EOT
&

The **mail** command can be abbreviated to **m**.

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* you can delete it using the **delete** command. In our example,

**delete 1**

will prevent **Mail** from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, **Mail** will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The **delete** command can be abbreviated to simply **d**.

Many features of **Mail** can be tailored to your liking with the **set** command. The **set** 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 you for a subject header, to be included in the message. To set the **ask** option, type

**set ask**

Another useful **Mail** option is **hold**. Unless told otherwise, **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, you can set the **hold** option.

Valued options set numeric or string values which **Mail** uses to adapt to your tastes. For example, the **SHELL** option tells **Mail** which shell you like to use, and is specified by

**set SHELL=/bin/csh**

for example. Note that no spaces are allowed in **SHELL=/bin/csh**. 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 will 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**. For example,

**set crt=24**

will pipe any message longer than 24 lines through **more**.

**Note:** If you are using **Mail** on a DOMAIN node, the **crt** option will not be necessary, since you will be able to scroll back through the message transcript at your 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 Sam, Sally, Steve, and Susan. To define an alias called **project** for them, you would use the **Mail** command:

**alias project sam sally steve susan**

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 had the login name *walukiewicz_c*, you might want to use:

**alias walukiewicz_c cindy**

so that you could avoid typing (and probably misspelling) the longer name walukiewicz_c.

You may create a special file of aliases and options that will be placed in effect automatically every time you invoke **Mail**. Whenever **Mail** is invoked, it first reads a system-wide file */usr/lib/Mail.rc*, then a user specific file, **.mailrc**, which is found in your home directory. The system-wide file is maintained by the system administrator and contains **set** commands that are applicable to all users of the system. You may create a *.mailrc* file set options and define individual aliases. A typical .mailrc file looks like this:

**set ask nosave SHELL=/bin/csh**

As you can see, it is possible to set many options in the same set command. The **nosave** option is described in section 5.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail* (documented in the appendices to this manual.) These aliases are stored in the file */usr/lib/aliases* and are accessible to all users of the system. The lines in /usr/lib/aliases have the form:

**alias:** *name1, name2, name3*

where *alias* is the mailing list name and the *namei* are the members of the list. Long lists can be continued 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 with:

% **Mail -f letters**

You can use all the **Mail** commands described in this document to examine, modify, or delete messages from the file *letters*, which will be 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 using simply

% **Mail -f**

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. If you wish 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 do not examine are normally retained in your system mailbox. If you wish to have such a message saved in *mbox* without reading it, use the **mbox** command. For example,

**mbox 2**

in our example would cause 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 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 will see them the next time you type:

% **Mail**

The **quit** command can be abbreviated to simply **q**.

If you wish to leave **Mail** without altering either your system mailbox or *mbox*, you can type the **x** command (short for **exit**), which will immediately return you to the shell without changing anything.

To execute a shell command without leaving **Mail**, you can type the command preceded by an exclamation point, just as in the **vi** text editor. For example,

**!date**

will display the current date without leaving **Mail**.

Finally, the **help** command is available to print out a brief summary of the **Mail** commands, using only the single character command abbreviations.

## 2.2 MAINTAINING FOLDERS

**Mail** includes a simple facility for maintaining groups of messages together in folders.

To use the folder facility, you must tell **Mail** where you wish to keep your folders. Each folder of messages will be 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 form

**set folder=letters**

in your *.mailrc* file. If, as in the example above, your folder directory does not begin with a "/", **Mail** will look 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 file name is expected, you can use a folder name, preceded with "+". For example, to put a message into a folder with the **save** command, you can use:

**save +letters**

to save the current message in the *letters* folder. If the *letters* folder does not yet exist, it will be created. Note that messages which are saved with the **save** command are automatically removed from your system mailbox.

In order 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,

**copy +letters**

copies the current message into the *letters* folder and leaves a copy in your system mailbox.

The **folder** command can be used to direct **Mail** to the contents of a different folder. For example,

**folder +letters**

directs **Mail** to read the contents of the *letters* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including **type**, **delete**, and **reply**. To inquire which folder you are currently editing, use simply

**folder**

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,

% **Mail -f +classwork**

will cause **Mail** to read your *classwork* folder without looking at your system mailbox.

## 2.3 TILDE ESCAPES

While typing in a message, it is often useful 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**

which will print 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 ~ 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**

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** will respond by typing

(continue)

after which you may continue typing text which will be appended 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. For example, you might prefer:

**set EDITOR=/bin/ex**

To use the *screen* editor on your current message, you can use the escape,

**~v**

~v works like ~e, except that **vi** is invoked instead. A default screen editor is defined by **Mail**. If it does not suit you, you can set the valued option **VISUAL** to the pathname of a different editor.

It is often useful to be able to include the contents of some file in your message; the escape

**~r filename**

is provided for 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 like * and ? which are expanded according to the conventions of your shell.

As a special case of ~r, the escape

**~d**

reads in the file **dead.letter** in your home directory. This is often 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, you may use the

**~w filename**

escape. **Mail** will print 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**

which will read 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 ...**

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will 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**

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 ...**

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. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape

**~h**

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 can also 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 # and @ symbols,

**~h**
To: root ers####eve

would change 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.

To effect a temporary escape to the shell, the escape

**~!command**

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

**~|command**

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**

It is also useful for setting options and modifying aliases.

If you wish 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.**

sends the line

~This line begins with a tilde.

Finally, the escape

**~?**

prints out a brief summary of the available tilde escapes.

**Mail** allows you to change the escape character with the **escape** option. For example, the line

**set escape=]**

sets the escape character to a right bracket instead of a tilde. Doing this causes everything said above about the tilde to apply to the right bracket. Changing the escape character removes the special meaning of ~.

## 2.4 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.

### 2.4.1 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 of the form

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** (see Chapter 1 of this section), sending mail is a bit 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 that is 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.

### 2.4.2 Special Recipients

As described previously, you can send mail to either user names or **alias** names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a "/" in it or begins with a "+", it is assumed to be the path name 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 "/" would not usually be needed) you can precede the name with "./" So, to send mail to the file **memo** in the current directory, you can give the command:

% **Mail ./memo**

If the name begins with a "+," it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used 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, you might give the command:

alias project sam sally steve susan /usr/project/mail_record

Then, all mail sent to "project" would be saved on the file **/usr/project/mail_record** as well as being 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. For example one might write a project billboard program and want to access it using **Mail**. To send messages to the billboard program, one can send mail to the special name "|billboard" for example. **Mail** treats recipient names that begin with a "|" as a program to send the mail to. An **alias** can be set up to reference a "|" prefaced name if desired.

**Note:** The shell treats "|" specially, so it must be quoted on the command line. Also, the "| program" must be presented as a single argument to mail. The safest course is to surround the entire name with double quotes. This also applies to usage in the **alias** command. For example, if we wanted to alias "rmsgs" to "rmsgs -s" we would need to say:

alias rmsgs "| rmsgs -s"

### 2.4.3 Message lists

Several **Mail** commands accept a list of messages as an argument. Along with **type** and **delete**, described, there is the **from** command, which 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 below.

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, which 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, ^ 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 user names 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**, do

**type root**

As a shorthand notation, you can specify simply ***** to get every relevant message. Thus,

**type ***

prints all undeleted messages,

**delete ***

deletes all undeleted messages, and

**undelete ***

undeletes all deleted messages.

You can search for the presence of a word in subject lines with /. For example, to print the headers of all messages that contain the word **PASCAL**, do:

**from /pascal**

Note that subject searching ignores upper/lowercase differences.

## 2.5 SUMMARY OF COMMANDS

**!** Used to preface a command to be executed by the shell.

**-** The command goes to the previous message and prints it. The command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

**Print** Like **print**, but also prints out ignored header fields. See also **print** and **ignore**.

**Reply** Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). 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 will go *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 to messages that were sent to enormous distribution groups when you really just want to send a message to the originator. Use it often.

**Type** Identical to the **Print** command.

**alias** Defines a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example,

**alias project john sue willie kathryn**

creates an alias *project* which expands to the four people John, Sue, Willie, and 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 that were sent to one of these alternate names, **Mail** will not 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** The **chdir** command allows you to change your current directory. **Chdir** takes a single argument, which is taken to be the pathname of the new working directory. If no argument is given, **chdir** changes to your home directory.

**copy** The **copy** command does the same thing that **save** does, except that it does not mark relevant messages for deletion when you quit.

**delete** Deletes a list of messages. Deleted messages can be reclaimed with the **undelete** command.

**dt** The **dt** command deletes the current message and prints the next message.

**edit** To edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of messages as described under the **type** command 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. Edit the message and execute the editors "write and quit" command to make **Mail** read the message back and remove Message$x$. **Edit** 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 you were reading. Thus, if you accidentally delete messages you should have saved, you can use **exit** to recover.

**file** The same as **folder**.

**folders** Lists the names of the folders in your folder directory.

**folder** The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read the new file. Some special conventions are

recognized for the name:

| Name | Meaning |
|---|---|
| # | Previous file read |
| % | Your system mailbox |
| %name | *Name*'s system mailbox |
| & | Your ~/mbox file |
| +folder | A file in your folder directory |

**from** The **from** command takes a list of messages and prints out the header lines for each one; hence

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:** header field of each message, if present. In addition, **Mail** tags the message header of each message that has been the object of the **preserve** command with a **P**. Messages that have been **saved** or **written** are flagged with a *****. Finally, **deleted** messages are not printed at all. If you wish to reprint the current list of message headers, you can do so with 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. You can move **Mail's** notion of the current window directly to a particular message by using, for example,

**headers 40**

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 will happen by default.

**if** Commands in your **.mailrc** file can be executed conditionally depending on whether you are sending or receiving mail with the **if** command. For example, you can do:

```
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*. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as *Via* which are not usually of interest. The **Type** and **Print** commands can be used to print a message in its 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. If you have the *ask* option set, **Mail** will prompt you for a subject to your message. Then you can type in your message, using tilde escapes as described in section 4 to edit, print, or modify your message. To signal your satisfaction with the message and send it, type ↑Z at the beginning of a line, or a **.** 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. This is the default action for messages if you do *not* have the *hold* option set.

**next** The **next** command 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**

goes to the next message sent by **root** and types it. The **next** 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 the magic characters ↑ **.** or **$**. Thus,

```
.
```

prints the current message and

```
4
```

prints message 4, as described previously.

**preserve** Same as **hold**. Causes a list of messages to be held in your system mailbox when you quit.

**quit** Leave **Mail** and update the file, folder, or system mailbox you were reading. Messages that you have examined are marked as **read** and messages that existed when you started are marked as **old**. If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and you did not have *hold* set, all messages which have not been deleted, saved, or preserved will be moved to the file *mbox* in your home directory.

**reply** Frame a reply to a single message. The reply will be sent to the person who sent you the message to which you are replying, plus all the people who received the original message, except you. 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 was already prefaced with **Re:**. If the original message included a **reply-to** header field, the reply will go *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** It is often useful to be able to save messages on related topics in a file. The **save** command gives you ability to do this. The **save** command takes as argument a list of message numbers, followed by the name of the file on which to save the messages. The messages are appended to the named file, thus allowing one to keep several messages in the file, stored in the order they were put 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
```

**Saved** messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, 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

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** The **shell** command allows you to escape to the shell. **Shell** invokes an interactive shell and allows you to type commands to it. When you leave the shell, you will 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

**source** The **source** command reads **Mail** commands from a file. It is useful when you are trying to fix your **.mailrc** file and you need to re-read it.

**top** The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**. If you wish, you can change the number of lines that **top** prints out by setting the valued option **toplines**. On a CRT terminal,

set toplines=10

might be preferred.

**type** Print a list of messages on your terminal. If you have 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 will be piped through *more*.

**undelete** The **undelete** command causes a message that had been deleted previously to regain its initial status. Only messages that have been deleted may be undeleted. This command may be abbreviated to **u**.

**unset** Reverses the action of setting a binary or valued option.

**visual** It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The operation of the **visual** command is otherwise identical to that of the **edit** command. Both the **edit** and **visual** commands assume some default text editors. These default editors can be overridden by the valued options **EDITOR** and **VISUAL** for the standard and screen editors. You

might want to do:

set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi

**write** The **save** command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the **write** command. The **write** command has the same syntax as the **save** command, and can be abbreviated to simply **w**. Thus, we could write the second message by doing:

**w 2 file.c**

As suggested by this example, the **write** command is useful for such tasks as sending and receiving source program text over the message system.

**z** **Mail** presents message headers in windowfuls as described under the **headers** command. You can move **Mail's** attention forward to the next window by giving the

**z+**

command. You can move to the previous window with:

**z-**

## 2.6 CUSTOM OPTIONS

Throughout this manual, we have seen examples of both binary and valued options. This section describes each of the options in alphabetical order, including some we have not yet discussed.

**EDITOR** The valued option **EDITOR** defines the pathname of the text editor to be used in the **edit** command and ~e. If not defined, a standard editor is used.

**SHELL** The valued option **SHELL** gives the path name of your shell. This shell is used for the **!** command and ~! escape. In addition, this shell expands file names with shell metacharacters like * and ? in them.

**VISUAL** The valued option **VISUAL** defines the pathname of your screen editor for use in the **visual** command and ~v escape. A standard screen editor is used if you do not define one.

**append** The **append** option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, **Mail** will put messages in *mbox* in the same order that the system puts messages in your system mailbox. By setting **append**, you are requesting that new messages be put at the end of *mbox* regardless of the order in which they were received.

**ask** **Ask** is a binary option which causes **Mail** to prompt you for the subject of each message you send. If you respond by typing RETURN, no subject field will be sent.

**askcc** **Askcc** is a binary option which causes you to be prompted for additional carbon copy recipients at the end of each message. Type RETURN to use the current list.

**autoprint** **Autoprint** is a binary option which causes the **delete** command to behave like **dp** (after deleting a message, the next one will be typed).

**debug** The binary option **debug** causes debugging information to be displayed. Use of this option is the same as using the **-d** command line flag.

**dot** **Dot** is a binary option which, if set, causes **Mail** to interpret a period alone on a line as the terminator of a message you are sending.

**escape** To allow you to change the escape character used when sending mail, you can set the valued option **escape**. 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. If you change your escape character, then ~ loses all its special meaning, and need no longer be doubled at the beginning of a line.

**folder** The name of the directory to use for storing folders of messages. If this name begins with a "/", **Mail** considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

**hold** The binary option **hold** 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** The binary option **ignore** causes ↑I characters from your terminal to be ignored and echoed as @'s while you are sending mail, ↑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 as described in section 6.

**ignoreeof** An option related to **dot** is **ignoreeof** which makes **Mail** refuse to accept a ↑Z as the end of a message. **Ignoreeof** also applies to **Mail** command mode.

**keep** The **keep** 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*

**keepsave** When you **save** a message, **Mail** usually discards it when you **quit**. To retain all saved messages, set the **keepsave** option.

**metoo** When sending mail to an alias, **Mail** makes sure that if you are included in the alias, that mail will not be sent to you. This is useful if a single alias is being used by all members of the group. If however, you wish to receive a copy of all the messages you send to the alias, you can set the binary option **metoo**.

**noheader** The binary option **noheader** 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** Normally, when you abort a message with two ↑I signals, **Mail** copies the partial letter to the file **dead.letter** in your home directory. Setting the binary option **nosave** prevents this.

**quiet** The binary option **quiet** suppresses the printing of the version when **Mail** is first invoked, as well as printing the for example **Message 4:** from the **type** command.

**record** The valued option **record** can be set to the name of a file in which outgoing mail will be saved. Each new message you send is appended to the end of the file.

**screen** When **Mail** initially prints the message headers, it determines the number to print by looking at the speed of your terminal interface. The faster the baud rate, the more it prints. The valued option **screen** overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the **z** command.

**sendmail** To alternate delivery system, set the **sendmail** option to the full pathname of the program to use.

**toplines** The valued option **toplines** defines the number of lines that the **top** command will print out instead of the default five lines.

**verbose** The binary option "verbose" causes **Mail** to invoke sendmail with the **-v** flag, which causes it to go into verbose mode and announce expansion of aliases, etc. Setting the "verbose" option is equivalent to invoking **Mail** with the **-v** flag.

## 2.7 COMMAND LINE OPTIONS

This section describes command line options for **Mail**.

-N Suppresses the initial printing of headers.

-d Turns on debugging information.

-f *file* Shows the messages in *file* instead of your system mailbox. If *file* is omitted, Mail reads *mbox* in your home directory.

-i Ignores tty interrupt signals. Useful when connecting on noisy phone lines, which may generate spurious interrupt characters. It's usually more effective to change your interrupt character to ↑C using the *stty* shell command.

-n Inhibits reading of /usr/lib/Mail.rc. Not generally useful, since /usr/lib/Mail.rc is usually empty.

-s *string* Used for sending mail. *String* is used as the subject of the message being composed. If *string* contains blanks, you must surround it with quote marks.

-u *name* Read *name*'s mail instead of your own. Essentially, **-u** *user* is a shorthand way of doing **-f /usr/spool/user**.

-v Use the **-v** flag when invoking sendmail. This feature may also be enabled by setting the option "verbose".

**Note:** The following command line flags are also recognized, but are intended for use by programs (not users) invoking **Mail**,

-T *file* Arranges to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. **-T** is for the *readnews* program and should not be used for reading your mail.

-h *number* Passes on hop count information. **Mail** will take the *number*, increment it, and pass it with **-h** to the mail delivery system. **-h** only has effect when sending mail and is used for network mail forwarding.

-r *name* Used for network mail forwarding: interpret *name* as the sender of the message. The *name* and **-r** are simply sent along to the mail delivery system. Also, Mail will wait for the message to be sent and return the exit status. Also restricts formatting of message.

Note that **-h** and **-r**, which are for network mail forwarding, are not used in practice since mail forwarding is handled separately.

## 2.8 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 described in section 3 of the *DOMAIN/IX Programmer's Reference for BSD4.2*. 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 header field line is 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 any 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 message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

It should be noted that some network transport protocols enforce limits to the lengths of messages.

## 2.9 SUMMARY OF COMMANDS, OPTIONS, AND ESCAPES

This section gives a quick summary of the **Mail** commands, binary and valued options, and tilde escapes.

The following table describes the commands:

| Command | *Description* |
|---|---|
| **!** | 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, don't include headers |
| **z** | Scroll to next/previous screenful of headers |

The following table describes the options and indicates whether an option is binary or valued.

| Option | *Type* | *Description* |
|---|---|---|
| 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 . alone on line to terminate message input |
| escape | *valued* | Escape character to be used instead of ~ |
| folder | *valued* | Directory to store folders in |
| 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 **save**d 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 and 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 |

The following table summarizes the tilde escapes available while sending mail.

| Es-cape | *Argu-ments* | *Description* |
|---|---|---|
| ~! | *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 ~ in front of *string* |

The following table shows the command line flags that **Mail** accepts:

| Flag | Description |
|---|---|
| -N | Suppress the initial printing of headers |
| -T *file* | Article-id's of read/deleted messages to *file* |
| -d | Turn on debugging |
| -I *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 by programs that call mail, not by people.

# Index

## SECTION 4

# SUPPORT TOOLS

CONTENTS

# Chapter 1: Awk—A Pattern Scanning and Processing Language

## 1.1 INTRODUCTION

**Awk** is a programming language that enables you to prepare programs that search a file or set of files for patterns, then perform actions on lines or parts of lines that contain instances of those patterns. **Awk** makes certain data selection and transformation operations easy to express; for example, the following very simple **awk** program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

**Awk** patterns may include arbitrary Boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, if-else, while, and for statements, and multiple output streams.

This chapter explains how to write **awk** programs. It also includes a discussion of the design and implementation of **awk**, intended to furnish insight into the way UNIX software development tools can be combined to produce programs for specific tasks.

## 1.2 OVERVIEW

**Awk** is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

When invoked, **awk** scans a set of input lines (usually from a specified *file*) in order, searching for instances of *patterns* specified in the *program*. For any pattern, an *action* can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX program **grep**[1] will recognize the approach, although in **awk** the patterns may be more general than in **grep**, and while **grep** allows only one action (print the line), **awk**

provides the programmer with a variety of actions that may be taken on all or part of a line in which the matching pattern occurs. For example, the **awk** program

```
{print $3, $2}
```

prints the third and second fields of an input line in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

### 1.2.1 Usage

The command line

**awk '***program***'** [*input_file(s)*]

executes the **awk** commands in the *program* string on the named *input_file(s)*.

**Note:** When you include the **awk** program in the command line, it must be delimited by single quotes, as shown above, so that the shell knows that the entire program is the first argument to **awk**.

As is the case with other UNIX programs, **awk** reads the standard input if no file is specified, or if the "file" specified is "-", as shown below.

**awk '***program***' -**

If *program* is more than a few statements long, you may want to place it in a file and execute it by including the **-f** option on the **awk** command line, as shown below.

**awk -f** *program_file input_file(s)*

### 1.2.2 Program Structure

An **awk** program is a sequence of statements of the form:

*pattern* { *action* }
*pattern* { *action* }
...

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all specified patterns have been tested against the contents of the first input line, the next line is fetched and the matching process starts again.

Either the pattern or the action may be left out of an **awk** program line, but not both. If there is no action for a pattern, **awk** simply copies all matching input line(s) to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

### 1.2.3 Records and Fields

**Awk** divides each input file into records terminated by a record separator. The default record separator is the newline, so by default **awk** processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into *fields*. Fields are normally separated by white space (blanks or tabs), although the input field separator may be changed to any other character by resetting the *FS* variable as described below. Fields are referred to as **$1**, **$2**, and so forth, where $1 is the first field, $2 is the second field, and **$0** is the entire input record. Fields may be assigned to a numeric or string value. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators. These may be changed to another (single) character at any time. The optional command-line argument **-F** *c* may also be used to set **FS** to a character represented here by *c*.

If the record separator is empty, an empty input line is taken as the record separator. Blanks, tabs, and newlines are then treated as field separators.

The variable *FILENAME* contains the name of the current input file.

### 1.2.4 Printing

If an action has no pattern, the action is executed for **all** input lines (records). The simplest action is to use the **awk** command print to print some or all of a record. The simple *awk* program below

```
{ print }
```

prints each input record. It merely copies the input to the output — something to which *cat* is far better suited. A more useful **awk** program might print a field or possibly selected fields from each record. For instance, the program

```
{ print $2, $1 }
```

prints the first two fields of each input record (since no pattern has been specified) in reverse order. Items separated by a comma in the *print* statement will be separated by the current output field separator when

printed. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined numeric variables **NF**, (Number of Fields) and **NR**, (Number of Records) have a number of uses. For example, the program

```
{ print NR, NF, $0 }
```

prints each record preceded by its record number and the number of fields it contains.

Output may be diverted to multiple files; the program

```
{ print $1  >"foo1"; print $2 >"foo2"  }
```

writes the first field on the file *foo1* and the second field on file *foo2*. The >> notation familiar to UNIX users can be used to append **awk** output to a file. The program

```
{ print $1  >>"foo"  }
```

appends the first field of every input record to the file *foo*.

**Note:** When printing or appending output to a file, **awk** creates the specified output file if it does not already exist.

The filename can be derived from a variable or a field as well as a constant; for example,

```
{ print $1  >$2 }
```

uses the contents of field 2 of the current input record as the output file name.

**Note:** You may not specify more than **10** output files in an **awk** program.

**Awk** output can also be piped into another process; for instance,

```
{ print | "mail bob"  }
```

mails the current input record to mail user **bob**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

**Awk** also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8-digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number followed by a newline. Output separators are not produced automatically; you must add them yourself, as in this example. The **awk** version of **printf** is identical to that used in the C programming language.

## 1.3 PATTERNS

A pattern to the left of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of all three.

### 1.3.1 The BEGIN and END Patterns

The special pattern BEGIN matches the beginning of the input, before the first record is read. The special pattern END matches the end of the input, after the last record has been processed. BEGIN and END provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN { FS = ":" }
... body of program ...
```

The line below will finish an **awk** program by displaying a count of input lines.

```
END  { print NR }
```

**Note:** If BEGIN is present, it must be the first pattern; If END is used, it must be the last pattern.

### 1.3.2 Regular Expressions

The simplest regular expression is a literal string of characters delimited by slashes, for example

```
/smith/
```

This is actually a complete awk program which will print all lines which contain any occurrence of the name "smith". Lines that contain "smith" as part of a larger word (e.g., blacksmithing) will also be printed.

Awk regular expressions include the regular expression forms found in the UNIX text editor **ed**[1] as well as those used by **grep**[1] (without back-referencing). In addition, **awk** allows parentheses for grouping, the vertical line

|

to separate alternatives,

+

for "one or more", and

**?**

for "zero or one." All of these usages should be familiar to **lex**[1] users. Character classes may be abbreviated:

[a-zA-Z0-9]

matches the set of all letters and digits. As an example, the brief **awk** program below

/[Aa]ho|[Ww]einberger|[Kk]ernighan/

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether or not the first letter of the name is capitalized.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in **ed**[1] and **sed**[1]. Within a regular expression, blanks and the regular expression metacharacters are significant. To escape a regular expression character and restore its "real" meaning, precede it with a backslash. The pattern

/\/.*\//

matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

$1 ~ /[jJ]ohn/

prints all lines where the first field matches "john" or "John." Note that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

$1 ~ /^[jJ]ohn$/

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

### 1.3.3 Relational Expressions

An **awk** pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

$2 > $1 + 100

which selects lines where the second field is at least 100 greater than the first field. Similarly,

NF % 2 == 0

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise, a numeric comparison is made. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

### 1.3.4 Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators || (or), **&&** (and), and **!** (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". **&&** and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

### 1.3.5 Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2 { ... }
```

In this case, the action is performed for each line starting at an occurrence of **pat1** and ending at the first subsequent occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

## 1.4 ACTIONS

An **awk** action is a sequence of one or more action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks, many of which will be described in this section.

### 1.4.1 Built-in Functions

**Awk** provides a "length" function to compute the length of a string of characters. The program below prints each record preceded by its length:

```
{ print length, $0 }
```

**length** by itself is a "pseudo-variable" which yields the length of the

current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{ print length($0), $0 }
```

The argument may be any expression.

**Awk** also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e*, logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**.

### 1.4.2 Variables, Expressions, and Assignments

**Awk** variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns the value 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by

```
    { s1 += $1; s2 += $2 }
END  { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, **-**, ***, /, and % (mod). The C increment ++ and decrement **--** operators are also available, as are the assignment operators +=, **-**=, ***=, /=, and %=. These operators may all be used in expressions.

### 1.4.3 Field Variables

Fields in **awk** share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to a numeric or string value. **Awk** allows you, for example, to replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 >  1000)
      $3 = "too big"
  print
}
```

which replaces the third field by the string "too big" when the field exceeds an arbitrary size (in this case, 1000 characters), then prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the string **s** into **array(1)**, ..., **array(n)**. **Awk** returns a value indicating the number of elements found. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

### 1.4.4 String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

### 1.4.5 Arrays

Array elements are not declared; they are created as necessary. Subscripts may have **any** non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the **NR**-th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the **awk** program

```
    { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/  { x["apple"]++ }
/orange/ { x["orange"]++ }
END      { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

Any expression can be used as a subscript in an array reference. Thus

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array **x**.

Suppose each line of input contains two fields, a name and a non-zero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

```
    { amount[$1] += $2 }
END { for (name in amount)
        print name, amount[name] }
```

To sort the output, replace the last line by

```
print name, amount[name] | "sort"
```

### 1.4.6 Flow-of-Control Statements

**Awk** provides these flow-of-control statements: **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement earlier without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i  <= NF; i++)
        print $i
```

does the same job as the **while** statement above.

There is an alternate form of the for statement which is suited for accessing the elements of an associative array:

```
for (i in array)
        statement
```

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while**, or **for** can include relational operators like <, <=, >, >=, == ("is equal to"), and **!=** ("not equal to"); regular expression matches with the match operators **~** and **!~**; the logical operators ||, **&&**, and **!**; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes **awk** to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in **awk** programs: they begin with the character # and end with the end of the line, as in

```
print x, y # this is a comment
```

## 1.5 DESIGN

UNIX provides several programs that operate by passing input through a selection mechanism. **Grep**[1], one of the simplest, merely prints all lines which match a single specified pattern. **Egrep** provides more general patterns, i.e., regular expressions in full generality; and **fgrep** searches for a set of keywords with a particularly fast algorithm. The stream editor **sed**[1] applies most of the editing facilities of the editor **ed**[1] to a stream of input. None of these programs provide numeric capabilities, logical relations, or variables.

**Lex**[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of **lex**, however, requires a knowledge of C programming, and a **lex** program must be compiled and loaded before use, which discourages its use for one-shot applications.

**Awk** provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation, nor does it presuppose extensive knowledge of C. Finally, **awk** provides a convenient way to access fields within lines; it is unique in this respect.

**Awk** also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases, you can simply ignore the differences.

Most of the effort in developing **awk** went into deciding what it should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation), rather than on writing or debugging the code. The authors of the program (A. V. Aho, P. J. Weinberger, and B. W. Kernighan) tried to make the syntax powerful, easy to use, and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, **awk** usage seems to fall into two broad categories. One is what might be called "report generation"—processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

The actual implementation of **awk** uses several of the UNIX language development tools discussed in this section of the *User's Guide*. The grammar is specified with **yacc**, the lexical analysis is done by **lex**. The regular expression recognizers are deterministic, finite automata constructed directly from the expressions. An **awk** program is translated into a parse tree which is then directly executed by a simple interpreter.

# Chapter 2: Sed—the Stream Editor

## 2.1 INTRODUCTION

**Sed** is a non-interactive context editor designed to be especially useful in three cases:

1. To edit files too large for comfortable interactive editing;
2. To edit a file of any size where the sequence of editing commands is too complicated to be comfortably typed in interactive mode, and
3. To perform multiple "global" editing functions efficiently in one pass through the input.

This chapter is a manual for **sed** users.

Since only a few lines of the input reside in real memory at one time, and no temporary files are used, the effective size of a file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to **sed** as a command file. This often saves considerable typing, and provides a way to make special-purpose filters based on **sed**.

The principal loss of functionality in **sed**, as compared with an interactive editor, are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

**Sed** is a lineal descendant of the UNIX editor, **ed**. Because of the differences between interactive and non-interactive operation, considerable changes have been made between **ed** and **sed**; even confirmed users of **ed** will need to read this document before proceeding to use **sed**. The most striking family resemblance between the two editors is in the class of patterns ("regular expressions") they recognize. The code for matching patterns is copied almost verbatim from the code for **ed**, so the two programs behave identically in this respect.

## 2.2 NORMAL OPERATION

**Sed** by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line.

The general format of an editing command is:

**[address1,address2][function][arguments]**

One or both addresses may be omitted; the format of addresses is given in the next section. Any number of blanks or tabs may separate the addresses from the function. The function must be present. The arguments may be required or optional, depending on the function. Functions and arguments are discussed in a later section.

Tab characters and spaces at the beginning of lines are ignored.

### 2.2.1 Command-Line Flags

**Sed** recognizes three command line flags.

**-n** tells **sed** not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions.

**-e** tells **sed** to take the next argument as an editing command,

**-f** *name* tells **sed** to get its commands from file *name*. *Name* must be a file that contains editing commands, one to a line.

### 2.2.2 Order of Application of Editing Commands

Before any editing is done or any input file opened, all the editing commands given to **sed** are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands **t** and **b**. Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

## 2.3 THE PATTERN SPACE

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the **N** command.

### 2.3.1 Example 1

**Note:** Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text.

In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

The command

**2q**

will quit after copying the first two lines of the input. The output will be:

In Xanadu did Kubla Khan
A stately pleasure dome decree:

## 2.4 ADDRESSES — SELECTING LINES FOR EDITING

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands in braces as shown below.

{*commands*}

### 2.4.1 Line-Number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line that causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

### 2.4.2 Context Addresses

A context address is a pattern ("regular expression") enclosed in slashes ("/"). The regular expressions recognized by **sed** are constructed as follows:

[1] An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

[2] A circumflex "^" at the beginning of a regular expression matches the null character at the beginning of a line.

[3] A dollar-sign "$" at the end of a regular expression matches the null character at the end of a line.

[4] The characters "\n" match an imbedded newline character, but not the newline at the end of the pattern space.

[5] A period "." matches any character except the terminal newline of the pattern space.

[6] A regular expression followed by an asterisk "*" matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[7] A string of characters in square brackets "[ ]" matches any character in the string, and no others. If, however, the first character of the string is circumflex "^l", the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.

[8] A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

[9] A regular expression between the sequences "\(" and "\)" is identical in effect to the unadorned regular expression, but has side-effects which are described under the s command below and specification [10] immediately below.

[10] The expression "*d*" means the same string of characters matched by an expression enclosed in "\(" and "\)" earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th. occurrence of "\(" counting from the left. For example, the expression "^\(.*\)\1" matches a line beginning with two repeated occurrences of the same string.

[11] The null regular expression standing alone (e.g., "//") is equivalent to the last regular expression compiled.

To use one of the special characters (^ $ . * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash "\".

For a context address to "match" the input requires that the whole pattern within the address match some portion of the pattern space.

### 2.4.3 Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command, we list the maximum number of allowable addresses. It is an error for a command to have more addresses than the maximum number allowed.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and

including) the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses must be separated by a comma.

Some examples are offered below.

/an/ *matches lines 1, 3, 4 in our sample text*
/an.*an/ *matches line 1*
/^an/ *matches no lines*
/./ *matches all lines*
/\./ *matches line 5*
/r*an/ *matches lines 1,8, 4 (number = zero)*
/\(an\).*\1/ *matches line 1*

## 2.5 FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does.

**Note:** The angles around the arguments are not part of the argument, and should not be typed in actual editing commands.

### 2.5.1 Whole-Line-Oriented Functions

*(2)***d** **delete lines** This function deletes from the file (does not write to the output) all those lines matched by its address(es). As a side effect, no further commands are attempted on the deleted line. As soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

*(2)***n** **next line** The **n** function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued.

*(1)***a***text* **append** *text* Causes *text* to be written to the output after the line matched by its address. The **a** command is inherently multi-line. It must appear at the end of a line, and *text* may contain any number of lines. To preserve the one-command-to-a-line convention, interior newlines in *text* must be hidden by a backslash character ("\") immediately preceding the newline. The *text* argument is terminated by the first unhidden newline. Once an **a** function is successfully executed, *text* will be written to the output regardless of what later commands do to the line which triggered it.

The triggering line may be deleted entirely; *text* will still be written to the output. The *text* is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

*(1)***i***text* **insert** *text* The **i** function behaves like **a**, except that *text* is written to the output before, rather than after, the matched line.

*(2)***c***text* **change** *text* The **c** function deletes the lines selected by its address(es), and replaces them with *text*. Like **a** and **e**, lines in **c** must be followed by a newline hidden by a backslash; and interior new lines in *text* must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of *text* is written to the output, not one copy per line deleted. As with **a** and **i**, *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter. After a line has been deleted by a **c** function, no further commands are attempted on the line. If text is appended after a line by **a** or **r** functions, and the line is subsequently changed, the text inserted by the **c** function will be placed before the text of the **a** or **r** functions.

**Note:** Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in **sed** commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

### 2.5.2 Example 2

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this case, the same effect would be produced by either of the two following command lists:

```
n          n
i\         c\
XXXX       XXXX
d
```

### 2.5.3 The Substitute Function

The substitute function **s** changes parts of lines selected by a context search within the line. In prototype, it looks like this.

*(2)***s***pattern replacement flags*

The **s** function replaces the part of a line matched by *pattern* with the text of *replacement*. The *pattern* argument contains a pattern, exactly like the patterns in addresses. The only difference between *pattern* and a context address is that the context address must be delimited by slash ("/") characters; *pattern* may be delimited by any character other than space or newline. By default, only the first string matched by *pattern* is replaced. See the **g** flag below.

The *replacement* argument begins immediately after the second delimiting character of *pattern*, and must be followed immediately by another instance of the delimiting character. (Thus, there are exactly three instances of the delimiting character.)

The *replacement* is not a pattern, and the characters which are special in patterns do not have special meaning in *replacement*. Instead, other characters are special:

- **&** is replaced by the string matched by pattern

*d* (where *d* is a single digit) is replaced by the *d*th substring matched by parts of *pattern* enclosed in "\(" and "\)". If nested substrings occur in pattern, the *d*th is determined by counting opening delimiters ("\(").

As in patterns, special characters may be made literal by preceding them with backslash ("\").

The *flags* argument may contain the following flags:

**g** substitute *replacement* for all (non-overlapping) instances of *pattern* in the line. After a successful substitution, the scan for the next instance of *pattern* begins just after the end of the inserted characters; characters put into the line from *replacement* are not rescanned.

**p** print the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** function. Notice that if several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output, one for each successful substitution.

**w***filename* write the line to *filename* if a successful replacement was done. The w flag causes lines which are actually substituted by the s function to be written to a file named by filename. If filename exists before sed is run, it is overwritten; if not, it is created. A single space must separate w and filename. The possibilities of multiple, somewhat different copies of one input line being written are the same as for p. A maximum of 10 different filenames may be mentioned after w flags and functions.

### 2.5.4 Example 3

The following command,

```
s/to/by/w changes
```

when applied to our standard input, produces, on the standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file "changes":

```
Through caverns measureless by man
Down by a sunless sea.
```

If the **nocopy** option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the **g** flag, the command:

```
/X/s/an/AN/p
```

produces (in **nocopy** mode):

```
In XANadu did Kubla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubla KhAN
```

### 2.5.5 Input/output Functions

*(2)***p** **print** The print function writes the addressed lines to the standard output file. They are written at the time the **p**

function is encountered, regardless of what succeeding editing commands may do to the lines.

*(2)***w** *filename*

**write** on *filename* The write function writes the addressed lines to *filename*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate w from *filename*. A maximum of ten different files may be mentioned in write functions and **w** flags after **s** functions, combined.

*(1)***r** *filename* **read** the contents of *filename* The read function reads the contents of *filename* and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If **e** and **a** functions are executed on the same line, the text from the **a** functions and the **r** functions is written to the output in the order that the functions are executed.

Exactly one space must separate **r** from *filename*. If a file mentioned by **r** cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

**Note:** Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in **w** functions or flags; that number is reduced by one if any **r** functions are present. (Only one read file is open at one time.)

### 2.5.6 Example 4

Assume that the file *note1* has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
```

Then the following command:

```
/Kubla/r note1
```

produces:

In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

### 2.5.7 Multiple Input-Line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

*(2)***N** **Next** line The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

*(2)***D** **Delete** first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

*(2)***P** **Print** first part of the pattern space. Print up to and including the first newline in the pattern space.

The **P**, **I**, and **D** functions are equivalent to their lowercase counterparts if there are no imbedded newlines in the pattern space.

### 2.5.8 Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

*(2)***h** **hold** pattern space. The **h** function copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

*(2)***H** **Hold** and append to pattern space. The **H** function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

*(2)***g** **get** contents of hold area The **g** function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

*(2)***G** **Get** (and append) contents of hold area The **G** function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

*(2)***x** **exchange** The exchange command interchanges the contents of the pattern space and the hold area.

### 2.5.9 Example 5

The commands

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

### 2.5.10 Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

*(2)***!** Don't. The **!** command causes the next command (written on the same line), to be applied to all and only those input lines not selected by the address part.

*(2)***{***group***}** The grouping command "{" causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the "{" or on the next line. The group of commands is terminated by a matching "}" standing on a line by itself. Groups can be nested.

*(0)***:***label* The label function marks a place in the list of editing commands which may be referred to by **b** and **t** functions. The *label* may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

*(2)***b***label* Branch to label. The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same *label* was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted. A **b** function with no *label* is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning

on the new line.

*(2)***t***label* Test substitutions. The **t** function tests whether any successful substitutions have been made on the current input line; if so, it branches to *label*; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by reading a new input line, or executing a **t** function.

### 2.5.11 Miscellaneous Functions

*(1)*= The = function writes to the standard output the line number of the line matched by its address.

*(1)***q** The **q** (quit) function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

# Chapter 3: Lint — a C Program Checker

## 3.1 INTRODUCTION

**Lint** is a command which examines C source code, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects certain constructions which though technically "legal," are nonetheless wasteful, error-prone, or otherwise best avoided.

**Lint** accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between **lint** and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible, in part, because the compilers do not do sophisticated type checking, especially between separately-compiled programs. **Lint** takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This chapter discusses the use of **lint**, gives an overview of the implementation, and gives some hints on the writing of machine independent C code. It is based on a 1978 Bell Labs memorandum by S. C. Johnson.

### 3.1.1 Usage

Suppose there are two C source files, *file1.c* and *file2.c*, which are ordinarily compiled and loaded together. Then the command

**lint file1.c file2.c**

produces messages describing inconsistencies and inefficiencies in the programs. The command

**lint -p file1.c file2.c**

will produce, in addition to the above messages, additional messages that relate to the "portability" (to other operating systems and machines) of the programs. Replacing the **-p** by **-h** will produce messages about constructions that, though legal, are examples of poor (in **lint**'s opinion) programming style. You may use both flags

**lint -hp file1.c file2.c**

to get both types of messages.

Many of the facts that **lint** needs to establish may, in reality, be impossible to discover. For example, it may not be possible to know whether a

given function in a program ever gets called without also knowing the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the **lint** algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, **lint** assumes it can be called.

**Lint** tries to give only relevant information. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, **lint** loses credibility, and its "error" messages merely clutter up the output, obscuring other, possibly more important messages.

### 3.1.2 Unused Variables and Functions

As sets of programs evolve, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally help you to discover bugs; if a function does a necessary job and is never called, something is probably wrong.

**Lint** complains about variables and functions that are defined but not otherwise mentioned. An exception is variables that are declared through explicit **extern** statements but are never referenced; thus, the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the DOMAIN C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **-x** flag to the **lint** invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of complaints about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when **lint** is applied to some, but not all, files in a collection that is normally loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere

may be used. The **-u** flag may be used to suppress the spurious messages which might otherwise appear.

### 3.1.3 Set/Used Information

**Lint** attempts to detect cases where a variable is used before it is set. This is not easy to do. Many algorithms take a good deal of time and space, and still produce "error" messages about perfectly valid programs. **Lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow-of-control need not be discovered. This genre of complaint has its roots in stylistic, rather than actual, error. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables that are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

### 3.1.4 Flow of Control

**Lint** attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that can never be left at the bottom, detecting the special cases **while**( 1 ) and **for**(;;) as infinite loops. **Lint** also complains about loops that cannot be entered at the top. As is often true when **lint** makes false accusations, this condition may not be a bug, but it is an affront to good programming style.

Lint has an important area of blindness in the flow of control algorithm, it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which **lint** does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

A **break** statement that cannot be reached causes no message. Programs generated by **yacc**, and especially **lex**, may have literally hundreds of unreachable **break** statements. The **-O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up **lint**'s output. If you want to see these messages, invoke **lint** with the **-b** option.

### 3.1.5 Function Values

Sometimes functions return values that are never used; sometimes programs incorrectly use function "values" that have never been returned. **Lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

return( *expr* );

and

return;

statements is cause for alarm; **lint** will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f(a) {
        if ( a ) return ( 3 );
        g();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from **lint**. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. This bug has actually been observed in "working" programs where, by chance, the desired function value was computed in the function return register.

### 3.1.6 Type Checking

**Lint** enforces the type checking rules of C more strictly than the compilers do. The additional checking goes on in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional ( ? : ), and

relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, **!**=, and function arguments and return values.

### 3.1.7 Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1;
```

where *p* is a character pointer. **Lint** will have reason to complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for **lint** to continue to complain about this. On the other hand, if this code is to be truly portable, such constructs should be examined carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

### 3.1.8 Nonportable Character Use

On most C implementations, characters take on only positive values. **Lint** will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar( )) < 0 ) ....
```

works where the version of C allows a character to have a negative value, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, **lint** will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two-bit field declared as type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

### 3.1.9 Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy in some implementations. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

### 3.1.10 Unorthodox Constructions

**Lint** flags several perfectly legal, but somewhat unorthodox, constructions in the hope of promoting better code quality and clearer style, and even of pointing out bugs. The **-h** flag enables these checks. For example, in the statement

```
*p++;
```

the * does nothing; this provokes the message "null effect" from **lint**. In the following program fragment,

```
unsigned x ;
if( x < 0 ) ...
```

the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **Lint** will accuse you of making a "degenerate unsigned comparison" in these cases. If the code says

```
if( 1 != 0 ) ....
```

lint will report "constant in conditional context", since the comparison

of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and lint encourages this by an appropriate message.

Finally, when the **-h** flag is in force **lint** complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

### 3.1.11 Antiquated Syntax

There are several forms of older syntax which **lint** attempts to discourage. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . . ) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc. ) have no such ambiguities. To spur the abandonment of the older forms, **lint** complains about these old fashioned operators.

A similar issue arises with initialization. Older versions of C allowed

```
int x 1 ;
```

to initialize $x$ to 1. This also caused syntactic difficulties: for example,

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read some distance past $x$ in order to be sure

what the declaration really is. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

int x = -1 ;

This is free of any possible syntactic ambiguity.

### 3.1.12 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. On machines where double precision values may begin on any integer boundary, it is reasonable to assign integer pointers to double pointers. On other machines, double precision values must begin on even word boundaries; thus, not all such assignments make sense. **Lint** tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **-p** or **-h** flags are in effect.

### 3.1.13 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

**Lint** checks for the important special case where a simple scalar variable is affected. For example, the statement

$a[i] = b[i{+}{+}]$ ;

will draw the complaint:

warning: $i$ evaluation order undefined

## 3.2 IMPLEMENTATION DETAILS

**Lint** consists of two programs and a driver. The first program is a version of the Portable C Compiler. This compiler does lexical and syntax

analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, **lint** produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring together all information collected about a given external name. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of **lint**.

### 3.2.1 Portability

This section describes some of the differences between C implementations, and discusses the **lint** features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The loader will resolve these declarations and cause only a single word of storage to be set aside for a. Under some implementations, this is not feasible, so each such declaration causes a word of storage to be set aside and called a. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If **lint** is invoked with the **-p** flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. Names known externally to UNIX software have seven significant characters, with the upper/lowercase distinction preserved. On other systems, the number of characters used and the preservation of case distinction may not be handled the same way. This leads to situations where programs that run fine under UNIX encounter loader problems on other systems. **Lint -p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling. UNIX

uses eight-bit ASCII. Other systems may use other character lengths or even other encoding schemes (e.g., EBCDIC). Moreover, character strings go from high to low bit positions ("left to right") on some systems, and low to high ("right to left") on the others. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. Lint is of little help here, except to flag multi-character character constants.

Other problems are likely to arise in shifting or masking words. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on many machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed unsigned. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, **lint** is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, **lint** has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

### 3.2.2 Suppressing Unwanted Output

There are occasions when you want **lint** to refrain from citing various constructs that, while technically "wrong," are, in fact, there for a good reason. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of throttling **lint**'s output is often desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by **lint** when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, **lint** directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the **lint** directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to **lint**, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The **-v** flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file. We cover this topic in detail in the next section.

### 3.2.3 Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

**Lint** library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. **Lint** does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

## 3.3 SUMMARY OF LINT OPTIONS

The command currently has the form

**lint** [-options ] files... library-descriptors...

The following options are available.

**h** Perform heuristic checks

**p** Perform portability checks

**v** Don't report unused arguments

**u** Don't report unused or undefined externals

**b** Report unreachable **break** statements

**x** Report unused external declarations

**a** Report assignments of **long** to **int** or shorter

**c** Complain about questionable casts

**n** No library checking is done

**s** Same as **h**

# Chapter 4: Make—A Program for Maintaining Programs

## 4.1 INTRODUCTION

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g.,**Yacc**[1] or **Lex**[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can render obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

**Make** is a program that mechanizes many of the activities of program development and maintenance. If the information on inter-file dependencies and command sequences is stored in a file, the simple command

make

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — **make** — test . . .

**Make** is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. This chapter is a guide for users of **make**. It is based on the original Technical Report on **make** by S. I. Feldman of Bell Labs,

## 4.2 BASIC FEATURES

The basic operation of **make** is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. **Make** does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
	cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information were stored in a file named **makefile**, the command

```
make
```

would perform the operations needed to recreate prog after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

**Make** operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, **make** discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

The following (somewhat lengthy) description file is equivalent to the one above, but takes no advantage of **make**'s innate knowledge:

```
prog : x.o y.o z.o
	cc x.o y.o z.o -lS -o prog
x.o : x.c defs
	cc -c x.c
y.o : y.c defs
	cc -c y.c
z.o : z.c
	cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would simply announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise, the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases, one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

**Make** has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

**Note:** To get a dollar sign, escape it with another dollar sign. The sequence $$ is escaped to $.

All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command:

- $*
- $@
- $?
- $<

They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o prog
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex ("-ll") and the Standard ("-lS") libraries, since macro definitions on the command line override definitions in the description. (The shell requires that you quote arguments that include embedded blanks.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

## 4.3 DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains three types of information:

- macro definitions
- dependency information
- executable commands

There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a

sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the **make** command line (see below).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2...] :[:] [dependent1...] [; commands] [#...]
[(tab) commands] [#...]
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either

- after a semicolon on a dependency line or
- on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked.

In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

if a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate

invocation of the shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). **Make** normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "-i" flag has been specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set.

- $@ is set to the name of the file to be "made"
- $? is set to the string of names that were found to be younger than the target.

If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, **make** prints a message and stops.

### 4.4 USAGE

The **make** command takes four kinds of arguments: macro definitions, flags, description file names, and target file names. The prototypical **make** command line is:

**make** [ *flags* ] [ *macro definitions* ] [ *targets* ]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments are made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

**-i** Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

**-s** Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

**-r** Do not use the built-in rules.

**-n** No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

**-t** Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

**-q** Question. The make command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

**-p** Print out the complete set of macro definitions and target descriptions.

**-d** Debug mode. Print out detailed information on files and times examined.

**-f** Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f " arguments, the file named **makefile** or **Makefile** in the current directory is read.

**Note:** The contents of the description files override the built-in rules if they are present.

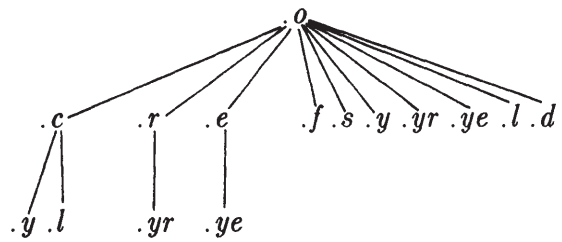
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

### 4.4.1 Implicit Rules

Make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is:

| | |
|---|---|
| *.o* | Object file |
| *.c* | C source file |
| *.e* | Efl source file |
| *.r* | Ratfor source file |
| *.f* | Fortran source file |
| *.s* | Assembler source file |
| *.y* | Yacc-C source grammar |
| *.yr* | Yacc-Ratfor source grammar |
| *.ye* | Yacc-Efl source grammar |
| *.l* | Lex source grammar |

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.




If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, then **make** would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

make CC=newcc

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

make "CFLAGS=-O"

causes the optimizing C compiler to be used.

### 4.4.2 An Example

As an example of the use of **make**, we present the description file used to maintain the make command itself. The code for make is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = und -3 | opr -r2 # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print: $(FILES)      # print recently changed files
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

**Make** usually prints out each command before issuing it. The following output results from typing the simple command

**make**

in a directory containing only the source and description file:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
        or
make print "P= cat >zap"
```

## 4.5 SUGGESTIONS AND WARNINGS

The most common difficulties arise from **make**'s specific understanding of what constitutes a dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what **make** would do, the "-n" option is very useful. The command

```
make -n
```

orders **make** to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the **-t** (touch) option can save a lot of time, instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

make -ts

("touch silently") causes the relevant files to appear up-to-date. Care is necessary, since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag **-d** causes **make** to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

## 4.6 SUMMARY OF SUFFIXES AND RULES

The **make** program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **-r** flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; **make** looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, **make** acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed.)

The following is an excerpt from the default rules file:

```
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
        $(CC) $(CFLAGS) -c $<
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
        $(AS) -o $@ $<
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

# Chapter 5: Lex — A Lexical Analyzer Generator

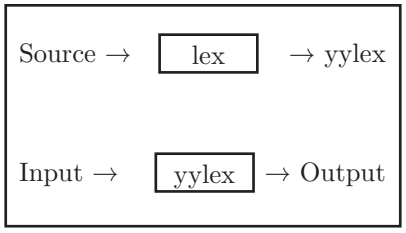
## 5.1 INTRODUCTION

**Lex** is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the programmer in the source specifications given to **lex**. The code written by **lex** recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete the tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

**Lex** is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, **lex** can write code in different host languages. The host language is used for the output code generated by **lex** and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes **lex** adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C. Although **lex** is a UNIX program, code generated by **lex** may be taken anywhere the appropriate compilers exist.

**Lex** turns input expressions and actions (known collectively as *source*), into the host general-purpose language; the generated program is named *yylex*. The *yylex* program recognizes expressions in a stream (*input*) and performs the specified actions for each expression as it is detected. The diagram below summarizes these features.




As an admittedly trivial example, consider the following program to delete from the input all blanks or tabs at the ends of lines.

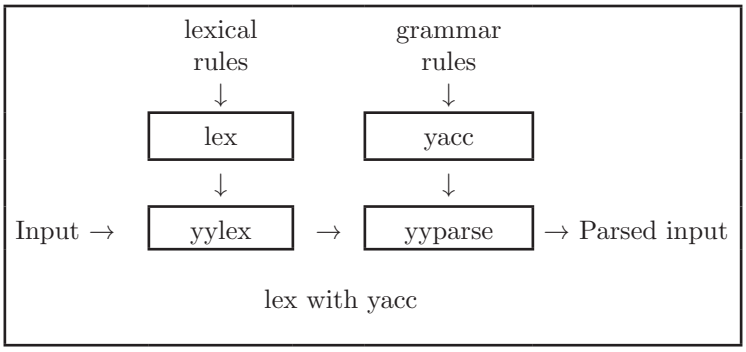
```
[ \t]+$xtx;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by **lex** (*yylex*) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[\t]+$xtx;
[\t]+xtxprintf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

**Lex** can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. **Lex** can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface **lex** and **yacc**[3]. **Lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. (See Chapter 6 for more on **yacc**.) When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown below. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**.




**Yacc** users will realize that the name *yylex* is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

**Lex** generates a deterministic finite automaton from the regular expressions in the source [4], The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number of **lex** rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex**.

In the program written by **lex**, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

**Lex** is not limited to source which can be interpreted on the basis of one-character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, **Lex** will recognize *ab* and leave the input pointer just before *cd*. This type of backing up is more costly than the processing of simpler languages.

## 5.2 LEX SOURCE

The general format of **lex** source is:

{definitions}
%%
{rules}
%%
{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum **lex** program is thus,

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of **lex** programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* and the right column contains *actions* — program fragments to be executed when the expressions are recognized. Thus, an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example, the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. **Lex** rules such as

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described later.

## 5.3 LEX REGULAR EXPRESSIONS

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

### 5.3.1 Operators

The operator characters are listed below.

- "
- \

- [
- ]
- ^
- -
- ?
- .
- *
- +
- |
- (
- )
- $
- /
- {
- }
- %
- <
- >

If any operator is to be used as a text character, you must escape it with quotes. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus,

xyz"++"

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz++"

is the same as the one above. If you quote every non-alphanumeric character being used as a text character, you can avoid having to remember the list of current operator characters, or having to worry that further extensions to **lex** might lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+\+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an

expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

### 5.3.2 Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special, these are \ - and ^. The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message, (e.g., [0-z] in ASCII is many more characters than it is in EBCDIC.) If you need to include the character - in a character class, it should be first or last; thus,

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus,

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters, and

```
[^a-zA-Z]
```

matches any character that is not a letter. The \ character provides the usual escapes within character class brackets.

### 5.3.3 Arbitrary Character Match

To match almost any character, the operator character "dot",

```
.
```

matches all characters except newline. Escaping into octal is possible although non-portable.

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

### 5.3.4 Optional Expressions

The question mark operator (**?**) indicates an optional element of an expression. Thus,

```
ab?c
```

matches either *ac* or *abc*.

### 5.3.5 Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

matches one or more instances of *a*. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

### 5.3.6 Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that we use parentheses for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abed*, or *abedef*.

### 5.3.7 Context Sensitivity

**Lex** recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special

case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string *ab*, but only if followed by *cd*. Thus,

ab$

is the same as

ab/\n

Left context is handled in **lex** by *start conditions*, which are explained later. If a rule is only to be executed when the **lex** automaton interpreter is in start condition *x*, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

### 5.3.8 Repetitions and Definitions

The brace operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example,

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the **lex** input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, an initial % is special, being the separator for **lex** source segments.

## 5.4 LEX ACTIONS

When an expression written as above is matched, **lex** executes the corresponding action. This section describes some features of **lex** which aid in writing actions.

**Note:** There is a default action — copy the input to the output — that is performed on all strings not otherwise matched.

The **lex** user who wishes to absorb the entire input without producing any output must provide rules to match everything. This is the normal situation when **lex** is being used with **yacc**. If you assume that an action is something **lex** does instead of the default action (copying the

input to the output), it follows that a rule which merely copies can be safely omitted. As a corollary to this, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

A simple "solution" is to ignore the input. Specifying a C null statement (**;**), as an action causes this result. A frequent rule is

```
[\t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is to use the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "     |
"\t"    |
"\n"    ;
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a-z]+*. **Lex** leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+      printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and s indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+      ECHO;
```

is the same as the above. Since the default action is just to print the characters found, why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule that matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a-z]+* is needed. This is explained further below.

It is sometimes more convenient to know the end of what has been found; hence **lex** also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, you might write

```
[a-zA-Z]+                   {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yylex*(n) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* that are to be retained. Additional characters that were previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

### 5.4.1 An Example

Consider a language that defines a string as a set of characters between quotation (") marks. To include a " in a string, you must escape it with a preceding \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]* {
        if (yytext[yyleng-1] == ´ \\ ´ )
        yymore(); else
        ... normal user processing
}
```

which will, when faced with a string such as *abc*\def" first match the five characters "*abc*\; then the call to *yymore()* will cause the next part of the string, "*def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=-[a-zA-Z]        {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
}
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively, it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]        {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
}
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^\t\n]
```

an even better rule.

In addition to these routines, **lex** also permits access to the I/O routines it uses. They are:

**input()** returns the next input character

**output(***c***)** writes the character *c* on the output

**unput(***c***)** pushes the character *c* back onto the input stream to be read later by **input()**.

By default these routines are provided as macro definitions, but you can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from unusual places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the lex lookahead will not work. **Lex** does not look ahead at all if it does not have to, but every rule ending in + ***** **?** or **$** or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by **lex**.

**Note:** The standard lex library imposes a 100-character limit on backup.

Another **lex** library routine that you may want to redefine is *yywrap()* which is called whenever **lex** reaches an end-of-file. If *yywrap* returns a 1, **lex** continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new

source. In this case, provide a *yywrap* that arranges for new input and returns 0. This instructs **lex** to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

## 5.5 AMBIGUOUS SOURCE RULES

**Lex** can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Assume that the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

have been given in that order. If the input is *integers*, it is taken as an identifier, because *[a-z]+* matches 8 characters while integer matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. In fact it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second'
```

here the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after '*first*'. The consequences of errors like this are mitigated by the fact that the . (dot) operator will not match a newline. Thus, expressions like

```
.*
```

stop on the current line. Don't try to defeat this with expressions like

```
[.\n]+
```

or equivalents; the generated program will try to read the entire input file, causing internal buffer overflows.

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that dot does not include newline. Since *she* includes *he*, **lex** will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

You may override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose you really want to count the included instances of *he*:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, you could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+    { ... ; REJECT;}
a[cd]+    { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is, the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* is to be incremented, the appropriate source is

```
%%
[a-z][a-z]          {digram[yytext[0]][yytext[1]]++; REJECT;}
\n          ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

## 5.6 LEX SOURCE DEFINITIONS

As we have stated, the format of **lex** source is:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. There is a need for additional options to define variables for use in user programs and for use by **lex**. These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source not intercepted by **lex** is copied into the generated program. Such source falls into three classes.

1. Any line which is not part of a **lex** rule or action which begins with a blank or tab is copied into the **lex** generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **lex** which contains the actions. This material must look like program fragments, and should precede the first **lex** rule. As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the **lex** output.

Definitions intended for **lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define **lex** substitution strings. The format of such lines is

```
name translation
```

where *translation* becomes associated with *name*. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D     [0-9]
E     [DEde][-+]?{D}+
%%
{D}+       printf("integer");
{D}+"."{D}*({E})?              |
{D}*"."{D}+({E})?              |
{D}+{E} printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/" ."EQ      printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. We discuss these possibilities further below in the "Summary of Source Format."

## 5.7 USAGE

There are two steps to compiling a **lex** source program. First, the **lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of **lex** subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use **lex** with **yacc** see below. Although the default **lex** I/O routines use the C standard library, the **lex** automata themselves do not do so; if private versions of *input*, *output*, and *unput* are given, the library can be avoided.

## 5.8 LEX AND YACC

If you want to use **lex** with yacc, note that what **lex** writes is a program named *yylex()*, the name required by yacc for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if yacc is loaded, and its main program is used, yacc will call *yylex()*. In this case, each **lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to yacc's names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line

```
# include "lex.yy.c"
```

in the last section of **yacc** input. If the grammar is named "good" and the lexical rules are named "better," the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (-ly) should be loaded before the **lex** library to obtain a main program which invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

## 5.9 MORE EXAMPLES

Consider copying an input file while adding 3 to every positive number divisible by 7. Here is a **lex** source program to do just that.

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
          printf("%d", k+3);
        else
          printf("%d",k);
        }
```

The rule [0-9]+ recognizes strings of digits; *atoi* converts the digits to

binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objectionable that this program alters such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one.

```
%%
      int k;
-?[0-9]+                    {
      k = atoi(yytext);
      printf("%d", k%7 == 0 ? k+3 : k);
      }
-?[0-9.]+                   ECHO;
[A-Za-z][A-Za-z0-9]+        ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.           |
\n          ;
%%
l s.
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; \++)
if (lengs[i] > 0)
printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input, it prints the table. The final statement *return(1);* indicates that **lex** is to perform wrapup. If *yywrap* returns zero (false), it implies that further input is available and the program is to continue reading and processing. If you provide a *yywrap* that never returns true, it will generate an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish uppercase and lowercase

letters, this routine begins by defining a set of classes including both cases of each letter:

```
a    [aA]
b    [bB]
c    [cC]
...
z    [zZ]
```

An additional class recognizes white space:

```
W    [\t]*
```

The first rule changes “double precision” to “real”, or “DOUBLE PRECISION” to “REAL”.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
printf(yytext[o]==’d’? ”real” : ”REAL”);
}
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^” ”[^ 0]                    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as “beginning of line, then five blanks, then anything but blank or zero.” Note the two different meanings of ^ . Next come a few rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ |
[0-9]+{W}”.”{W}{d}{W}[+-]?{W}[0-9]+ |
”.”{W}[0-9]+{W}{d}{W}[+-]}{W}[0-9]+ {
/* convert constants */
for(p=yytext; *p != 0; p++)
{
if (*p == ’d’ || *p == ’D’)
*p=+ ’e’- ’d’;
ECHO;
}
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *’e’-’d’*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. Next comes a series of names that must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}                    |
{d}{c}{o}{s}                    |
{d}{s}{q}{r}{t}                 |
{d}{a}{t}{a}{n}                 |
...
{d}{f}{l}{o}{a}{t}              printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}                    |
{d}{l}{o}{g}10                  |
{d}{m}{i}{n}1                   |
{d}{m}{a}{x}1                   {
            yytext[0] =+ 'a' - 'd';
            ECHO;
            }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}                {yytext[0] =+ 'r' - 'd';
                                ECHO;
                                }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*            |
[0-9]+      |
\n          |
.           ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

## 5.10 LEFT CONTEXT SENSITIVITY

It is sometimes desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often, the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another; the use of *start conditions* on rules, and the

possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all.

It may be more convenient, however, to have **lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag. The program below should be adequate.

```
            int flag;
%%
^a          {flag = 'a'; ECHO;}
^b          {flag = 'b'; ECHO;}
^c          {flag = V; ECHO;}
\n          {flag = 0 ; ECHO;}
magic       {
            switch (flag)
            {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
            }
            }
```

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definitions section with a line reading

%Start name1 name2 ...

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the < > brackets:

< name1 > expression

is a rule which is only recognized when **lex** is in the start condition *name1*. To enter a start condition, execute the action statement

BEGIN name1;

which changes the start condition to *name1*. To resume the normal state,

BEGIN 0;

resets the initial condition of the **lex** automaton interpreter. A rule may be active in several start conditions:

< name1,name2,name3 >

is a legal prefix. Any rule not beginning with the < > prefix operator is always active.

The previous example can be written another way.

```
%START AA BB CC
%%
^a    {ECHO; BEGIN AA;}
^b    {ECHO; BEGIN BB;}
^c    {ECHO; BEGIN CC;}
\n    {ECHO; BEGIN 0;}
<AA> magic                printf("first");
<BB> magic                printf("second");
<CC> magic                printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than the user's code.

## 5.11 CHARACTER SET

The programs generated by **lex** handle character I/O only through the routines *input*, *output*, and *unput*. Thus, the character representation provided in these routines is accepted by **lex** and employed to return values in *yytext*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, **Lex** must be given a *translation table*. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus, the example character table below

```
%T
1       Aa
2       Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

maps the lowercase and uppercase letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

## 5.12 SUMMARY OF SOURCE FORMAT

The general form of a **lex** source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

- Definitions, in the form "name space translation".
- Included code, in the form "space code".
- Included code, in the form

  ```
  %{
  code
  %}
  ```

- Start conditions, given in the form

  ```
  %S name1 name2 ...
  ```

- Character set tables, in the form

%T
number space character-string
...
%T

- Changes to internal array sizes, in the form

  %*x* *nnn*

  where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

| Letter | Parameter |
|---|---|
| P | positions |
| n | states |
| e | tree nodes |
| a | transitions |
| k | packed character classes |
| o | output array size |

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

| | |
|---|---|
| x | the character "x" |
| "x" | an "x", even if x is an operator. |
| \x | an "x", even if x is an operator. |
| [xy] | the character x or y. |
| [x-z] | the characters x, y or z. |
| [^x] | any character but x. |
| . | any character but newline. |
| ^x | an x at the beginning of a line. |
| <y>x | an x when **lex** is in start condition y. |
| x$ | an x at the end of a line. |
| x? | an optional x. |
| x* | 0,1,2, ... instances of x. |
| x+ | 1,2,3, ... instances of x. |
| x\|y | an x or a y. |
| (x) | an x. |
| x/y | an x but only if followed by y. |
| {xx} | the translation of xx from the definitions section. |
| x{m,n} | *m* through *n* occurrences of x |

## 5.13 CAVEATS

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the

only restriction on the user's ability to manipulate the not-yet-processed input.

# Chapter 6: Yacc (Yet Another Compiler Compiler)

## 6.1 INTRODUCTION

**Yacc** provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. **Yacc** then generates a function to control the input process. This function, called a "parser," calls the user-supplied low-level input routine (the "lexical analyzer") to pick up the basic items (called "tokens") from the input stream. These tokens are organized according to the input structure rules, called "grammar rules"; when one of these rules has been recognized, then user code supplied for this rule, an "action" is invoked; actions have the ability to return values and make use of the values of other actions.

**Yacc** is written in C, and the actions and output subroutine are in C as well. Moreover, many of the syntactic conventions of **Yacc** follow those used in C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ´,´ year ;

Here, *date*, *month_ name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_ name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a "terminal symbol," while the structure recognized by the parser is called a "nonterminal symbol." To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : ´J´ ´a´ ´n´ ;
month_name : ´F´ ´e´ ´b´ ;
 . . .
month_name : ´D´ ´e´ ´c´ ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_ name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond **yacc**'s ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_ name* was seen; in this case, *month_ name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month ´/´ day ´/´ year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The theory underlying **yacc** has been described elsewhere. **Yacc** has been extensively used in numerous practical applications, including the C Program checker **lint**.

## 6.2 BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. **Yacc** requires token names to be declared as such. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal **yacc** specification is

```
%%
rules
```

Blanks, tabs, and newlines may not appear in names or multi-character reserved symbols. Otherwise, they are ignored. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus,

´\n´ newline
´\r´ return
´\´´ single quote "´"
´\\´ backslash "\"
´\t´ tab
´\b´ backspace
´\f form feed
´\xxx´ "xxx" in octal

**Note:** For a number of technical reasons, the NUL character (´\0´ or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar "|" can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to **yacc** as

```
A: B C D
| E F
| G
;
```

It is not necessary for all grammar rules with the same left side to appear together in the grammar rules section, although it makes the input easier to read and change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 . . .
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the **%start** keyword:

%start symbol

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate. Usually, the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

### 6.3 ACTIONS

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A : ´( ´ B ´)´
{ hello( 1, "abc" ); }
```

and

```
XXX : YYY ZZZ
{ printf("a message\n");
flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A : B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr : ´(´ expr ´)´ ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr : ´(´ expr ´)´ { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. **Yacc** permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A : B
{ $$ = 1; } C
{ x = $2; y = $3; }
;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. **Yacc** actually treats the above example as if it had been written:

```
$ACT : /* empty */
{ $$ = 1; }
;

A : B $ACT C
{ x = $2; y = $3; }
;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written such that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the

index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr : expr ´+´ expr
{ $$ = node( ´+´,$1, $3); }
```

in the specification.

You may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The **yacc** parser uses only names beginning in "yy". You should avoid using such names in your own variables.

In these examples, all the values are integers. A discussion of other value types will be found in a later section.

## 6.4 LEXICAL ANALYSIS

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc**, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
extern int yylval;
int c;
c = getchar();
switch( c ) {
case ´0´:
case ´1´:
case ´9´:
yylval = c-´0´;
return( DIGIT );
}
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily-modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should be used carefully.

As mentioned above, the token numbers may be chosen by **yacc** or by the user. In the default situation, the numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

**Note:** For historical reasons, the endmarker must have token number 0 or negative.

This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

**Lex**, described in Chapter 5, was designed to work in close harmony with **yacc** parsers. The specifications for **lex** and other lexical analyzers use regular expressions instead of grammar rules. **Lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

## 6.5 HOW THE PARSER WORKS

**Yacc** turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here. The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

*Shift* is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right-hand side by the left-hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.reduce 18
```

refers to *grammar rule* 18, while the action

IF shift 34

refers to *state* 34.

Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left-hand symbol (A in this case), and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

A goto 20

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stack. The uncovered state is, in fact, the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be

followed by anything that would result in a legal input. The parser reports an error and attempts to resume parsing. The error recovery mechanism (as opposed to the one for error detection) will be covered in a later section.

As an initial example, consider the specification:

```
%token DING DONG DELL
%%
rhyme : sound place
;
sound : DING DONG
;
place : DELL
;
```

When **yacc** is invoked with the **-v** option, a file called *y.output* is produced, with an english-like description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
$accept : _rhyme $end

DING shift 3

rhyme goto 1
sound goto 2

state 1
$accept : rhyme_$end

$end accept

state 2
rhyme : sound_place

DELL shift 5

place goto 4

state 3
sound : DING_DONG

DONG shift 6

state 4
rhyme : sound place_ (1)


state 5
place : DELL_ (3)


state 6
sound : DING DONG_ (2)
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser as it processes this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is "shift 3", so state 3 is pushed onto the stack, and

the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

sound : DING DONG

This rule has two symbols on the right-hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

sound goto 2

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "Send" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

Consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spent examining this and other simple examples will help you understand the problems that can arise in more complicated contexts.

## 6.6 AMBIGUITY AND CONFLICTS

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr ´-´ expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

( expr - expr ) - expr

or as

expr - ( expr - expr )

(The first is called *left association*, the second *right association*).

**Yacc** detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

- expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr - expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

expr - expr - expr

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

expr - expr

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

expr - expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*.

**Note:** There are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever

it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

**yacc** invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives you marginal control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most earlier parser generators considered conflicts to be fatal errors. **Yacc** proceeds on the assumption that this rewriting is somewhat unnatural and produces slower parsers; thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat : IF ´(´ cond ´)´ stat
| IF ´(´ cond ´)´ stat ELSE stat
;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
IF ( C2 ) S1
}
ELSE S2
```

or

```
IF ( C1 ) {
IF ( C2 ) S1
ELSE S2
}
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right-hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (**-v**) option output **file**. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

stat : IF ( cond ) stat_ (18)
stat : IF ( cond ) stat_ELSE stat


ELSE shift 45
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by "**.**", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF ´(´ cond ´)´ stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there is reduce action possible in the state. This will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

## 6.7 PRECEDENCE

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left ´+´ ´-´
%left ´*´ ´/´
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in FORTRAN, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in **yacc**. As an example of the behavior of these declarations, the description

```
%right ´=´
%left ´+´ ´-´
%left ´*´ ´/´

%%

expr : expr ´=´ expr
| expr ´+´ expr
| expr ´-´ expr
| expr ´*´ expr
| expr ´/´ expr
| NAME
;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary ´-´; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. It appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left ´+´ ´-´
%left ´*´ ´/´

%%

expr : expr ´+´ expr
| expr ´-´ expr
| expr ´*´ expr
| expr ´/´ expr
| ´-´ expr %prec ´*´
| NAME
;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

## 6.8 ERROR HANDLING

Error handling is an extremely difficult area. Many of the problems found here are semantic. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **Yacc** provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the

parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc..

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat : error ´;´
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ´;´. All tokens after the error and before the next ´;´ cannot be shifted, and are discarded. When the ´;´ is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error ´\n´ { printf( "Reenter last line: " ); } input
{ $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input : error ´\n´
{ yyerrok;
printf( "Reenter last line: " ); }
input
{ $$ = $4; }
;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error
{ resynch();
yyerrok ;
yyclearin ; }
;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

## 6.9 THE YACC ENVIRONMENT

When the user inputs a specification to **yacc**, the output is a file of C programs, called *y.tab.c*. The function produced by **yacc** is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

You must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

You must supply these two routines in one form or another. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main* and *yyerror*. To access the library, supply a **-ly** argument to the loader. These default programs are trivial (though sufficient to the task), as can be seen by examining the source below.

```
main(){
return( yyparse() );
}
```

and

```
# include <stdio.h>
yyerror(s) char *s; {
fprintf( stderr, "%s\n", s );
}
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the **yacc** library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

## 6.10 HINTS FOR PREPARING SPECIFICATIONS

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

### 6.10.1 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file.

[1] Use all capital letters for token names, all lowercase letters for nonterminal names. If you follow this rule, debugging it will be easier, since you'll know who to blame when things go wrong.

[2] Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

[3] Put all rules with the same left-hand side together. Put the left-hand side in only once, and let all the following rules begin with a vertical bar.

[4] Put a semicolon only after the last rule with a given left-hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

[5] Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in the next section is written following this style, as are all of the examples seen so far (or at least all the examples where space permits). You'll have to make up your own mind about these stylistic questions. The first principle, as always, is to make the rules visible through the body of action code.

### 6.10.2 Left Recursion

The algorithm used by the **yacc** parser encourages so called "left recursive" grammar rules: rules of the form

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
| list ´,´ item
;
```

and

```
seq : item
| seq item
;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq : item
| item seq
;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */ | seq item
;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen, when it hasn't seen enough to know.

### 6.10.3 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
int dflag;
%}

%%

prog : decls stats
;

decls : /* empty */
{ dflag = 1; }
| decls declaration

stats : /* empty */
{ dflag = 0; }
| stats statement
;
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach, while arguably impure (and conducive to error), represents a way of doing some things that are difficult, if not impossible, to do otherwise.

### 6.10.4 Reserved Words

Some programming languages permit the programmer to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if is a keyword, and that instance is a variable." The programmer can make a stab at it, using the mechanism described in the previous subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, that is, be forbidden for use as variable names. This is always the preferred course, especially where style is concerned.

## 6.11 YACC INPUT SYNTAX

This section has a description of the **yacc** input syntax, as a **yacc** specification. Context dependencies, etc., are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decides whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
/* grammar for the input to yacc */

/* basic entities */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by colon */
%token NUMBER /* [0-9]+ */

/* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the %{ mark */
%token RCURL /* the %} mark */

/* ascii character literals stand for themselves */

%start spec

%%

spec : defs MARK rules tail
;

tail : MARK { In this action, eat up the rest of the file }
| /* empty: the second MARK is optional */
;

defs : /* empty */
```

```
| defs def
;

def : START IDENTIFIER
| UNION { Copy union definition to output }
| LCURL { Copy C code to output file } RCURL
| ndefs rword tag nlist
;

rword : TOKEN
| LEFT
| RIGHT
| NONASSOC
| TYPE
;

tag : /* empty: union tag is optional */
| ´ < ´ IDENTIFIER ´ > ´
;

nlist : nmno
| nlist nmno
| nlist ´,´ nmno
;

nmno : IDENTIFIER /* NOTE: literal illegal with %type */
| IDENTIFIER NUMBER /* NOTE: illegal with %type */
;

/* rules section */
rules : C_IDENTIFIER rbody prec
| rules rule
;

rule : C_IDENTIFIER rbody prec
| ´|´ rbody prec
;

rbody : /* empty */
| rbody IDENTIFIER
| rbody act
;

act : ´{´ { Copy action, translate $$, etc. } ´}´
;

prec : /* empty */
```

```
| PREC IDENTIFIER
| PREC IDENTIFIER act
| prec ´;´
;
```

## 6.12 EXAMPLES

In this section, we supply two related examples of **yacc** in action. One is fairly simple; the other is a little more sophisticated. The second builds upon the foundation laid by the first, so we urge you to examine both.

### 6.12.1 A Simple Example

This example gives the complete **yacc** specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; this job is probably better done by the lexical analyzer.

```
%{
#  include <stdio.h>
#  include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left ´|´
%left ´&´
%left ´+´  ´-´
%left ´*´  ´/´  ´%´
%left UMINUS /* supplies precedence for unary minus */

%% /* beginning of rules section */

list : /* empty */
| list stat ´\n´
| list error ´\n´
{ yyerrok; }
;

stat : expr
{printf( "%d\n" , $1 ); }
| LETTER ´ = ´ expr
{ regs[$1] = $3; }
;

expr : ´(´ expr ´)´
{ $$ = $2; }
| expr ´+´ expr
{ $$ = $1 + $3; }
| expr ´-´ expr
{ $$ = $1 - $3; }
| expr ´*´ expr
{ $$ = $1 * $3; }
| expr ´/´ expr
{ $$ = $1 / $3; }
| expr ´%´ expr
{ $$ = $1 % $3; }
| expr ´&´ expr
{ $$ = $1 & $3; }
| expr ´|´ expr
```

```
{ $$ = $1 | $3; }
| ´-´ expr %prec UMINUS
{ $$ = _ $2; }
| LETTER
{ $$ = regs[$1]; }
| number
;

number : DIGIT
{ $$ = $1; base = ($1==0) ? 8 : 10; }
| number DIGIT
{ $$ = base * $1 + $2; }
;

%% /* start of programs */

yylex() { /* lexical analysis routine */
/* returns LETTER for a lower case letter, yylval = 0 through 25 */
/* return DIGIT for a digit, yylval = 0 through 9 */
/* all other characters are returned immediately */

int c;

while( (c=getchar()) == ´ ´ ) { /* skip blanks */ }

/* c is now nonblank */

if( islower( c ) ) {
yylval = c - ´ a´;
return ( LETTER );
}
if( isdigit( c ) ) {
yylval = c - ´0´;
return( DIGIT );
}
return( c );
}
```

### 6.12.2 An Advanced Example

This section gives an example of a grammar using some of the more advanced features of **yacc**. The desk calculator of the previous example is here modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

( x , y )

where $x$ is less than or equal to $y$. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in the previous section; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *doubles*. This structure is given a type name, INTERVAL, by using *typedef*. The **yacc** value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + ( 3.5 - 4. )

and

2.5 + ( 3.5 , 4. )

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts

even more so. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

#  include <stdio.h>
#  include <ctype.h>

typedef struct interval {
double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start lines

%union {
int ival;
double dval;
INTERVAL wal;
}

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST /* floating point constant */

%type <dval> dexp /* expression */

%type <wal> vexp /* interval expression */

/* precedence information about the operators */

%left ´+´ ´-´
%left ´*´ ´/´
```

```
%left UMINUS /* precedence for unary minus */

%%

lines : /* empty */
| lines line
;

line :dexp ´\n´
{ printf( "%15.8f\n", $1 ); }
| vexp ´\n´
{ printf( "(%15.8f , %15.8f )\n", $1.lo, $l.hi ); }
| DREG ´=´ dexp ´\n´
{ dreg[$1] = $3; }
| VREG ´=´ vexp ´\n´
{ vreg[$1] = $3; }
| error ´\n´
{ yyerrok; }
;

dexp : CONST | DREG
{ $$ = dreg[$l]; } {
| dexp ´+´ dexp
{ $$ = $1 + $3; }
| dexp ´-´ dexp
{ $$ = $1 _ $3; }
| dexp ´*´ dexp
{ $$ = $1 * $3; }
| dexp ´/´ dexp
{ $$ = $1 / $3; }
| ´-´ dexp %prec UMINUS
{ $$ = - $2; }
| ´(´ dexp ´)´
{ $$ = $2; }
;

vexp : dexp
{ $$.hi = $$.lo= $1; }
| ´(´ dexp ´,´ dexp ´)´
{
$$.lo = $2;
$$.hi = $4;
if( $$.lo > $$.hi ){
printf( "interval out of order\n" );
YYERROR;
}
}
```

```
| VREG
{ $$ = vreg[$1]; }
| vexp ´+´ vexp
{ $$.hi = $l.hi + $3.hi;
$$.lo = $1.lo + $3.lo; }
| dexp ´+´ vexp
{ $$.hi = $1 + $3.hi;
$$.lo = $1 + $3.lo; }
| vexp ´-´ vexp
{ $$.hi = $1.hi - $3.lo;
$$.lo = $1.lo - $3.hi; }
| dexp ´-´ vexp
{ $$.hi = $1 - $3.lo;
$$.lo = $1 - $3.hi; }
| vexp ´*´ vexp
{ $$ = vmul( $l.lo, $1.hi, $3 ); }
| dexp ´*´ vexp
{ $$ = vmul( $1, $1, $3 ); }
| vexp ´/´ vexp
{ if( dcheck( $3 ) ) YYERROR;
$$ = vdiv( $1.lo, $1.hi, $3 ); }
| dexp ´/´ vexp
{ if( dcheck( $3 ) ) YYERROR;
$$ = vdiv( $1, $1, $3 ); }
| ´-´ vexp %precUMINUS
{ $$.hi = -$2.lo; $$.lo = -$2.hi; }
| ´(´ vexp ´)´
{ $$ = $2; }
;

%%

# define BSZ 50 /* buffer size for floating point numbers */

/* lexical analysis */
yylex(){
register c;

while( (c=getchar()) == ´ ´ ){ /* skip over blanks */ }

if( isupper( c ) ){
yylval.ival = c - ´A´;
return( VREG );
}
if( islower( c ) ){
yylval.ival = c - ´a´;
return( DREG );
```

```
}

if( isdigit(c) || c==´.´ ){
/* gobble up digits, points, exponents */
char buf[BSZ+1], *cp = buf; int dot = 0, exp = 0;
for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

*cp = c;
if( isdigit( c ) ) continue;
if( c == ´.´ ){
if( dot++ || exp ) return( ´.´ ); /* will cause syntax error */
continue;
}
if( c==´e´ ){
if( exp++ ) return( ´e´ ); /* will cause syntax error */
continue;
}

/* end of number */
break;
}
*cp = ´\0´;
if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
else ungetc( c, stdin ); /* push back last char read */
yylval.dval = atof( buf );
return( CONST );
}
return( c );
}

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
/* returns the smallest interval containing a, b, c, and d */
/* used by *, / routines */
INTERVAL v;

if( a>b ) { v.hi = a; v.lo = b; }
else { v.hi = b; v.lo = a; }

if( c>d ) {
if( c>v.hi ) v.hi = c;
if( d<v.lo ) v.lo = d;
} else {
if( d>v.hi ) v.hi = d;
```

```
if( c<v.lo ) v.lo = c;
}
return( v ); }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
}

dcheck( v ) INTERVAL v; {
if( v.hi >= 0. && v.lo <= 0. ){
printf( "divisor interval contains 0.\n" );
return( 1 );
}
return( 0 );
}

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
}
```

## 6.13 OLD FEATURES SUPPORTED BUT NOT ENCOURAGED

This section mentions synonyms and features that are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.
3. The use of multi-character literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job which must be actually done by the lexical analyzer.
4. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc..
5. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

6. Actions may also have the form

={...}

and the curly braces can be dropped if the action is a single C statement.

7. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

# Chapter 7: The Source Code Control System

## 7.1 INTRODUCTION

In this chapter, we give an overview of the Source Code Control System (SCCS). The material here is for programmers who are concerned with getting their task done rather than with how SCCS works. Those who need more detailed information will find it under the heading "Further Information" at the end of this chapter.

SCCS is a source management system. It is actually a collection of programs that, used together, help you maintain a record of versions of a program. A record kept with each set of changes details what the changes are, why and when they were made, and who made them. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will ensure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file." There are three major operations that can be performed on the s-file:

1. Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file should not be edited or changed in any way. If you make any changes made to a file retrieved in this way, they will probably be lost.
2. Get a file for editing. This operation retrieves from the s-file, a version of the file that may be edited, then incorporated back into the s-file. Only one person may be editing a file at one time.
3. Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

## 7.2 TERMINOLOGY

There are a number of terms that are worth learning before we go any further.

*S-file* The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format. That is to say, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for

each version, including the comments given by the person who created the version explaining why the changes were made.

Deltas — Each set of changes to the s-file which is approximately equivalent to a version of the file — is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before. This matches normal usage, where the previous changes are not saved at all; so all changes are automatically based on all other changes that have happened through history. It is possible to get a version of the file that has selected deltas removed from the middle of the list of changes. This is equivalent to removing your changes later.

SID — A SID (SCCS ID) is a number that represents a delta. This is normally a two-part number consisting of a *release* number and a *level* number. Normally, the release number stays the same; however, it is possible to move into a new release if some major change is being made. Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

ID keywords — When you get a version of a file with intent to compile and install it (**not** edit it), some special keywords are expanded inline by SCCS. These *ID Keywords* can be used to include the current version number or other information in the file. All ID keywords are of the form **%***x***%**, where *x* is an upper case letter. For example, **%I%** is the SID of the latest delta applied, **%W%** includes the module name, SID, and a mark that makes it findable by a program, and **%G%** is the date of the latest delta applied.

When you retrieve a file for editing, the ID keywords are not expanded. After you put them back in to the s-file, they will be expanded automatically on each new version. (If keywords were to be expanded accidentally, then a file would always appear to be the same version.) If you install a version of the program without expanding the ID keywords, it will be impossible to tell what version it is,

## 7.3 CREATING SCCS FILES

To put source files into SCCS format, run the following C Shell script.

```
#! /bin/csh
mkdir save
foreach i (*.[ch])
        admin -i$i s.$i
        mv $i save/$i
end
```

This will create s-files in the current directory. The files will be removed from the current directory and hidden away in the directory *save*. To get all the files out of this directory, use the procedure described below. When you are convinced that SCCS has created the proper s-files, you should remove the directory *save*.

If you want to have ID keywords in the files, it is best to put them in before you create the s-files. Otherwise, **admin**[1] will display the "No ID Keywords (cm7)", warning message.

### 7.4 Getting Files for Compilation

To get a copy of the latest version of the file prog.c, use the line:

**get s.prog.c**

Get will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved, and that it has 87 lines.

**Note:** *Get* obtains the version number from the SID of the final delta applied.

The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

**Caution:** This copy of the file **should not be changed**, since **delta** is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a **get**.

### 7.5 Changing Files (Creating Deltas)

In this section, we explain how to retrieve and change (or *delta*) a file.

#### 7.5.1 Getting a Copy to Edit

To edit a source file, you must first get it, requesting permission to edit it, as shown below.

**get -e prog.c**

The response will be the same as with **get** except that it will also say:

New delta 1.2

You then edit it, using any text editor.

### 7.5.2 Merging the Changes Into the s-file

When you have completed the desired changes, you can merge them into the SCCS file using the **delta** command:

**delta s.prog.c**

Delta will prompt you for "comments?" before it merges the changes in. At this prompt, you may type a one-line comment. To enter a multi-line comment, end each line but the last with a concealed newline (a backslash \). *Delta* will then display a message like:

1.2
5 inserted
3 deleted
84 unchanged

meaning that

- delta 1.2 was created
- it inserted five lines
- it removed three lines
- it left 84 lines unchanged. (Changes to a line are counted as a line deleted and a line inserted.)

The *prog.c* file will be removed; it can be retrieved using **get**.

### 7.5.3 When to Make Deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3." However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get the files, and recompile everything.

### 7.5.4 The sact Command

To find out what files where being edited, you can use:

**sact .**

to print out all the files being edited in the current directory. The output for each named file consists of five fields separated by spaces. These fields specify the following information:

Field 1 SID of an existing delta in the SCCS file to which changes will be made to create the new delta.

Field 2 SID for the new delta to be created.

Field 3 Logname of the user who will make the delta (i.e., the person who executed a get for editing).

Field 4 Date that **get -e** was executed.

Field 5 Time that **get -e** was executed.

**Note:** The command:

**sacct s.prog.c**

prints a report for file *prog.c* only.

### 7.5.5 ID Keywords

ID keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c          1.208/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string @(#) is a special string that signals the beginning of an SCCS ID keyword.

### 7.5.6 The what Command

To find out what version of a program is being run, use:

**what prog.c /usr/bin/prog**

which will print all strings it finds that begin with @(#). This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c                          1.208/29/80
/usr/bin/prog:
        prog.c                          1.102/05/79
```

This means that the source in *prog.c* will not compile into the same version as the binary in /usr/bin/prog.

### 7.5.7 Where to Put ID Keywords

ID keywords can be inserted anywhere, including in comments, but ID Keywords that are compiled into the object module are especially useful, since it lets you get version information from object as well as the source code. This practice requires additional address space for the object module, something that will not normally present a problem on virtual memory machines.

When you put ID Keywords into header files, it is important that you assign them to different variables. For example, you might use:

static char AccessSid[] = "%W% %G%";

in the file *access.h* and:

static char OpsysSid[] = "%W% %G%";

in the file *opsys.h*. Otherwise, you will get compilation errors because *SccsId* is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put ID keywords in header files in comments.

### 7.5.8 Keeping SID's Consistent Across Files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always edit all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) given new deltas. This can be done fairly easily by

**get -e s.***

which will create editable copies of all files in the current directory. To make the delta, use:

**delta s.***

You will be prompted for comments only once.

### 7.5.9 Creating a New Release

When you want to create a new release of a program, you can specify the release number you want to create on the **get** command line. For example,

**get -e -r2 s.prog.c**

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

**get -e -r2 s.***

## 7.6 RESTORING OLD VERSIONS

### 7.6.1 Reverting to Old Versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3, which introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

**get -r1.2 s.prog.c**

This will produce a version of *prog.c* as it was at delta 1.2. This version can then be reinstalled.

In some cases you don't know the SID of the delta you want. However, you can revert to the version of the program that was running as of a certain date by using the **-c** (cutoff) flag. For example,

**get -C850122120000 s.prog.c**

will retrieve whatever version was current as of January 22, 1985 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

**get -c"85/01/22 12:00:00" s.prog.c**

### 7.6.2 Selectively Deleting Old Deltas

Suppose that you later decided that the delta you made at 1.3 was incorrect. You could remove it by

**rmdel -x1.3 s.prog.c**

**Rmdel** removes the latest delta from each named SCCS file. The delta to be removed must be the newest (most recent) one in its branch in the delta chain. The SID specified must not be that of a version being edited for the purpose of making a delta. Thus, if a *p-file* exists for the named SCCS file, the SID specified must not appear in any entry of the *p-file*.

If you give a directory as an argument, rmdel behaves as though each file in the directory were specified as a named file, except that it silently ignores any non-SCCS or otherwise unreadable files. If you supply a dash (-) in place of a filename, the standard input is read. Each line of the standard input is taken to be the name of an SCCS file to be processed. Here, too, non-SCCS files and unreadable files are silently ignored.

Permission to remove a delta is granted to the person who created it, as well as to the owner of the associated file and directory.

## 7.7 AUDITING CHANGES

### 7.7.1 The prs Command

When you created a delta, you presumably gave a reason for the delta to the "comments?" prompt. To print out these comments later, use:

**prs s.prog.c**

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
% prs //ice/kate/s.foo.h
//ice/kate/s.foo.h:

D 2.2 85/05/16 12:53:45 kate 4 3          00001/00000/00002
MRs:
COMMENTS:
third time is a charmer

D 2.1 85/05/16 12:52:58 kate 3 2          00001/00000/00001
MRs:
COMMENTS:
new test

D 1.2 85/05/16 12:48:12 kate 2 1          00000/00000/00001
MRs:
COMMENTS:
testint

D 1.1 85/05/16 12:41:58 kate 1 0          00001/00000/00000
MRs:
COMMENTS:
date and time created 85/05/16 12:41:58 by kate
```

### 7.7.2 Finding Why Lines Were Inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
get -m s.prog.c
```

You can then find out what this delta did by printing the comments using **prs**.

To find out what lines are associated with a particular delta (e.g., 1.3), use:

```
get -m -p s.prog.c | grep ´^1.3´
```

The **-p** flag causes **get** to output the generated source to the standard output rather than to a file.

### 7.7.3 Finding What Changes You Have Made

To compare two versions that are in deltas, use:

```
sccsdiff -r1.3 -r1.6 s.prog.c
```

to see the differences between delta 1.3 and delta 1.6.

### 7.7.4 Unget

If you inadvertently edit the wrong file, you can back out by using the **unget** command.

**unget s.prog.c**

## 7.8 USING SCCS ON A PROJECT

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a typical session might include the following command lines and messages. (We show a C Shell prompt, although SCCS will run in any shell, under any version of DOMAIN/IX)

% **get -e s.a.c s.g.c s.t.c**
% **vi a.c g.c t.c**
*...do testing of the (experimental) version*
% **delta s.a.c s.g.c s.t.c**
% **get s.a.c s.g.c s.t.c**
% **sact**
No outstanding deltas for s.a.c
No outstanding deltas for s.g.c
No outstanding deltas for s.t.c
% **make install**

As a general rule, all source files should be given new deltas before installing the program for general use. This will ensure that it is possible to restore any version in use at any time.

## 7.9 ERROR RECOVERY

### 7.9.1 Recovering a Damaged Edit File

Sometimes you may find that you have destroyed or damaged a file that you were trying to edit. Unfortunately, you can't just remove and re-edit it. SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using since that would expand the ID Keywords. Instead, you must say:

**get -k s.prog.c**

This will not expand the ID Keywords, so it is safe to do a delta with it.

### 7.9.2 Restoring the s-file

Occasionally, the SCCS file itself may get damaged (usually edited). Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

**admin -z s.prog.c**

## 7.10 USING THE admin[1] COMMAND

There are a number of parameters that can be set using the **admin**[1] command. The most interesting of these are flags. Flags can be added by using the **-f** flag. For example:

**admin —fd1 s.prog.c**

sets the *d* flag to the value 1. This flag can be deleted by using:

**admin -dd s.prog.c**

The most useful flags are:

**b** Allow branches to be made using the -b flag to **get**.

**d***S* *S* is the default SID to be used on a **get**. If this is just a release number, it constrains the version to a particular release only.

**i** Gives a fatal error if there are no ID Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the ID Keywords inserted as constants instead of internal forms.

**y** The *type* of the module. Actually, the value of this flag is unused by SCCS except that it replaces the **%Y%** keyword.

The **-t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **-t** flag ensures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use **prs -t**.

The **admin** command can be used safely any number of times on files. A file need not be gotten for **admin** to work.

## 7.11 MAINTAINING DIFFERENT VERSIONS (BRANCHES)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a *branch*. Normally, deltas continue in a straight line, each depending on the delta before. Creating a branch *forks off* a version of the program.

The ability to create branches must be enabled in advance using:

**admin -fb s.prog.c**

The **-fb** flag can be specified when the SCCS file is first created.

### 7.11.1 Creating a Branch

To create a branch, use:

**get -e -b s.prog.c**

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

### 7.11.2 Getting from a Branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

**get -r1.5.1 s.prog.c**

### 7.11.3 Merging a Branch Back into the Main Trunk

At some point, you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime, someone may have created a delta 1.6 that you do not want to lose. The commands:

**get -e -i1.5.1.1-1.5.1 s.prog.c**
**delta s.prog.c**

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

**Note:** Branches should be kept to a minimum. After the first branch from the trunk, SID'S are assigned rather haphazardly, and the structure gets complex fast.

## 7.12 USING SCCS WITH MAKE

SCCS and **make** can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few entries that every makefile should include. These are:

| | |
|---|---|
| a.out | (or other makefile output) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called *all* and should in turn have dependencies on everything the makefile can generate. |
| install | Moves the objects to the final resting place, doing any special **chmod**'s or **ranlib**'s that are required. |
| sources | Creates all the source files from SCCS files. |
| clean | Removes all unwanted objects from the directory. |
| print | Prints the contents of the directory. |

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed.

### 7.12.1 To Maintain Single Programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= -i -s
prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h
example: example.o
	$(CC) $(LDFLAGS) -o example example.o
example.o: example.c
.DEFAULT:
	get s.$<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS= s.prog.c s.prog.h s.example.c

LDFLAGS= -i -s

prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h
example: example.o
	$(CC) $(LDFLAGS) -o example example.o
sources: $(SRCS)
$(SRCS):
	get s.$@
```

There are a couple of advantages to this approach.

1. The explicit dependencies of the .o on the .c files are not needed.
2. There is an entry called "sources" so if you want to get all the sources you can just say

   **make** *sources*

3. The makefile is less likely to do confusing things since it won't try to **get** things that do not exist.

### 7.12.2 To Maintain a Library

Libraries that are largely static are best updated using explicit commands, since **make** doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
#  configuration information
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.c d.s x.h y.h z.h
TARG= /usr/lib

#  programs
GET= get
REL=
AR= -ar
RANLIB= ranlib

lib.a: $(OBJS)
	$(AR) rvu lib.a $(OBJS)
	$(RANLIB) lib.a

install: lib.a
	cp lib.a $(TARG)/lib.a
	$(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
	$(GET) $(REL) s.$@

print: sources
	pr *.h *.[cs]
clean:
	rm -f *.o
	rm -f core a.out $(LIB)
```

The $(*REL*) in the get can be used to get old versions easily; for example:

**make b.o REL=-r1.3**

### 7.12.3 To Maintain a Large Program

```
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.y d.s x.h y.h z.h

GET= get
REL=

a.out: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
	$(GET) $(REL) $@
```

It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since **make** does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
	touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the **touch** command can be removed completely; the equivalent effect will be achieved by doing an automatic **get** on z.h.

## 7.13 SUMMARY OF COMMANDS AND KEYWORDS

### 7.13.1 Commands

**get** Gets files for compilation (not for editing). ID Keywords are expanded.

- **-r***SID* Version to get.
- **-p** Send to standard output rather than to the actual file.
- **-k** Don't expand ID Keywords.
- **-i***list* List of deltas to include.
- **-x***list* **List of deltas to exclude.**
- **-m** Precede each line with SID of creating delta.
- **-c***date* Don't apply any deltas created after *date*.
- **-e** Gets files for editing.
- **-b** Create a branch.
- **-i***list* Same as **get**.
- **-x***list* Same as **get**.

**delta** Merge a file gotten using **get -e** back into the s-file. Collect comments about why this delta was made.

**unget** Remove a file that has been edited previously without merging the changes into the s-file.

**prs** Produce a report of changes.

**what** Find and print ID Keywords.

**admin** Create or set parameters on s-files.

**-i***file* Create, using *file* as the initial contents.

**-z** Rebuild the checksum in case the file has been damaged.

**-t***file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for ensuring that documentation gets distributed with the s-file.

**-f***flag* Turn on the *flag*.

**-d***flag* Turn off (delete) the *flag*.

flags

**b** Allow branches to be made using the -b flag to edit.

**d***S* *S* is the default SID to be used on a **get**.

**i** Cause *No ID Keywords* error message to be a fatal error rather than a warning.

**t** The module *type*; the value of this flag replaces the **%Y%** keyword.

### 7.13.2 ID Keywords

%Z% Expands to @(#) for the *what* command to find.

%M% The current module name, e.g., *prog.c*.

%l% The highest SID applied.

%W% A shorthand for *%Z%%M%* < *tab*> *%I%*.

%G% The date of the delta corresponding to the *%I%* keyword.

%K% The current release number, i.e., the first component of the *%I%* keyword.

%Y% Replaced by the value of the **t** flag (set by **admin**.)

# Index

# APPENDICES

# Appendix A: The C Language—Reference Manual†

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

This manual describes the C language on the DEC PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, such dependencies follow directly from the properties of the hardware; the various compilers are generally quite compatible.

## 2. LEXICAL CONVENTIONS

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

### 2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

### 2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

| | |
|---|---|
| DEC PDP-11 | 7 characters, 2 cases |
| Honeywell 6000 | 6 characters, 1 case |
| IBM 360/370 | 7 characters, 1 case |
| Interdata 8/32 | 8 characters, 2 cases |

---

† This manual is reprinted, with minor changes, from *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1978.

### 2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

| | | | |
|---|---|---|---|
| **int** | **short** | **goto** | **for** |
| **char** | **unsigned** | **return** | **do** |
| **float** | **auto** | **sizeof** | **while** |
| **double** | **extern** | **break** | **switch** |
| **struct** | **register** | **continue** | **case** |
| **union** | **typedef** | **if** | **default** |
| **long** | **static** | **else** | **entry** |

The **entry** keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words **fortran** and **asm**.

### 2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics that affect sizes are summarized in §2.6.

#### 2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero), decimal otherwise. The digits **8** and **9** have octal value 10 and 11 respectively. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**.

#### 2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on some machines integer and long values may be considered identical.

#### 2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote ‘ and the backslash \, may be represented according to the following table of escape sequences:

| | | |
|---|---|---|
| new-line | NL (LF) | **n** |
| horizontal tab | HT | **t** |
| backspace | BS | **b** |
| carriage return | CR | **r** |
| form feed | FF | **f** |
| backslash | \ | \\ |
| single quote | ' | \' |
| bit pattern | *ddd* | *ddd* |

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

### 2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

## 2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in **". . ."**. A string has type "array of characters" and storage class **static** (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte at the end of each string so that programs which scan the string can find its end. In a string, the double quote character **"** must be preceded by a **;** in addition, the same escapes as described for character constants may be used. Finally, a \ and the immediately following new-line are ignored.

## 2.6 Hardware characteristics

The following table summarizes certain hardware properties that vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought a priori.

| | DEC PDP-11 ASCII | Honeywell 6000 ASCII | IBM 370 EBCDIC | Interdata 8/32 ASCII |
|---|---|---|---|---|
| **char** | 8 bits | 9 bits | 8 bits | 8 bits |
| **int** | 16 | 36 | 32 | 32 |
| **short** | 16 | 36 | 16 | 16 |
| **long** | 32 | 36 | 32 | 32 |
| **float** | 32 | 36 | 32 | 32 |
| **double** | 64 | 72 | 64 | 64 |
| range | $\pm 10^{\pm 38}$ | $\pm 10^{\pm 38}$ | $\pm 10^{\pm 76}$ | $\pm 10^{\pm 76}$ |

## 3. SYNTAX NOTATION

In the syntax notation used in this manual, syntactic categories are indicated by italic type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

$$\{\ \textit{expression}_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

## 4. WHAT'S IN A NAME?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo $2^n$ where $n$ is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double-precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will collectively be called *integral* types, **float** and **double** will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

*arrays* of objects of most types;

*functions* which return objects of a given type;
*pointers* to objects of a given type;
*structures* containing a sequence of objects of various types;
*unions* capable of containing any one of several objects of various types.
In general these methods of constructing objects can be applied recursively.

## 5. OBJECTS AND LVALUES

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

## 6. CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

### 6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from -128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, **'\377'** has the value **-1**.

When a longer integer is converted to a shorter or to a **char**, it is truncated on the left; excess bits are simply discarded.

### 6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a **float** appears in an expression it is lengthened to **double** by zero-padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length.

### 6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

### 6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

### 6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{\text{wordsize}}$). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

### 6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.
Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.
Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
Otherwise, both operands must be **int**, and that is the type of the result.

## 7. EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. All existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

### 7.1 Primary expressions

Primary expressions involving **.**, **->**, subscripting, and function calls group left to right.

*primary-expression:*
*identifier*
*constant*
*string*
**(** *expression* **)**
*primary-expression* **[** *expression* **]**
*primary-expression* **(** *expression-list*$_{opt}$ **)**
*primary-lvalue* **.** *identifier*
*primary-expression* **->** *identifier*

*expression-list:*
*expression*
*expression-list* **,** *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of . . .", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to . . .". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning . . .", when used except in the function-name position of a call, is converted to "pointer to function returning . . .".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int**; floating constants are **double**.

A string is a primary expression. Its type is originally "array of **char**"; but following the same rule given above for identifiers, this is modified to "pointer to char" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to . . .", the subscript expression is **int**, and the type of the result is " . . . ". The expression **E1 [E2]** is identical (by definition) to ***( (E1)+(E2) )**. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning . . .", and the result of the function call is of type " . . . ". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call; any of type **char** or **short** are converted to **int**; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a **-** and a >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression **E1->MOS** is the same as **( *E1 ) .MOS**. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

### 7.2 Unary operators

Expressions with unary operators group right-to-left.

*unary- expression:*
- ***** *expression*
- **&** *lvalue*
- **-** *expression*
- **!** *expression*
- **~** *expression*
- ++ *lvalue*
- **--** *lvalue*
- *lvalue* ++
- *lvalue* **--**
- **(** *type-name* **)** *expression*
- **sizeof** *expression*
- **sizeof (** *type-name* **)**

The unary ***** operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to . . .", the type of the result is " . . . ".

The result of the unary **&** operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is " . . . ", the type of the result is "pointer to . . .".

The result of the unary **-** operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from $2^n$, where $n$ is the number of bits in an **int**. There is no unary + operator.

The result of the logical negation operator **!** is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The **~** operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression **++x** is equivalent to **x+=1**. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix **--** is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix **--** is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix **--** operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

The **sizeof** operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***) - 2** is the same as **(sizeof(***type***)) - 2**.

### 7.3 Multiplicative operators

The multiplicative operators *****, /, and **%** group left-to-right. The usual arithmetic conversions are performed.

*multiplicative-expression:*
*expression * expression*
*expression / expression*
*expression % expression*

The binary ***** operator indicates multiplication. The ***** operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b) *b + a%b** is equal to **a** (if **b** is not 0).

The binary **%** operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be **float**.

### 7.4 Additive operators

The additive operators + and **-** group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

*additive-expression:*
*expression* + *expression*
*expression* **-** *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the **-** operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

### 7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

*shift-expression:*
*expression* << *expression*
*expression* >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits; vacated bits are 0-filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit

positions. The right shift is guaranteed to be logical (0-fill) if **E1** is **unsigned**; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

### 7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; **a<b<c** does not mean what it seems to.

*relational-expression:*
*expression* < *expression*
*expression* > *expression*
*expression* <= *expression*
*expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

### 7.7 Equality operators

*equality-expression:*
*expression* == *expression*
*expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

### 7.8 Bitwise AND operator

*and-expression:*
*expression* & *expression*

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

### 7.9 Bitwise exclusive OR operator

*exclusive-or-expression:*
*expression* ^ *expression*

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

### 7.10 Bitwise inclusive OR operator

*inclusive-or-expression:*
*expression* | *expression*

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

### 7.11 Logical AND operator

*logical-and-expression:*
*expression* **&&** *expression*

The **&&** operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike **&**, **&&** guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

### 7.12 Logical OR operator

*logical-or-expression:*
*expression* || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

### 7.13 Conditional operator

*conditional-expression:*
*expression* **?** *expression* **:** *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

### 7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

*assignment-expression:*
*lvalue* = *expression*
*lvalue* += *expression*
*lvalue* **-**= *expression*
*lvalue* ***=** *expression*
*lvalue* /= *expression*
*lvalue* %= *expression*
*lvalue* <<= *expression*
*lvalue* >>= *expression*
*lvalue* &= *expression*
*lvalue* ^= *expression*
*lvalue* |= *expression*

In the simple assignment with =j the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form **E1** *op*= **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* **(E2)**; however, **E1** is evaluated only once. In += and **-**=, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

### 7.15 Comma operator

*comma-expression:*
*expression* **,** *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

**f ( a , (t=3 , t+2) , c)**

has three arguments, the second of which has the value 5.

## 8. DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

*declaration:*
*decl-specifiers declarator-list*$_{opt}$ **;**

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

*decl-specifiers:*
*type-specifier decl-specifiers*$_{opt}$
*sc-specifier decl-specifiers*$_{opt}$

The list must be self-consistent in a way described below.

**8. Storage class specifiers**
The sc-specifiers are:

*sc-specifier:*
**auto**
**static**
**extern**
**register**
**typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an auto declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int**, **char**, or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

**8.2 Type specifiers**
The type-specifiers are

*type-specifier:*
**char**
**short**
**int**
**long**
**unsigned**
**float**
**double**
*struct-or-union-specifier*
*typedef-name*

The words **long**, **short**, and **unsigned** may be thought of as adjectives; the following combinations are acceptable.

**short int**
**long int**
**unsigned int**
**long float**

The meaning of the last is the same as **double**. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures and unions are discussed in §8.5; declarations with **typedef** names are discussed in §8.8.

### 8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

*declarator-list:*
*init-declarator*
*init-declarator* **,** *declarator-list*

*init-declarator:*
*declarator initializer*$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

*declarator:*
*identifier*
**(** *declarator* **)**
***** *declarator*
*declarator* **(** **)**
*declarator* **[** *constant-expression*$_{opt}$ **]**

The grouping is the same as in expressions.

### 8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

**T D1**

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type " . . . **T**," where the " . . . " is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is " . . . pointer to **T**."

If **D1** has the form

**D()**

then the contained identifier has the type " . . . function returning **T**."

If **D1** has the form

**D[***constant-expression***]**

or

**D[ ]**

then the contained identifier has type " . . . array of **T**." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is **int**. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures, unions or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

**int i, *ip, f(), *fip(), (*pfi)();**

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer. As another example,

**float fa[17], *afp[17];**

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

**static int x3d[3][5][7];**

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array," the last has type **int**.

### 8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

*struct-or-union-specifier:*
*struct-or-union* **{** *struct-decl-list* **}**
*struct-or-union identifier* **{** *struct-decl-list* **}**
*struct-or-union identifier*

*struct-or-union:*
**struct**
**union**

The struct-decl-list is a sequence of declarations for the members of the structure or union:

*struct-decl-list:*
*struct-declaration*
*struct-declaration struct-decl-list*

*struct-declaration:*
*type-specifier struct-declarator-list* **;**

*struct-declarator-list:*
*struct-declarator*
*struct-declarator* **,** *struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

*struct-declarator:*
*declarator*
*declarator* **:** *constant-expression*
**:** *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on the PDP-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values. In all implementations, there are no arrays of fields, and the address-of operator & may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

**struct** *identifier* **{** *struct-decl-list* **}**
**union** *identifier* **{** *struct-decl-list* **}**

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

**struct** *identifier*
**union** *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

**sp->count**

refers to the **count** field of the structure to which **sp** points;

**s.left**

refers to the left subtree pointer of the structure **s**; and

**s.right->tword [0]**

refers to the first character of the **tword** member of the right subtree of **s**.

## 8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and consists of an expression or a list of values nested in braces.

*initializer:*
= *expression*
= **{** *initializer-list* **}**
= **{** *initializer-list* **,** **}**

*initializer-list:*
*expression*
*initializer-list* **,** *initializer-list*
**{** *initializer-list* **}**

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] ={ 1, 3, 5 };
```

declares and initializes **x** as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float  y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely the same effect could have been achieved by

```
float  y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace, but that for **y[0]** does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float  y[4][3] = {
        { 1 },{ 2 },{ 3 },{ 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0. Finally,

```
char msg[] = "Syntax error on line %s0;
```

shows a character array whose members are initialized with a string.

## 8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of **sizeof**) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

*type-name:*
    *type-specifier abstract-declarator*

*abstract-declarator:*
    *empty*
    ( *abstract-declarator* )
    * *abstract-declarator*
    *abstract-declarator* ( )
    *abstract-declarator* [ $constant\text{-}expression_{opt}$ ]

To avoid ambiguity, in the construction

( *abstract-declarator* )

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of 3 integers," "function returning pointer to integer," and "pointer to function returning an integer."

### 8.8 Typedef

Declarations whose "storage class" is typedef do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

*typedef-name:*
*identifier*

Within the scope of a declaration involving typedef, each identifier appearing as part of any declarator therein become syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
comp lex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**," and that of **z** is the specified structure, **zp** is a pointer to such a structure.

**typedef** does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

## 9. STATEMENTS

Except as indicated, statements are executed in sequence.

### 9.1 Expression statement

Most statements are expression statements, which have the form

*expression* **;**

Usually expression statements are assignments or function calls.

### 9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

*compound-statement:*
**{** *declaration-list*$_{opt}$ *statement-list*$_{opt}$ **}**

*declaration-list:*
*declaration*
*declaration declaration-list*

*statement-list:*
*statement*
*statement statement-list*

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

### 9.3 Conditional statement

The two forms of the conditional statement are

**if** ( *expression* ) *statement*
**if** ( *expression* ) *statement* **else** *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

### 9.4 While statement

The **while** statement has the form

**while** ( *expression* ) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

### 9.5 Do statement

The **do** statement has the form

**do** *statement* **while** ( *expression* ) **;**

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

### 9.6 For statement

The **for** statement has the form

**for** ( *expression-1* **;** *expression-2*$_{opt}$ **;** *expression-3*$_{opt}$ ) *statement*

This statement is equivalent to

*expression-1* **;**
**while** ( *expression-2* ) **{**
*statement*
*expression-3* **;**
**}**

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

### 9.7 Switch statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

**switch** ( *expression* ) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be int. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

**case** *constant-expression* **:**

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

**default :**

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a **default** prefix, control passes to the prefixed statement. If no case matches and if there is no **default** then none of the statements in the switch is executed.

**case** and **default** prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see

**break**, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

### 9.8 Break statement

The statement

**break ;**

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

### 9.9 Continue statement

The statement

**continue ;**

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {     do {              for (...) {
   ...               ...               ...
contin: ;         contin: ;         contin: ;
}                 } while (...);    }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, §9.13.)

### 9.10 Return statement

A function returns to its caller by means of the **return** statement, which has one of the forms

**return ;**
**return** *expression* **;**

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

### 9.11 Goto statement

Control may be transferred unconditionally by means of the statement

**goto** *identifier* **;**

The identifier must be a label (§9.12) located in the current function.

### 9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

*identifier* **:**

which serve to declare the identifier as a label. The only use of a label is as a target of a goto. The scope of a label is the current function, excluding any

sub-blocks in which the same identifier has been redeclared. See §11.

### 9.13 Null statement

The null statement has the form

**;**

A null statement is useful to carry a label just before the **}** of a compound statement or to supply a null body to a looping statement such as **while**.

## 10. EXTERNAL DEFINITIONS

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

### 10.1 External function definitions

Function definitions have the form

*function-definition:*
*decl-specifiers*$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

*function-declarator:*
*declarator* **(** *parameter-list*$_{opt}$ **)**

*parameter-list:*
*identifier*
*identifier* **,** *parameter-list*

The function-body has the form

*function-body:*
*declaration-list compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
        int  m;

        m = (a > b) ? a : b ;
        return((m>c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c ;** is the declaration-list for the formal parameters; **{ . . . }** is the block giving the code for the statement.

C converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of . . . " are adjusted to read "pointer to . . . ". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

### 10.2 External data definitions

An external data definition has the form

*data-definition:*
    *declaration*

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

## 11. SCOPE RULES

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

### 11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they

appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers, **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
        auto int distance;
        ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance***.

### 11.2 Scope of externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the **extern** keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the **extern** specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the **extern** in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

## 12. COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

### 12.1 Token replacement

A compiler-control line of the form

#**define** *identifier token-string*

* It is agreed that the ice is thin here.

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

#**define** *identifier*( *identifier* , **...** , *identifier* ) *token-string*

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a ) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#**undef** *identifier*

causes the identifier's preprocessor definition to be forgotten.

### 12.2 File inclusion

A compiler control line of the form

#**include** "*filename*"

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

#**include** <*filename*>

searches only the standard places, and not the directory of the source file.

#**include**'s may be nested.

### 12.3 Conditional compilation

A compiler control line of the form

#**if** *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

#**ifdef** *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a #**define** control line. A control line of the form

#ifndef identifier

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#**else**

and then by a control line

#**endif**

If the checked condition is true then any lines between #**else** and #**endif** are ignored. If the checked condition is false then any lines between the test and an #**else** or, lacking an #**else**, the #**endif**, are ignored.

These constructions may be nested.

### 12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

#**line** *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

## 13. IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions, since **auto** functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning . . . ", it is implicitly declared to be **extern**.

In an expression, an identifier followed by ( and not already declared is contextually declared to be "function returning **int**".

## 14. TYPES REVISITED

This section summarizes the operations which can be performed on objects of certain types.

### 14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the **.** operator); or take its address (by unary **&**). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with **.** or **->**) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction

is not firmly enforced by the compiler. In fact, any lvalue is allowed before **.**, and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a **->** is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

### 14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
int (* funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

### 14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[ ]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules which apply to +, if **E1** is an array and E2 an integer, then **E1 [E2]** refers to the **E2**-th member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If **E** is an $n$-dimensional array of rank $i \times j \times \cdots \times k$, then **E** appearing in an expression is converted to a pointer to an ($n$-1)-dimensional array with rank $j \times \cdots \times k$. If the ***** operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to ($n$-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves

multiplying **i** by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

### 14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a char pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp =22.0 / 7.0;
```

**alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the use of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is measured in bytes, **char**s have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word just to their right. Thus **char** pointers are measured in units of $2^{16}$ bytes; everything else is measured in units of $2^{18}$ machine words, **double** quantities and aggregates containing them must lie on an even word address (0 mod $2^{19}$).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to **short** must be 0 mod 2, to int and **float** 0 mod 4, and to **double** 0 mod 8. Aggregates are aligned on the strictest boundary

required by any of their constituents.

## 15. CONSTANT EXPRESSIONS

In several places C requires expressions which evaluate to a constant: after case, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and sizeof expressions, possibly connected by the binary operators

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  <  >  <=  >=
```

or by the unary operators

```
-  ~
```

or by the ternary operator

```
? :
```

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary **&** operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary **&** can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

## 16. PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and left-to-right on other machines. These differences are invisible to

isolated programs which do not indulge in type punning (for example, by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bitfields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

## 17. ANACHRONISMS

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form $=op$ instead of $op=$ for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which actually decrements **x** since the = and the **-** are adjacent, but which might easily be intended to assign **-1** to **x** .

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
int x  = 1;
```

one used

```
int x  1;
```

The change was made because the initialization

```
int f  (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

## 18. SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

### 18.1 Expressions

The basic expressions are:

*expression:*
*primary*
***** *expression*
**&** *expression*
**-** *expression*
**!** *expression*
**~** *expression*
++ *lvalue*
**--** *lvalue*
*lvalue* ++
*lvalue* **--**
**sizeof** *expression*
**(** *type-name* **)** *expression*
*expression binop expression*
*expression* **?** *expression* **:** *expression*
*lvalue asgnop expression*
*expression* **,** *expression*

*primary:*
*identifier*
*constant*
*string*
**(** *expression* **)**
*primary* **(** *expression-list*$_{opt}$ **)**
*primary* **[** *expression* **]**
*lvalue* **.** *identifier*
*primary* **->** *identifier*

*lvalue:*
*identifier*
*primary* **[** *expression* **]**
*lvalue* **.** *identifier*
*primary* **->** *identifier*
** expression* **(** *lvalue* **)**

The primary-expression operators

**( )  [ ]  .  ->**

have highest priority and group left-to-right. The unary operators

***  &  -  !  ~  ++  --  sizeof(***type-name***)**

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority

decreasing as indicated below. The conditional operator groups right to left.

*binop:*
***** **/** **%**
**+** **-**
**>>** **<<**
**<** **>** **<=** **>=**
**&**
**^**
**|**
**&&**
**||**
**?** **:**

Assignment operators all have the same priority, and all group right-to-left.

*asgnop:*
**=** **+=** **-=** ***=** **/=** **%=** **>>=** **<<=** **&=** **^=** **|=**

The comma operator has the lowest priority, and groups left-to-right.

## 18.2 Declarations

*declaration:*
*decl-specifiers init-declarator-list*$_{opt}$ **;**

*decl-specifiers:*
*type-specifier decl-specifiers*$_{opt}$
*sc-specifier decl-specifiers*$_{opt}$

*sc-specifier:*
**auto**
**static**
**extern**
**register**
**typedef**

*type-specifier:*
**char**
**short**
**int**
**long**
**unsigned**
**float**
**double**
*struct-or-union-specifier*
*typedef-name*

*init-declarator-list:*
*init-declarator*
*init-declarator* **,** *init-declarator-list*

*init-declarator:*
*declarator initializer*$_{opt}$

*declarator:*
*identifier*
**(** *declarator* **)**
***** *declarator*
*declarator* **(** **)**
*declarator* **[** *constant-expression*$_{opt}$ **]**

*struct-or-union-specifier:*
**struct** **{** *struct-decl-list* **}**
**struct** *identifier* **{** *struct-decl-list* **}**
**struct** *identifier*
**union** **{** *struct-decl-list* **}**
**union** *identifier* **{** *struct-decl-list* **}**
**union** *identifier*

*struct-decl-list:*
*struct-declaration*
*struct-declaration struct-decl-list*

*struct-declaration:*
*type-specifier struct-declarator-list* **;**

*struct-declarator-list:*
*struct-declarator*
*struct-declarator* **,** *struct-declarator-list*

*struct-declarator:*
*declarator*
*declarator* **:** *constant-expression*
**:** *constant-expression*

*initializer:*
= *expression*
= **{** *initializer-list* **}**
= **{** *initializer-list* **,** **}**

*initializer-list:*
*expression*
*initializer-list* **,** *initializer-list*
**{** *initializer-list* **}**

*type-name:*
*type-specifier abstract-declarator*

*abstract-declarator:*
*empty*
**(** *abstract-declarator* **)**
***** *abstract-declarator*
*abstract-declarator* **(** **)**
*abstract-declarator* **[** *constant-expression*$_{opt}$ **]**

*typedef-name:*
*identifier*

## 18.3 Statements

*compound-statement:*
**{** *declaration-list*$_{opt}$ *statement-list*$_{opt}$ **}**

*declaration-list:*
*declaration*
*declaration declaration-list*

*statement-list:*
*statement*
*statement statement-list*

*statement:*
*compound-statement*
*expression* **;**
**if (** *expression* **)** *statement*
**if (** *expression* **)** *statement* **else** *statement*
**while (** *expression* **)** *statement*
**do** *statement* **while (** *expression* **) ;**
**for (** *expression-1*$_{opt}$ **;** *expression-2*$_{opt}$ **;** *expression-3*$_{opt}$ **)** *statement*
**switch (** *expression* **)** *statement*
**case** *constant-expression* **:** *statement*
**default :** *statement*
**break ;**
**continue ;**
**return ;**
**return** *expression* **;**
**goto** *identifier* **;**
*identifier* **:** *statement*

## 18.4 External definitions

*program:*
*external-definition*
*external-definition program*

*external-definition:*
*function-definition*
*data-definition*

*function-definition:*
*type-specifier*$_{opt}$ *function-declarator function-body*

*function-declarator:*
*declarator* **(** *parameter-list*$_{opt}$ **)**

*parameter-list:*
*identifier*
*identifier* **,** *parameter-list*

*function-body:*
*type-decl-list function-statement*

*function-statement:*
**{** *declaration-list*$_{opt}$ *statement-list* **}**

*data-definition:*
**extern**$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ **;**
**static**$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ **;**

### 18.5 Preprocessor

#**define** *identifier token-string*
#**define** *identifier* ( *identifier* , . . . , *identifier* ) *token-string*
#**undef** *identifier*
#**include** "*filename*"
#**include** <*filename*>
#**if** *constant-expression*
#**ifdef** *identifier*
#**ifndef** *identifier*
#**else**
#**endif**
#**line** *constant identifier*

# Recent Changes to C

*November 15, 1978*

A few extensions have been made to the C language beyond what is described in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1978.

## 1. STRUCTURE ASSIGNMENT

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

## 2. ENUMERATION TYPE

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

*enum-specifier*

with syntax

*enum-specifier:*
**enum { ***enum-list* **}**
**enum** *identifier* **{** *enum-list* **}**
**enum** *identifier*

*enum-list:*
*enumerator*
*enum-list* **,** *enumerator*

*enumerator:*
*identifier*
*identifier* = *constant-expression*

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, puce };
. . .
enum color *cp, col;
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were **int**.

# Appendix B: RATFOR — A Preprocessor for a Rational Fortran

*Brian W. Kernighan*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N-1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

## 2. LANGUAGE DESCRIPTION

**Design**

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

**Statement Grouping**

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is

true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "**x>100**"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**'s: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <== 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

### The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if(a<=b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0 else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

### Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to -1 if **x** is less than zero, to +1 if x is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        – – –
else if (...)
        – – –
else if (...)
        – – –
...
else
        – – –
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

**if-else ambiguity**

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the else go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

### The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

**switch (***expression***) {**

**case** *expr1* **:**
*statements*
**case** *expr2, expr3* **:**
*statements*
**...**
**default:**
*statements*
**}**

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match expression, and there is a default section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

### The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
        do 10 i = 1, n
                x(i) = 0.0
                y(i) = 0.0
                z(i) = 0.0
10      continue
```

The syntax is:

```
do legal-Fortran-DO-text
        Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do**.

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to -1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

**"break" and "next"**

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

**break** and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

**break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

**The "while" Statement**

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <=k)
        do i = j, k  {
                _ _ _

        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
#  returns sin(x) to accuracy e, by
#  sin(x) = x - x**3/3! + x**5/5! - ...

sin = x
term = x

i = 3
while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
}

return
end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x**3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal.

The language at one's disposal strongly influences how one thinks about a problem.

**The "for" Statement**

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-l))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

*init* is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init*, *condition*, and *increment* may be omitted, although the semicolons must always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) !=blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken, (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
            I = 81 - J
            IF (CARD(I) .NE.  BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** is zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
      sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

### The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
      Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

### More on break and next

**break** exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of do, while and **repeat-until** and to the increment step of a **for**.

**"return" Statement**

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value -1.

```
# equal _ compare str1 to str2;
# return 1 if equal, 0 if not
        integer function equal(strl, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1) {
                        equal = 1
                        return
                }
        equal = 0
        return
        end
```

In many languages (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function **F**, **return**(expression) is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
# return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1)
                        return(1)
        return(0)
        end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

**Cosmetics**

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

```
= i + l)
```

**Free-form Input**

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & ( _

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format ("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

**Translation Services**

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

| | | | |
|---|---|---|---|
| == | .eq. | != | .ne. |
| > | .gt. | >= | .ge. |
| < | .lt. | <= | .le. |
| & | .and. | \| | .or. |
| ! | .not. | ¬ | .not. |

In addition, the following translations are provided for input devices with restricted character sets.

| | | | |
|---|---|---|---|
| [ | { | ] | } |
| $( | { | $) | } |

**"define" Statement**

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

**define** is typically used to create symbolic parameters:

```
define      ROWS    100
define      COLS    50

dimension a(ROWS), b(ROWS, COLS)
        if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define YES  1
define NO   0
define EOS  -1
define ARB  100

#  equal _ compare str1 to str2;
#      return YES if equal, NO if not
       integer function equal(str1, str2)
       integer str1(ARB), str2(ARB)
       integer i

       for (i = 1; str1(i) == str2(i); i = i + 1)
               if (str1(i) === EOS)
                       return(YES)
       return(NO)
       end
```

**"include" Statement**

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

subroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

**Pitfalls, Botches, Blemishes and other Failings**

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

## 3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system [5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog    : stat
        | prog  stat
stat    : if (...) stat
        | if (...) stat else stat
        | while (...) stat
        | for (...; ...; ...) stat
        | do ...  stat
        | repeat stat
        | repeat stat until (...)
        | switch (...) { case ...: prog ...
              default: prog }
        | return
        | break
        | next
        | digits        stat
        | { prog }
        | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable"; In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords

is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1 continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt.  0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument

usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

## 4. EXPERIENCE

### Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

### Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the

implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

## 5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

**Acknowledgements**

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

**References**

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter*, August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

# Appendix C: The M4 Macro Processor

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

## Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The #**define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1], Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be res-canned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is res-canned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

## Usage

On UNIX, use

**m4 [files]**

Each argument file is processed in order; if there are no arguments, or if an argument is '-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

**m4 [files] >outputfile**

On GCOS, usage is identical, but the program is called **./m4**.

**Defining Macros**

The primary built-in function of M4 is **define**, which is used to define new macros. The input

**define(name, stuff)**

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**, **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter), **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

**define(N, 100)**
**...**
**if (i > N)**

defines **N** to be 100, and uses this "symbolic constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by '(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

**define(N, 100)**
**...**
**if (NNN > 100)**

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

**define(N, 100)**
**define(M, N)**

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true — **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

**define(M, 100)**

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

**define(M,N)**
**define(N, 100)**

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

**Quoting**

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, N)
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define´ = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N´, 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ´ are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**, **undefine** removes the definition of some macro or built-in:

```
undefine(`N´)
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins

can be removed with **undefine**, as in

**undefine(`define´)**

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

**ifdef(`unix´, `define(wordsize,16)´ )**
**ifdef(`gcos´, `define(wordsize,36)´ )**

makes a definition appropriate for the particular machine. Don't forget the quotes!

**ifdef** actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

**ifdef(`unix´, on UNIX, not on UNIX)**

**Arguments**

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

**define(bump, $1 = $1 + 1)**

generates code to increment its argument by 1:

**bump(x)**

is

**x = x + 1**

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

**define(cat, $1$2$3$4$5$6$7$8$9)**

Thus

**cat(x, y, z)**

is equivalent to

**xyz**

**$4** through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

**define(a, b c)**

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

**define(a, (b,c))**

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

### Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

**define(N, 100)**
**define(N1, `incr(N)´)**

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^   (exponentiation)
*  /  % (modulus)
+  -
==   !==   <   <=   >   >=
!          (not)
& or &&  (logical and)
| or ||            (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2******N**+**1**. Then

**define(N, 3)**
**define(M, `eval(2**N+1)´)**

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

### File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

**include(filename)**

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

**divert(n)**

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

**undivert**

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

### System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

**syscmd(date)**

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp*: a string of XXXXX in the argument is replaced by the process id of the current process.

### Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

**ifelse(a, b, c, d)**

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

**define(compare, `ifelse($1, $2, yes, no)´)**

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

**ifelse** can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

**ifelse(a, b, c, d, e, f, g)**

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

**ifelse(a, b, c)**

is **c** if **a** matches **b**, and null otherwise.

**String Manipulation**

The built-in **len** returns the length of the string that makes up its argument. Thus

**len(abcdef)**

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

**substr(`now is the time´, 1)**

is

**ow is the time**

If **i** or **n** are out of range, various sensible things happen.

**index(s1, s2)** returns the index (position) in **s1** where the string **s2** occurs, or -1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

**translit(s, f, t)**

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

**translit(s, aeiou, 12345)**

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

**translit(s, aeiou)**

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you

say

**define(N, 100)**
**define(M, 200)**
**define(L, 300)**

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

**divert(-1)**
**define(...)**
**...**
**divert**

### Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

**errprint(`fatal error´)**

**dumpdef** is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

### Summary of Built-ins

Each entry is preceded by the page number where it is described.

**Acknowledgements**

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

**References**

[1] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

## Appendix D: BC-An Arbitrary Precision Desk-Calculator Language

*ABSTRACT*

BC is a language and a compiler for doing arbitrary precision arithmetic. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

### Introduction

BC was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

**Simple Computations with Integers**

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

142857 + 285714

the program responds immediately with the line

428571

The operators -, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

7+-3

is interpreted to mean that -3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and -. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

a^b^c and a^(b^c)

are equivalent, as are the two expressions

a*b*c and (a*b)*c

BC shares with Fortran and C the undesirable convention that

a/b*c is equivalent to (a/b)*c

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

x = x + 3

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

x = sqrt(191) x

produce the printed result

13

**Bases**

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

ibase = 8
11

will produce the output line

9

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

ibase = 10

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A-F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10-15 respectively. The statement

ibase = A

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

obase = 16 1000

will produce the output line

3E8

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

**Scaling**

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

**Functions**

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return

statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
    auto z
    z = x*y
    return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

**Subscripted Variables**

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[ ])
define f(a[ ])
auto a[ ]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

**Control Statements**

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if (relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i==1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j = j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
            a = a*x
            b = b*n
            c = c + a/b
            n = n + 1
            if(c==d) return(c)
            d = c
    }
}
```

**Some Details**

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

| | | |
|---|---|---|
| x=y=z | is the same as | x=(y=z) |
| x =+ y | | x = x+y |
| x =- y | | x = x-y |
| x =* y | | x = x*y |
| x =/ y | | x = x/y |
| x =% y | | x = x%y |
| x =^ y | | x = x^y |
| x++ | | (x=x+1)-1 |
| x-- | | (x=x-1)+1 |
| ++x | | x = x+1 |
| --x | | x = x-1 |

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

**Three Important Things**

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

```
bc-l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

**Acknowledgement**

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

**References**

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines*, Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC— Yet Another Compiler-Compiler*. Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC - An Interactive Desk Calculator*.

# Appendix

## 1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [ ] is optional.

## 2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

### 2.1. Comments

Comments are introduced by the characters /* and terminated by */.

### 2.2. Identifiers

There are three kinds of identifiers - ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named x, all of which are separate and distinct.

### 2.3. Keywords

The following are reserved keywords:

| | |
|---|---|
| **ibase** | **if** |
| **obase** | **break** |
| **scale** | **define** |
| **sqrt** | **auto** |
| **length** | **return** |
| **while** | **quit** |
| **for** | |

### 2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A-F** are also recognized as digits with values 10-15, respectively.

## 3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

### 3.1. Primitive expressions

#### 3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

##### 3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

##### 3.1.1.2. *array-name*[*expression*]

Array elements are named expressions. They have an initial value of zero.

##### 3.1.1.3. **scale, ibase** and **obase**

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

#### 3.1.2. Function calls

##### 3.1.2.1. *function-name*(**[***expression***[,***expression***. . . ] ]**)

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

##### 3.1.2.2. **sqrt(***expression***)**

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

##### 3.1.2.3. **length(***expression***)**

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

##### 3.1.2.4. **scale(***expression***)**

The result is the scale of the expression. The scale of the result is zero.

#### 3.1.3. Constants

Constants are primitive expressions.

#### 3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

### 3.2. Unary operators

The unary operators bind right to left.

#### 3.2.1. **-***expression*

The result is the negative of the expression.

#### 3.2.2. +*named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

#### 3.2.3. **--***named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

#### 3.2.4. *named-expression*++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

#### 3.2.5. *named-expression***--**

The named expression is decremented by one. The result is the value of the named expression before decrementing.

### 3.3. Exponentiation operator

The exponentiation operator binds right to left.

#### 3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If $a$ is the scale of the left expression and $b$ is the absolute value of the right expression, then the scale of the result is:

min ( $a \times b$, max ( **scale**, $a$ ) )

### 3.4. Multiplicative operators

The operators *, /, % bind left to right.

#### 3.4.1. *expression* * *expression*

The result is the product of the two expressions. If $a$ and $b$ are the scales of the two expressions, then the scale of the result is:

min ( $a+b$, max ( **scale**, $a$, $b$ ))

**3.4.2.** *expression / expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

**3.4.3.** *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, *a*%*b* is *a*-*a*/*b***b*.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

### 3.5. Additive operators

The additive operators bind left to right.

**3.5.1.** *expression* + *expression*

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

**3.5.2.** *expression* — *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

### 3.6. assignment operators

The assignment operators bind right to left.

**3.6.1.** *named-expression* = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

**3.6.2.** *named-expression* =+ *expression*

**3.6.3.** *named-expression* =**-** *expression*

**3.6.4.** *named-expression* =***** *expression*

**3.6.5.** *named-expression* =/ *expression*

**3.6.6.** *named-expression* =% *expression*

**3.6.7.** *named-expression* =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

## 4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

**4.1.** *expression* < *expression*

**4.2.** *expression* > *expression*

**4.3.** *expression* <= *expression*

**4.4.** *expression* >= *expression*

**4.5.** *expression* == *expression*

**4.6.** *expression* **!**= *expression*

## 5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls, **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

## 6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

### 6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

### 6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

### 6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

### 6.4. If statements

**if(***relation***)***statement*

The substatement is executed if the relation is true.

### 6.5. While statements

**while**(*relation*)*statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

### 6.6. For statements

**for**(*expression*; *relation*; *expression*)*statement*

The for statement is the same as

*first-expression*
**while**(*relation*) {
    *statement*
    *last-expression*
}

All three expressions must be present.

### 6.7. Break statements

**break**

**break** causes termination of a **for** or **while** statement.

### 6.8. Auto statements

**auto** *identifier* [ **,***identifier* ]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

### 6.9. Define statements

**define**([*parameter*[**,***parameter*...] ] ){
*statements*}

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

### 6.10. Return statements return

**return**(*expression*)

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

### 6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

## Appendix E: DC-An Interactive Desk Calculator

DC is an arbitrary precision arithmetic package implemented in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

### SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

**number**

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A-F which are treated as digits with values 10-15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

\+ - * % ^

The top two values on the stack are added (+), subtracted (—), multiplied (*), divided (/), remaindered (%), or exponentiated (A). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

**s***x*
: The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

**l***x*
: The value in register *x* is pushed onto the stack. The register x is not altered. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

**d**
: The top value on the stack is duplicated.

**p**
: The top value on the stack is printed. The top value remains unchanged.

**f**
: All values on the stack and in registers are printed.

**x**
: treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

**[ ... ]**
: puts the bracketed character string onto the top of the stack.

**q**
: exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

**<***x* **>***x* **=***x* **!<***x* **!>***x* **!=***x*
: The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

**v**
: replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

**!**

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

**c**

All values on the stack are popped; the stack becomes empty.

**i**

The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

**o**

The top value on the stack is popped and used as the number radix for further output. If **o** is capitalized, the value of the output base is pushed onto the stack.

**k**

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, the value of the scale factor is pushed onto the stack.

**z**

The value of the stack level is pushed onto the stack.

**?**

A line of input is taken from the input source (usually the console) and executed.

## DETAILED DESCRIPTION

### Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0-99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always -1 and all other digits are in the range 0-99.

The digit preceding the high order -1 digit is never a 99. The representation of -157 is 43,98,-1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

**The Allocator**

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the inform at ion-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

### Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations, **scale** is the bound on the number of decimal places retained in arithmetic computations, **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack, **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

### Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,-1 by the digit -1. In any case, digits which are not in the range 0-99 must be brought into that range, propagating any carries or borrows that result.

### Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

### Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

### Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

### Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = (x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

### Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

### Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The

hexadecimal digits A-F correspond to the numbers 10-15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push the value of the input base on the stack.

### Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

### Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

### Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register *x*. *x* can be any character. **l***x* puts the contents of register *x* on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

### Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

### Subroutine Definitions and Calls

Enclosing a string in [] pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

### Internal Registers — Programming DC

The load and store commands together with [] to store strings, **x** to execute and the testing commands '<', '>', '—\ '!<', '!>', '!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the

stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+ si  li10>a]sa
0si  lax
```

### Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **h***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as push-down stacks. **l** doesn't effect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. **;***x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

### Miscellaneous Commands

The command **!** interprets the rest of the line as a UNIX command and passes it to UNIX to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

## DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were

to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

**References**

[1] L. L. Cherry, R. Morris, *BC - An Arbitrary Precision Desk-Calculator Language*.

[2] K. C. Knowlton, *A Fast Storage Allocator*, Comm. ACM **8**, pp. 623-625 (Oct. 1965).

*May 1979*

# Appendix F: Curses

*Kenneth C. R. C. Arnold*

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

## 1. Overview

In making available the generalized terminal descriptions in **/etc/termcap**, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

### 1.1. Terminology

In this document, the following terminology is kept to with reasonable consistency:

***window:*** An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

***terminal:*** Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*:

***screen:*** This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

### 1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

# **include** <**curses.h**>

at the top of the program source. The header file <**curses.h**> needs to include <**sgtty.h**>, so that one need not do so oneself[1]. Compilations should have the following form:

**cc** [ *flags* ] file ... **-lcurses -ltermlib**

1 The screen package also uses the Standard I/O library, so <**curses.h**> includes <**stdio.h**>. It is redundant (but harmless) for the programmer to do it too.

## 1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called, *refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does* ***not*** *change the terminal*. Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

## 1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for **w**indow-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
```

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

waddch(win, ch);

can be replaced by

mvwaddch(win, y, x, ch);

Note that the window description pointer
(*win*)
comes before the added (y, x) co-ordinates.
If such pointers are need,
they are always the first parameters passed.

**2.**

**Variables**

Many variables which are used to describe the terminal environment are available to the programmer. They are:

| type | name | description |
|---|---|---|
| WINDOW * | curscr | current version of the screen (terminal screen). |
| WINDOW * | stdscr | standard screen. Most updates are usually done here. |
| char * | Def_term | default terminal type if type cannot be determined |
| bool | My_term | use the terminal specification in *Def_ term* as terminal, irrelevant of real terminal type |
| char * | ttytype | full name of the current terminal. |
| int | LINES | number of lines on the terminal |
| int | COLS | number of columns on the terminal |
| int | ERR | error flag returned by routines on a fail. |
| int | OK | error flag returned by routines when things go right. |

There are also several "#define" constants and types which are of general usefulness:

| | |
|---|---|
| reg | storage class "register" (e.g., *reg int i*;) |
| bool | boolean type, actually a "char" (e.g., *bool doneit;*) |
| TRUE | boolean "true" flag (1). |
| FALSE | boolean "false" flag (0). |

**addch(ch) †**
*char ch;*

**waddch(win, ch)**
*WINDOW *win;*
*char ch;*

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline (´\n´) the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return (´\r´) will move to the beginning of the line on the window. Tabs (´\t´) will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

**addstr(str) †**
*char *str;*

**waddstr(win, str)**

*WINDOW* **win;*
*char* **str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

**box(win, vert, hor)**
*WINDOW* **win;*
*char* *vert, hor;*

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

**clear() †**

**wclear(win)**
*WINDOW* **win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

**clear ok(scr, boolf) †**
*WINDOW* **scr;*
*bool* *boolf;*

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh() refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

**clrtobot() †**

**wclrtobot(win)**
*WINDOW* **win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

**clrtoeol() †**

**wclrtoeol(win)**
*WINDOW* **win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

**delch()**

**wdelch(win)**
*WINDOW *win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

**deleteln()**

**wdeleteln(win)**
*WINDOW *win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

**erase() †**

**werase(win)**
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analogous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

**insch(c)**
*char c;*

**winsch(win, c)**
*WINDOW *win;*
*char c;*

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

**insertln()**

**winsertln(win)**
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged. This returns ERR if it would cause the screen to scroll illegally.

**move(y, x) †**
*int y, x;*

**wmove(win, y, x)**
*WINDOW* **win;*
*int* *y, x;*

Change the current (y, x) co-ordinates of the window to (*y*, *x*). This returns ERR if it would cause the screen to scroll illegally.

**overlay(win1, win2)**
*WINDOW* **win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

**overwrite(win1, win2)**
*WINDOW* **win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

**printw(fmt, arg1, arg2, ...)**
*char* **fmt;*

**wprintw(win, fmt, arg1, arg2, ...)**
*WINDOW* **win;*
*char* **fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

**refresh() †**

**wrefresh(win)**
*WINDOW* **win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

**standout() †**

**wstandout(wm)**
*WINDOW* **win;*

**standend() †**

**wstandend(win)**
*WINDOW *win;*

Start and stop putting characters onto win in standout mode, *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability), *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see next major section).

**crmode() †**

**nocrmode() †**

Set or unset the terminal to/from cbreak mode.

**echo() †**

**noecho() †**

Sets the terminal to echo or not echo characters.

**getch() †**

**wgetch(win)**
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echoes it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

**getstr(str) †**
*char *str;*

**wgetstr(win, str)**
*WINDOW *win;*
*char *str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

**raw() †**

**noraw() †**

Set or unset the terminal to/from raw mode. On version 7 UNIX[3] this also turns of newline mapping (see *nl()*).

**scanw(fmt, arg1, arg2, ...)**
*char* **fmt;*

**wscanw(win, fmt, arg1, arg2, ...)**
*WINDOW* **win;*
*char* **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

**delwin(win)**
*WINDOW* **win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

**endwin()**

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

**getyx(win, y, x)** †
*WINDOW* **win;*
*int* *y, x;*

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

**inch()** †

**winch(win)** †
*WINDOW* **win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

**initscr()**

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal

[3] UNIX is a trademark of Bell Laboratories.

whose name is pointed to by *Def_term* (initially "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

**leaveok(win, boolf) †**
*WINDOW* **win;*
*bool* *boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initially FALSE) retains its value until changed by the user.

**longname(termbuf, name)**
*char* **termbuf, *name;*

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

**mvwin(win, y, x)**
*WINDOW* **win;*
*int* *y, x;*

Move the home position of the window *win* from its current starting coordinates to (*y*, *x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything.

*WINDOW**
**newwin(lines, cols, begin_y, begin_x)**
*int* *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

**nl() †**

**nonl() †**

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <**RETURN**> to <**LINE-FEED**>. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

**scrollok(win, boolf) †**
*WINDOW* **win;*
*bool* *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

**touchwin(win)**
*WINDOW *win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

*WINDOW**
**subwin(win, lines, cols, begin_y, begin_x)**
*WINDOW *win;*
*int lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y*, *begin_x*) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y*, *begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively.

**unctrl(ch) †**
*char ch;*

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their uppercase equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have #**include** <**unctrl.h**> in your file.

**gettmode()**

Get the tty stats. This is normally called by *initscr()*.

**mvcur(lasty, lastx, newy, newx)**
*int lasty, lastx, newy, newx;*

Moves the terminal's cursor from (*lasty*, *lastx*) to (*newy*, *newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

**8croll(win)**
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

**savetty() †**

**resetty() †**

*savetty()* saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

**setterm(name)**
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

**tstp()**

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

## 3. Capabilities from termcap

### 3.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

### 3.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

### 3.3. Variables Set By setterm()

variables set by *setterm()*

| Type | Name | Pad | Description |
|---|---|---|---|
| char * | AL | P* | Add new blank Line |
| bool | AM | | Automatic Margins |
| char * | BC | | Back Cursor movement |
| bool | BS | | Backspace works |
| char * | BT | P | Back Tab |
| bool | CA | | Cursor Addressable |
| char * | CD | P* | Clear to end of Display |
| char * | CE | P | Clear to End of line |
| char * | CL | P* | CLear screen |
| char * | CM | P | Cursor Motion |
| char * | DC | P* | Delete Character |
| char * | DL | P* | Delete Line sequence |
| char * | DM | | Delete Mode (enter) |

| | | | |
|---|---|---|---|
| char * | DO | | DOwn line sequence |
| char * | ED | | End Delete mode |
| bool | EO | | can Erase Overstrikes with ´ ´ |
| char * | EI | | End Insert mode |
| char * | HO | | HOme cursor |
| bool | HZ | | HaZeltine ~ braindamage |
| char * | IC | P | Insert Character |
| bool | IN | | Insert-Null blessing |
| char * | IM | | enter Insert Mode (IC usually set, too) |
| char * | IP | P* | Pad after char Inserted using IM+IE |
| char * | LL | | quick to Last Line, column 0 |
| char * | MA | | ctrl character MAp for cmd mode |
| bool | MI | | can Move in Insert mode |
| bool | NC | | No Cr: \r sends \r\n then eats \n |
| char * | ND | | Non-Destructive space |
| bool | OS | | OverStrike works |
| char | PC | | Pad Character |
| char * | SE | | Standout End (may leave space) |
| char * | SF | P | Scroll Forwards |
| char * | SO | | Stand Out begin (may leave space) |
| char * | SR | P | Scroll in Reverse |
| char * | TA | P | TAb (not ^I or with padding) |
| char * | TE | | Terminal address enable Ending sequence |
| char * | TI | | Terminal address enable Initialization |
| char * | UC | | Underline a single Character |
| char * | UE | | Underline Ending sequence |
| bool | UL | | UnderLining works even though !OS |
| char * | UP | | UPline |
| char * | US | | Underline Starting sequence[4] |
| char * | VB | | Visible Bell |
| char * | VE | | Visual End sequence |
| char * | VS | | Visual Start sequence |
| bool | XN | | a Newline gets eaten after wrap |

Names starting with *X* are reserved for severely nauseous glitches

### 3.4. Variables Set By gettmode()

variables set by *gettmode()*

| type | name | description |
|---|---|---|
| bool | NONL | Term can't hack linefeeds doing a CR |
| bool | GT | Gtty indicates Tabs |
| bool | UPPERCASE | Terminal generates only uppercase letters |

## 4. The WINDOW structure

The WINDOW structure is defined as follows:

```
# define        WINDOW struct _win_st

struct _win_st {
        short   _cury, _curx;
```

[4] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm()*

```
        short   _maxy, _maxx;
        short   _begy, _begx;
        short   _flags;
        bool    _clear;
        bool    _leave;
        bool    _scroll;
        char    **_y;
        short   *_firstch;
        short   *_lastch;
};
# define        _SUBWIN         01
# define        _ENDLINE        02
# define        _FULLWIN        04
# define        _SCROLLWIN      010
# define        _STANDOUT       0200
```

*_cury* and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point, *_maxy* and *_maxx* are the maximum values allowed for (*_cury*, *_curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy*, *_begx*), not the terminal's home.

*_clear* tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

*_y* is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line.

*_flags* can have one or more values or'd into it. **_SUBWIN** means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. **_ENDLINE** says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_STANDOUT** says that all characters added to the screen are in standout mode.

## 5. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

5 AH variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

## 6. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

### 6.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
# include  <curses.h>
# include  <signal.h>

/*
 *  the idea for this program was a product of the imagination of
 *  Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS           80
# define        NLINES          24
# define        MAXPATTERNS     4

struct locs {
        char    y, x;
};

typedef struct locs     LOCS;  /* current board layout */

LOCS    Layout[NCOLS * NLINES];

int     Pattern,                /* current pattern number */
        Numstars;               /* number of stars in pattern */

main() {

    char        *getenv();
    int         die();

    srand(getpid());            /* initialize random sequence */

    initscr();
    signal(SIGINT, die);
    noecho();
    nonl();
    leaveok(stdscr, TRUE);
    scrollok(stdscr, FALSE);

    for (;;) {
```

```
                makeboard();            /* make the board setup */
                puton('*');             /* put on '*' s */
                puton(' ');             /* cover up with ' ' s */
        }
}

/*
 *  On program exit, move the cursor to the lower left corner by
 *  direct addressing, since current location is not guaranteed.
 *  We lie and say we used to be at the upper right corner to guarantee
 *  absolute addressing.
 */
die() {

        signal(SIGINT, SIGIGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}

/*
 *  Make the current board setup.  It picks a random pattern and
 *  calls ison() to determine if the character is on that pattern
 *  or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 *  Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int         y, x; {

        switch (Pattern) {
                case 0:         /* alternating lines */
                        return !(y & 01);
                case 1:         /* box */
                        if (x >= LINES && y >= NCOLS)
                                return FALSE;
                        if (y < 3 || y >= NLINES - 3)
```

```
                return TRUE;
        return (x < 3 || x >= NCOLS - 3);
    case 2:     /* holy pattern! */
        return ((x + y) & 01);
    case 3:     /* bar across center */
        return (y >= 9 && y <= 15);
    }
    /* NOTREACHED */
}

puton(ch)
reg char        ch; {

    reg LOCS        *lp;
    reg int         r;
    reg LOCS        *end;
    LOCS            temp;

    end = &Layout[Numstars];
    for (lp = Layout; lp < end; lp++) {
        r = rand() % Numstars;
        temp = *lp;
        *lp = Layout[r];
        Layout[r] = temp;
    }

    for (lp = Layout; lp < end; lp++) {
        mvaddch(lp->y, lp->x, ch);
        refresh();
    }
```

**6.2.**
**Life**

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include       <curses.h>
# include       <signal.h>

/*
 *          Run a life game.  This is a demonstration program for
 *  the Screen Updating section of the -lcurses cursor package.
 */

struct lst_st {                         /* linked list element */
        int             y, x;           /* (y, x) position of piece */
```

```
        struct lst_st       *next, *last;   /* doubly linked */
};

typedef struct lst_st   LIST;

LIST    *Head;                          /* head of linked list */

main(ac, av)
int     ac;
char    *av[]; {

        int     die();

        evalargs(ac, av);               /* evaluate arguments */

        initscr();                      /* initialize screen package */
        signal(SIGINT, die);            /* set to restore tty stats */
        crmode();                       /* set for char-by-char */
        noecho();                       /* input */
        nonl();                         /* for optimization */

        getstart();                     /* get starting position */
        for (;;) {
                prboard();              /* print out current board */
                update();               /* update board position */
        }
}

/*
 *  This is the routine which is called when rubout is hit.
 *  It resets the tty stats to their original values.  This
 *  is the normal way of leaving the program.
 */
die() {
        signal(SIGINT, SIG_IGN);        /* ignore rubouts */
        mvcur(0, COLS-1, LINES-1, 0);   /* go to bottom of screen */
        endwin();                       /* set terminal to initial state */
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and .  are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart() {

        reg char        c;
        reg int         x, y;
```

```
        box(stdscr, '|', '_');                  /* box in the screen */
        move(1, 1);                             /* move to upper left corner */

        do {
                refresh();                      /* print current position */
                if((c==getch())== 'q')
                        break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
                  case '.':
                        adjustyx(c);
                        break;
                  case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                  case 'x':
                        addch('X');
                        break;
                  case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                       /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
        }

/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;
```

```
        erase();                                    /* clear out last position */
        box(stdscr, '|', '_');                      /* box in the screen */

        /*
         *  go through the list adding each piece to the newly
         *  blank board
         */
         for (hp = Head; hp; hp = hp->next)
                 mvaddch(hp->y, hp->x, 'X');


        refresh();
}
```

**7.**

**Motion optimisation**

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

### 7.1. Twinkle

The twinkle program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

        reg char        *sp;
        char            *getenv();
        int             _putchar(), die();

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
                gettmode();
                if (sp=getenv("TERM"))
                        setterm(sp);
                signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
```

```
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 *      _putchar defined for tputs() (and _puts())
 */
_putchar(c)
reg char        c; {

        putchar(c);
}

puton(ch)
char    ch;{

        static int              lasty, lastx;
        reg LOCS                *lp;
        reg int                 r;
        reg LOCS                *end;
        LOCS                    temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
                        /* prevent scrolling */
                if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                        mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                        lasty = lp->y;
                        if ((lastx = lp->x + 1) >= NCOLS)
                                if (AM) {
                                        lastx = 0;
                                        lasty++;
                                }
                                else
                                        lastx = NCOLS - 1;
                }
}
```

# Appendix G: Introduction to SENDMAIL

Eric Allman†

*Britton-Lee, Inc. 1919 Addison Street, Suite 105. Berkeley, California 94704.*

*ABSTRACT*

*Routing mail through a heterogeneous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an ad hoc basis. However, this approach has become unmanageable as internets grow.*

*Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style ad hoc addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.*

*Sendmail* implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the address space.

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

*Sendmail* is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares sendmail to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

## 1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formatted addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really are used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

## 2. OVERVIEW

### 2.1. System Organization

*Sendmail* neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

### 2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

#### 2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

#### 2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

#### 2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2BSD IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a *sendmail* process on another machine.

### 2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

#### 2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is

1except when mailing to a file, when sendmail does the delivery directly.

deferred until delivery. Forwarding is also performed as the local addresses are verified.

*Sendmail* appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages delivered to the same recipient, as might occur if a person is in two groups.

#### 2.3.2. Message collection

*Sendmail* then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

#### 2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to *sendmail*. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

#### 2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

#### 2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

### 2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

### 2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling *sendmail* the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

[2]Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message - the "return to sender" function is always handled in one of these two ways.

## 3. USAGE AND IMPLEMENTATION

### 3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

(2) Anything in angle brackets ("< >") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name < machine-address >

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes ( " ) quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently - for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

### 3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar (" | ") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

### 3.3. Aliasing, Forwarding, Inclusion

*Sendmail* reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

#### 3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

#### 3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

[3]Disclaimer: Some special processing is done after rewriting local names; see below.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

" | /usr/local/newmail myname"

will use a different incoming mailer.

#### 3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

### 3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

### 3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

### 3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

### 3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recompiled except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

#### 3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

#### 3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

#### 3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

#### 3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcgl!tef

might be mapped into:

tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

### 3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

## 4. COMPARISON WITH OTHER MAILERS

### 4.1. Delivermail

*Sendmail* is an outgrowth of *delivermail*. The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

### 4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail*. For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[4].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

[4]Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

[5]The MMDF equivalent of a *sendmail* "mailer."

### 4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

## 5. EVALUATIONS AND FUTURE PLANS

*Sendmail* is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

*Sendmail* has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Pos-tel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

[6]This is similar to the NBS standard.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

REFERENCES


[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M.* 25, 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual*. R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages*. RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System*. R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility - MMDF*. 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*. RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19*, 7. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook*. NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard*. Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network*. RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems*. Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description*. Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., *Revised FTP Reply Codes*. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol*. NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol*. RFC 753, DEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure*. In *Proceedings of the Sixth Data Communications Symposium*, IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents*. RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol*. RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network*. University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual*. University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol*. RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications*. RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition*, Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

# Appendix H: Sendmail Installation and Operation Guide

Eric Allman
Britton-Lee, Inc.

Version 4.2

*Sendmail* implements a general purpose internetwork mail routing facility under the UNIX operating system. It is not tied to any one transport protocol - its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for sendmail, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The addenda give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail - An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

## 1. BASIC INSTALLATION

There are two basic steps to installing *sendmail*. The hard part is to build the configuration table. This is a file that *sendmail* reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of *sendmail* assuming you can use one of the existing configurations and that the standard installation parameters are acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

### 1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the *sendmail* directory. The ones used at Berkeley are in *m4* (1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucsfcgl.cf
```

This file is now ready for installation as */usr/lib/sendmail.cf*

### 1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/*system*.cf" where *system* is the name of your system (i.e., what is returned by *hostname* (1)). If you do not have *hostname* you can use the declaration "HOST=*system*" on the *make* (1) command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

```
make
make install
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

### 1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

#### 1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2BSD directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

```
cdlib
make ndir
```

This will compile and install the 4.2 compatibility routines in the library. You should then type:

```
cd lib      # if required
make
```

This will recompile and fill the library.

### 1.3.2. /usr/lib/sendmail

The binary for *sendmail* is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm -f *.o
make
cp sendmail /usr/lib
```

### 1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

### 1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

### 1.3.5. /usr/lib/sendmail.cf

The configuration file must be installed in /usr/lib. This is described above.

### 1.3.6. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the *sendmail* owner and mode 755.

### 1.3.7. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm* (3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

### 1.3.8. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/*sendmail*.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

### 1.3.9. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm -f [lnx]f*)
        /usr/lib/sendmail -bd -q30m &
        echo -n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "-bd" causes it to listen on the SMTP port, and "-q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the **-bd** flag.

### 1.3.10. /usr/lib/sendmail.hf

This is the help file used by the SMTP HELP command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

### 1.3.11. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail .st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

### 1.3.12. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that *sendmail* will log in. For a complete description of syslog, see the manual page for *syslog* (8) (located in *sendmail/doc* on the distribution).

### 1.3.13. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the **-bi** flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

### 1.3.14. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the **-bp** flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

## 2. NORMAL OPERATIONS

### 2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **-bz** flag:

/usr/lib/sendmail -bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **-C** flag is specified or if *sendmail* detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

### 2.2. The System Log

The system log is supported by the *syslog* (8) program.

#### 2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

#### 2.2.2. Levels

If you have *syslog* (8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

### 2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

#### 2.3.1. Printing the queue

The contents of the queue can be printed using the mailq command (or by specifying the **-bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

#### 2.3.2. Format of queue files

All queue files have the form *x***f***AA99999* where *AA99999* is the id for this file and the *x* is a type. The types are:

d The data file. The message body (excluding the header) is kept in this file.

l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous **lf** file can cause a job

to apparently disappear (it will not even time out!).

n This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q The queue control file. This file contains the information necessary to process the job.

t A temporary file. These are an image of the **qf** file when it is being rebuilt. It should be renamed to a **qf** file very quickly.

x A transcript file, existing during the life of a session showing everything that happens during that session.

The qf file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D The name of the data file. There may only be one of these lines.

H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S The sender address. There may only be one of these lines.

T The job creation time. This is used to compute when to time out the job.

P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

### 2.3.3. Forcing the queue

*Sendmail* should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it

ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The **-oQ** flag specifies an alternate queue directory and the **-q** flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the **-v** flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

## 2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

```
name: name1, name2, ...
```

Only local names may be aliased; e.g.,

```
eric@mit-xx: eric@berkeley
```

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm* (3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

### 2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

```
newaliases
```

This is equivalent to giving *sendmail* the **-bi** flag:

```
/usr/lib/sendmail -bi
```

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1) The DBM version of the database is mode 666. -or-

(2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

### 2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. It will wait up to five minutes for this entry to appear, at which point it will force a rebuild itself[2].

### 2.4.3. List owners

If an error occurs on sending to a certain address, say "*x*", *sendmail* will look for an alias of the form "owner-*x*" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintenance of the list itself; in this case the list maintainer would be the owner of the list. For example:

```
unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
        sam@matisse
owner-unix-wizards: eric@ucbarpa
```

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

## 2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

```
mckusick@ernie
kirk@calder
```

then any mail arriving for "mckusick" will be redirected to the specified accounts.

## 2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

---

[1]The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

[2]Note: the "D" option must be specified in the configuration file for this operation to occur.

#### 2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete, if the mailer has the **l** flag (local delivery) set in the mailer descriptor.

#### 2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

#### 2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

## 3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Addendum A. Some important arguments are described here.

### 3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the **-q** flag. If you run in mode **f** or **a** this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in **q** mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

### 3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the **-bd** flag. The **-bd** flag and the **-q** flag may be combined in one call:

```
/usr/lib/sendmail -bd -q30m
```

### 3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the **-q** flag (with no value). It is entertaining to use the **-v** flag (verbose) when this is done to watch what happens:

```
/usr/lib/sendmail -q -v
```

### 3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the **-d** option; the syntax is:

| | |
|---|---|
| debug-flag: | **-d** debug-list |
| debug-list: | debug-option [ , debug-option ] |
| debug-option: | debug-range [ . debug-level ] |
| debug-range: | integer \| integer - integer |
| debug-level: | integer |

where spaces are for reading ease only. For example,

| | |
|---|---|
| -d12 | Set flag 12 to level 1 |
| -d12.3 | Set flag 12 to level 3 |
| -d3-17 | Set flags 3 through 17 to level 1 |
| -d3-17.4 | Set flags 3 through 17 to level 4 |

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

### 3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the **-C** flag; for example,

/usr/lib/sendmail -Ctest.cf

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **-C** flag has no value it defaults to *sendmail.cf* in the current directory.

### 3.6. Changing the Values of Options

Options can be overridden using the **-o** flag. For example,

/usr/lib/sendmail -oT2m

sets the **T** (timeout) option to two minutes for this run only.

## 4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

### 4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

| | |
|---|---|
| s | seconds |
| m | minutes |
| h | hours |
| d | days |
| w | weeks |

#### 4.1.1. Queue interval

The argument to the **-q** flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

#### 4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the **r** option in the configuration file.

#### 4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the **T** option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

/usr/lib/sendmail -oT1d -q

will run the queue and flush anything that is one day old.

### 4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

| | |
|---|---|
| i | deliver interactively (synchronously) |
| b | deliver in background (asynchronously) |
| q | queue only (don't deliver) |

There are tradeoffs. Mode "i" passes the maximum amount of information to the sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

### 4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

| | |
|---|---|
| 0 | No logging. |
| 1 | Major problems only. |
| 2 | Message collections and failed deliveries. |
| 3 | Successful deliveries. |
| 4 | Messages being deferred (due to a host being down, etc.). |
| 5 | Normal message queueups. |
| 6 | Unusual but benign incidents, e.g., trying to process a locked queue file. |
| 9 | Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts. |
| 12 | Several messages that are basically only of interest when debugging. |
| 16 | Verbose information regarding the queue. |

### 4.4. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

#### 4.4.1. To suid or not to suid?

*Sendmail* can safely be made setuid to root. At the point where it is about to *exec* (2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the **u** and **g** options). (This can be overridden by setting the **S** flag to the mailer for mailers that are trusted and must be called as root.)

However, this will cause mail processing to be accounted (using *sa* (8)) to root rather than to the user sending the mail.

#### 4.4.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

#### 4.4.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

```
newaliases
```

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

## 5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

### 5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

#### 5.1.1. R and S – rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by

other rewriting sets.

The syntax of these two commands are:

**S***n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

**R***lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

### 5.1.2. D – define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

**D***xval*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence $*x*.

### 5.1.3. C and F – define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

**C***cword1 word2...*
**F***cfile* [ *format* ]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

CHmonet ucbmonet

and

CHmonet
CHucbmonet

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf* (3) pattern that should produce a single string.

### 5.1.4. M – define mailer

Programs and interfaces to mailers are defined in this line. The format is:

**M***name*, { *field=value* }*

where *name* is the name of the mailer (used internally only) and the "*field=name*" pairs define attributes of the mailer. Fields are:

| | |
|---|---|
| Path | The pathname of the mailer |
| Flags | Special flags for this mailer |
| Sender | A rewriting set for sender addresses |
| Recipient | A rewriting set for recipient addresses |
| Argv | An argument vector to pass to this mailer |
| Eol | The end-of-line string for this mailer |
| Maxsize | The maximum message length to this mailer |

Only the first character of the field name is checked.

### 5.1.5. H – define header

The format of the header lines that *sendmail* inserts into the message are defined by the **H** line. The syntax of this line is:

**H[?***mflags***?]***hname***:** *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

### 5.1.6. O – set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

**O***ovalue*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

### 5.1.7. T – define trusted users

Trusted users are those users who are permitted to override the sender address using the **-f** flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

**T***user1 user2...*

There may be more than one of these lines.

### 5.1.8. P – precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

**P***name=num*

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

```
Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100
```

## 5.2. The Semantics

This section describes the semantics of the configuration file.

### 5.2.1. Special macros, conditionals

Macros are interpolated using the construct $\$x$, where $x$ is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail*:

e The SMTP entry message
j The "official" domain name for this site
l The format of the UNIX from line
n The name of the daemon (for error messages)
o The set of "operators" in addresses
q default format of sender address

The **$e** macro is printed out when SMTP starts up. The first word must be the **$j** macro. The **$j** macro should be in RFC821 format. The **$l** and **$n** macros can be considered constants except under terribly unusual circumstances. The **$o** macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the **$o** macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the **$q** macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the **$q** macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

a The origination date in Arpanet format
b The current date in Arpanet format
c The hop count
d The date in UNIX (ctime) format
f The sender (from) address
g The sender address relative to the recipient
h The recipient host
i The queue id
p Sendmail's pid
r Protocol used
s Sender's host name
t A numeric representation of the current time
u The recipient user
v The version number of sendmail
w The hostname of this site
x The full name of the sender
y The id of the sender's tty
z The home directory of the recipient

There are three types of dates that can be used. The **$a** and **$b** macros are in Arpanet format; **$a** is the time as extracted from the "Date:" line of the message (if there was one), and **$b** is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, **$a** is set to the current time also. The **$d** macro is equivalent to the **$a** macro in UNIX (ctime) format.

The **$f** macro is the id of the sender as originally determined; when mailing to a specific host the **$g** macro is set to the address of the sender *relative to the recipient*. For example, if I send to "bollard@matisse" from the machine "ucbarpa" the **$f** macro will be "eric" and the **$g** macro will be "eric@ucbarpa."

The **$x** macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail*. The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the **$h**, **$u**, and **$z** macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the **$@** and **$:** part of the rewriting rules, respectively.

The **$p** and **$t** macros are used to create unique strings (e.g., for the "Message-Id:" field). The **$i** macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The **$y** macro is set to the id of the terminal of the sender (if known); some systems like to put this in the Unix "From" line. The **$v** macro is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The **$w** macro is set to the name of this host if it can be determined. The **$c** field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the **-h** flag on the command line or by counting the timestamps in the message.

The **$r** and **$s** fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?x text1 $| text2 $.

This interpolates *text1* if the macro **$x** is set, and *text2* otherwise. The "else" (**$|**) clause may be omitted.

### 5.2.2. Special classes

The class **$=w** is set to be the set of all names this host is known by. This can be used to delete local hostnames.

### 5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

| | |
|---|---|
| **$*** | Match zero or more tokens |
| **$+** | Match one or more tokens |
| **$-** | Match exactly one token |
| **$=***x* | Match any token in class *x* |
| **$~***x* | Match any token not in class *x* |

If any of these match, they are assigned to the symbol **$***n* for replacement on the right hand side, where *n* is the index in the LHS. For example, if the LHS:

$-:$+

is applied to the input:

UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

| | |
|---|---|
| $1 | UCBARPA |
| $2 | eric |

### 5.2.4. The right hand side

When the right hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

| | |
|---|---|
| **$***n* | Substitute indefinite token *n* from LHS |
| **$>***n* | "Call" ruleset *n* |
| **$#***mailer* | Resolve to *mailer* |
| **$@***host* | Specify *host* |
| **$:***user* | Specify *user* |

The **$***n* syntax substitutes the corresponding value from a **$+**, **$-**, **$***, **$=**, or **$~** match on the LHS. It may be used anywhere.

The **$>***n* syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset *n*. The final value of ruleset *n* then becomes the substitution for this rule.

The **$#** syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

**$#***mailer***$@***host***$:***user*

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceded by a **$@** or a **$:** to control evaluation. A **$@** prefix causes the ruleset to return with the remainder of the RHS as the value. A **$:** prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The **$@** and **$:** prefixes may precede a **$>** spec; for example:

R$+ $:$>7$1

matches anything, passes that to ruleset seven, and continues; the **$:** is necessary to avoid an infinite loop.

### 5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the C flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by *sendmail* before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer*, *host*, *user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

### 5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Addendum C Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

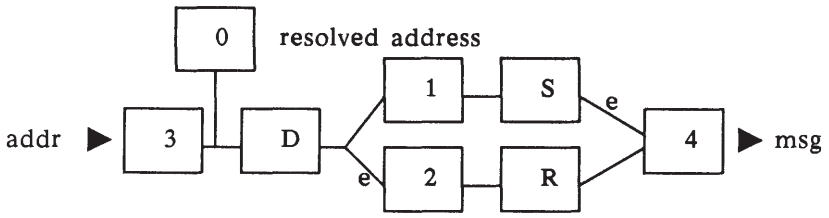



Figure 2 - Rewriting set semantics
D - sender domain addition
S - mailer-specific sender rewriting
R - mailer-specific recipient rewriting

#### 5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

```
$#error$:Host unknown in this domain
```

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

### 5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

#### 5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

#### 5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

##### 5.3.2.1. Large site, many hosts — minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transferred via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4* (1) or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

#### 5.3.2.2. Small site — complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the configuration remains relatively static, the update problem will probably not be too great.

#### 5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

### 5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included in */usr/doc/sendmail.dir* as *rfc819.lpr* and *rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

### 5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

### 5.3.5. Testing the rewriting rules – the -bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

```
sendmail -bt -Ctest.cf
```

which would read the configuration file "test.cf" and enter test mode. In this mode, you enter lines of the form:

```
rwset address
```

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

```
1,21,4 monet:bollard
```

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "-d21" flag to turn on more debugging. For example,

```
sendmail -bt -d21.99
```

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

### 5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a **-f** or **-r** flag respectively. These flags are only passed if they were passed to *sendmail*, so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "-f $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing **$u** will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[4].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa
To: wnj@monet, mckusick@ucbarpa

*if and only if* the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Addendum C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

[3] *Sendmail* must be running setuid to root for this to work.

[4] The "c" configuration option must be given for this to be effective.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC $h [ *port* ]

where *port* is the optional port number to connect to.

For example, the specifications:

Mlocal, P=/bin/mail, F=rlsm S=10, R=20, A=mail -d $u
Mether, P=[IPC], F=meC, S=11, R=21, A=IPC $h, M=100000

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky **-r** flag, does local delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a **-r** flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

# Addendum A

# COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

-f *addr* The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

-r *addr* An obsolete form of **-f**.

-h *cnt* Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.

-F*name* Sets the full name of this user to *name*.

-n Don't do aliasing or forwarding.

-t Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc." line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

-b*x* Set operation mode to *x*. Operation modes are:

| | |
|---|---|
| m | Deliver mail (default) |
| a | Run in arpanet mode (see below) |
| s | Speak SMTP on input side |
| d | Run as a daemon |
| t | Run in test mode |
| v | Just verify addresses, don't collect or deliver |
| i | Initialize the alias database |
| p | Print the mail queue |
| z | Freeze the configuration file |

The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

-q*time* Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

-C*file* Use a different configuration file.

-d*level* Set debugging level.

-o*xvalue* Set option *x* to the specified *value*. These options are described in Addendum B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the **-s** flag.

# Addendum B

# CONFIGURATION OPTIONS

The following options may be set using the **-o** flag on the command line or the **O** line in the configuration file:

A*file* — Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

a — If set, wait for an "@:@" entry to exist in the alias database before starting up. If it does not appear in five minutes, rebuild the database.

c — If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

d*x* — Deliver in mode *x*. Legal modes are:

- i Deliver interactively (synchronously)
- b Deliver in background (asynchronously)
- q Just queue the message (deliver during queue run)

D — If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **-bi**.

e*x* — Dispose of errors using mode *x*. The values for *x* are:

- p Print error messages (default)
- q No messages, just give exit status
- m Mail back errors
- w Write back errors (mail if user not logged in)
- e Mail back errors and give zero exit stat always

F*n* — The temporary file mode, in octal. 644 and 600 are good choices.

f — Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

g*n* — Set the default group id for mailers to run in to *n*.

H*file* — Specify the help file for SMTP.

i — Ignore dots in incoming messages.

L*n* — Set the default log level to *n*.

M*xvalue* — Set the macro *x* to *value*. This is intended only for use from the command line.

m — Send to me too, even if I am in an alias expansion.

o — Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir* — Use the named *dir* as the queue directory.

| | |
|---|---|
| r*time* | Timeout reads after *time* interval. |
| S*file* | Log statistics in the named *file*. |
| s | Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the client under any circumstances. |
| T*time* | Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender. |
| t*S,D* | Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six. |
| u*n* | Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user. |
| v | Run in verbose mode. |

# Addendum C

# MAILER FLAGS

The following flags may be set in the mailer description.

f The mailer wants a **-f** *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r Same as **f**, but sends a **-r** flag.

S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n Do not insert a UNIX-style "From" line on the front of the message.

l This mailer is local (i.e., final delivery will be performed).

s Strip quote characters off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F This mailer wants a "From:" header line.

D This mailer wants a "Date:" header line.

M This mailer wants a "Message-Id:" header line.

x This mailer wants a "Full-Name:" header line.

P This mailer wants a "Return-Path:" line.

u Upper case should be preserved in user names for this mailer.

h Upper case should be preserved in host names for this mailer.

A This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC821.

P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I This mailer will be speaking SMTP to another *sendmail* – as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain" clause

from the sender tacked on. This allows mail with headers of the form:

> From: usera@hosta
> To: userb@hostb, userc

to be rewritten as:

> From: usera@hosta
> To: userb@hostb, userc@hosta

automatically.

# Addendum D

# OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4 These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

### Parameters in md/config.m4

The following compilation flags may be defined in the m4CONFIG macro in *md/config.m4* to define the environment in which you are operating.

V6 If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork* (2) system call, special types defined in <sys/types.h> (e.g, u_char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

### Parameters in src/conf.h

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256] The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500] The maximum total length of any header field, including continuation lines.

MAXPV [40] The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30] When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the **-h** flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
: The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30] The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
: The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG If set, debugging information is compiled in. To actually get the debugging output, the **-d** flag must be used.

LOG If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

**Configuration in src/conf.c**

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below), The flags are:

H_ACHECK Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H_EOH If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H_FORCE Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H_TRACE If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H_RCPT If set, this field contains recipient addresses. This is used by the **-t** flag to determine who to send to when it is collecting recipients from the message.

H_FROM This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo              HdrInfo[] =
{
        /* originator fields, most to least significant  */
    "resent-sender",        H_FROM,
    "resent-from",          H_FROM,
    "sender",               H_FROM,
    "from",                 H_FROM,
    "full-name",            H_ACHECK,
        /* destination fields */
    "to",                   H_RCPT,
    "resent-to",            H_RCPT,
    "cc",                   H_RCPT,
        /* message identification and control */
    "message",              H_EOH,
    "text",                 H_EOH,
        /* trace fields */
    "received",             H_TRACE|H_FORCE,

    NULL,                   0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro **$x** and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char Arpa_Info[] =      "050";      /* arbitrary info */
char Arpa_TSyserr[] =   "455";      /* some (transient) system error */
char Arpa_PSyserr[] =   "554";      /* some (transient) system error */
char Arpa_Usrerr[] =    "554";      /* some (fatal) user error */
```

The class *Arpa_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa_TSyserr* and *Arpa_PSyserr* is printed by the *syserr* routine. TSyserr is

printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is m3.de based on the value of *errno*. Finally, *Arpa_Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
        register ADDRESS *to;
{
        if (MsgSize > 50000 && to->q_mailer != LocalMailer)
        {
                usrerr("Message too large for non-local delivery");
                NoReturn = TRUE;
                return (FALSE);
        }
        return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to suppress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

# Addendum E

# SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
: The binary of sendmail.

/usr/bin/newaliases
: A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **-bi** flag.

/usr/bin/mailq
: Prints a listing of the mail queue. This program is equivalent to using the **-bp** flag to *sendmail*.

/usr/lib/sendmail.cf
: The configuration file, in textual form.

/usr/lib/sendmail.fc
: The configuration file represented as a memory image.

/usr/lib/sendmail.hf
: The SMTP help file.

/usr/lib/sendmail.st
: A statistics file; need not be present.

/usr/lib/aliases
: The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
: The alias file in *dbm* (3) format.

/etc/syslog
: The program to do logging.

/etc/syslog.conf
: The configuration file for syslog.

/etc/syslog.pid
: Contains the process id of the currently running syslog.

/usr/spool/mqueue
: The directory in which the mail queue and temporary files reside.

/usr/spool/mqueue/qf*
: Control (queue) files for messages.

/usr/spool/mqueue/df*
: Data files.

/usr/spool/mqueue/lf*
: Lock files

/usr/spool/mqueue/tf*
: Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
: A file used when creating a unique id.

/usr/spool/mqueue/xf*
: A transcript of the current session.

# Appendix I: The Berkeley IPC Primer

## 1. INTRODUCTION

One of the most important parts of DOMAIN/IX *bsd4.2* is its interprocess communication (IPC) facilities. These facilities are the result of several years of research at the University of California at Berkeley. They incorporate many of the ideas from current research, while trying to maintain a philosophical compatbility with the UNIX system. This document is based on *A 4.2BSD Interprocess Communication Primer*, draft of July 9, 1984, by S. J. Leffler, R. S. Fabry, and W. N. Joy of the Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Prior to the advent of these IPC facilities, the only standard mechanism that allowed two processes to communicate was the pipe. Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the IPC facilities provided in 4.2BSD have been designed as a totally independent subsystem. The 4.2BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are pathnames) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities, they will be refined; only time will tell.

The remainder of this appendix is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

## 2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle

common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain, sockets are named with UNIX pathnames; e.g., a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2BSD IPC supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

### 2.1. Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes †.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram-oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in Section 5.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are currently unimplemented in 4.2BSD. As such, in this document we will concern ourselves only with the three socket types for which support exists.

† At SR9.0, DOMAIN/IX does not implement UNIX domain sockets.

### 2.2. Socket Creation

To create a socket, the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain, the constant is AF_UNIX †; for the Internet domain, AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use, a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol, one selects the protocol number, as defined within the communication domain. For the Internet domain, the available protocols are defined in *<netinet/in.h>* or, better yet, one may use one of the library routines discussed in Section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

### 2.3. Binding Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

---

† The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain, names are pathnames; in the Internet domain, names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

```
bind(s, "/dev/foo", sizeof ("/dev/foo") - 1);
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

### 2.4. Connection Establishment

With a bound socket, it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side, the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket (see Section 5.4.). An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2BSD, this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned "based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times, the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. It is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

The accept function normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call, there are alternatives; they will be considered in Section 5.

### 2.5. Data Transfer

With a connection established, data may begin to flow. There are a number of calls to send and receive data. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to read and write, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

| | |
|---|---|
| SOF_OOB | send/receive out of band data |
| SOF_PREVIEW | look at data without reading |
| SOF_DONTROUTE | send data without routing packets |

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

### 2.6. Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

### 2.7 Connectionless Sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance, when a network is unreachable), the call will return -1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multi-casting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

### 2.8. Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex I/O requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

*Select* takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The *nfds* parameter specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely †. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of -1 is returned along with the error number EINTR.

*Select* provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in Section 5.

## 3. NETWORK LIBRARY ROUTINES

The discussion in Section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task, a number of routines have been added to the standard C run-time library. In this section, we will consider the new routines provided to manipulate network addresses. While the 4.2BSD networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address databases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g., "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is undesirable for a network to require that hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location-independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

---

† To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

### 3.1. Host Names

A host name to address mapping is represented by the *hostent* structure:

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostybyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network, the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

### 3.2. Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 *  Assumption here is that a network number
 *  fits in 32 bits — probably a poor one.
 */
struct netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

### 3.3. Protocol Names

For protocols, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol # */
};
```

### 3.4. Service Names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port # */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

sp = getservbyname("telnet", (char *)0);

returns the service specification for a telnet server using any protocol, while the call

sp = getservbyname("telnet", "tcp",);

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

### 3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network-independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets, there will always be some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit (5);
        }
        ...
}
```

**Figure 1. Remote login client code.**

This example will be considered in more detail in Section 4.

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related database routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

| Call | Synopsis |
|---|---|
| bcmp(s1, s2, n) | compare byte-strings; 0 if same, not 0 otherwise |
| bcopy(s1, s2, n) | copy n bytes from s1 to s2 |
| bzero(base, n) | zero-fill n bytes starting at base |
| htonl(val) | convert 32-bit quantity from host to network byte order |
| htons(val) | convert 16-bit quantity from host to network byte order |
| ntohl(val) | convert 32-bit quantity from network to host byte order |
| ntohs(val) | convert 16-bit quantity from network to host byte order |

**Table 1. C Run-Time routines.**

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies that users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out, the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX, these routines are defined as null macros.

## 4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in Section 2. In this section, we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well Known address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous. The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

### 4.1. Servers

In 4.2BSD, most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time, it advertises it services by listening at a well known location. For example, the remote login server's main loop is of the form shown in Figure 2.

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        << disassociate server from controlling terminal >>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

**Figure 2. Remote login server.**

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to ensure that the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in Section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

### 4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next, the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit (4);
}
```

The details of the remote login protocol will not be considered here.

### 4.3. Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa        up   9:45,      5 users, load  1.15,  1.39,  1.31
cad         up   2+12:04,   8 users, load  4.67,  5.13,  4.59
calder      up   10:10,     0 users, load  0.27,  0.15,  0.14
dali        up   2+06:28,   9 users, load  1.04,  1.20,  1.65
degas       up   25+09:48,  0 users, load  1.49,  1.43,  1.41
ear         up   5+00:05,   0 users, load  1.51,  1.54,  1.56
ernie     down   0:24
esvax     down   17:04
ingres    down   0:26
kim         up   3+09:16,   8 users, load  2.03,  2.46,  3.11
matisse     up   3+06:18,   0 users, load  0.03,  0.03,  0.05
medea       up   3+09:39,   2 users, load  0.35,  0.37,  0.50
merlin    down   19+15:37
miro        up   1+07:20,   7 users, load  4.59,  3.28,  2.12
monet       up   1+00:43,   2 users, load  0.22,  0.09,  0.07
oz        down   16:09
statvax     up   2+15:57,   3 users, load  1.52,  1.81,  1.86
ucbvax      up   9:34,      2 users, load  6.08,  5.16,  3.28
```

**Figure 3. Ruptime Output.**

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to ensure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume that the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror(" rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, FWRONLY|FCREATE|FTRUNCATE, 0666);
                }
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

**Figure 4. Rwho Server.**

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. The necessity of deciding where to transmit the resultant packet

does, however, indicate some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast, another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information †.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information‡. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

## 5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

### 5.1. Out of Band Data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the

† One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself.

‡ An example of such a system call is the gethostname(2) call which returns the host's "official" name.

remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e., the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message, the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

**Figure 5. Flushing Terminal I/O on Receipt of Out of Band Data.**

### 5.2. Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals, each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

### 5.3. Pseudo Terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal*. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

### 5.4. Internet Address Binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association*. An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding, the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but

leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (i.e., the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range, the following code is used by the remote shell server:

```
struct sockaddr_in sin;
 ...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
...
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a

"secure" environment to perform authentication based on the originating address and port number.

In certain cases, the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two-step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure that any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding may be somewhat haphazard when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected †.

### 5.5. Broadcasting and Datagram Sockets

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

† For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the sender's address and port (datagram sockets are anchored before a message is allowed to go out).

### 5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven I/O" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();

 ...
sigset(SIGCHLD, reaper);
listen(f, 10);
for(;;){
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0); if (g < 0) {
        if (errno != EINTR)
                perror("rlogind: accept");
        continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;

}
```

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

Appendix J

# Configuring TCP/IP for DOMAIN/IX *bsd4.2*

# Contents

## Part 1 - Defining the TCP/IP Configuration



## Part 2 - Configuring TCP/IP

# Part 1 - Defining the TCP/IP Configuration

Before you can use TCP/IP, you must decide a number of configuration issues that affect how the TCP/IP software establishes connections between hosts. You will have to define both addressing information, which defines the names, addresses, and physical interfaces for the hosts, networks, and gateways, and process information, which specifies the processes that you run to support TCP/IP communications and the nodes on which these processes execute.

This part of Appendix J defines this information, and explains its functions. It also defines steps you should take *before* you install TCP/IP that will make the configuration procedure easier.

# Names and Addresses

Whenever you refer to a TCP/IP host, you usually use an easy to remember name, such as the node name //DIONYSUS. The operating system converts this name internally to an address value that is meaningful to it, for example the DOMAIN address 03a2cl76.06d49.

Within a DOMAIN network, this mapping between names and addresses is sufficient to get messages from one point to another. However, when you connect different local networks, such as DOMAIN and ETHERNET, each network's local addresses are meaningless to the next. Therefore, another addressing layer is required; in the case of connecting to an ETHERNET, this is the Internet addressing layer.

While local addresses need only be unique on the local network, Internet addresses must be unique across all connected networks. For example, let's give the node //DIONYSUS the Internet address 197.9.8.3.

TCP/IP also uses mnemonic names. To keep things simple, the local name is usually the Internet name. While DOMAIN/IX TCP/IP does not require that the node name and Internet name be the same, we recommend that you follow this convention for the sake of simplicity.

## Gateway Names and Addresses

While a host has one local name and address and one Internet name and address, a gateway must have more than one address, because it sits on *two* local networks and must be fully identified on both. For example, the DOMAIN node //JANUS is a gateway to an ETHERNET network. It has:

- The DOMAIN name //JANUS
- The DOMAIN address 03a2cl76.06a3
- The Internet address on the DOMAIN ring 197.9.8.1
- The Internet name JANUS
- The Internet address on the ETHERNET LAN 197.10.9.1
- The ETHERNET address 1.1.0.3.2.3

Figure 1-1 shows the network example that we have been using. It shows a DOMAIN ring and ETHERNET LAN connected to make an Internet. It illustrates the names and addresses for a DOMAIN host, gateway, and an ETHERNET host.

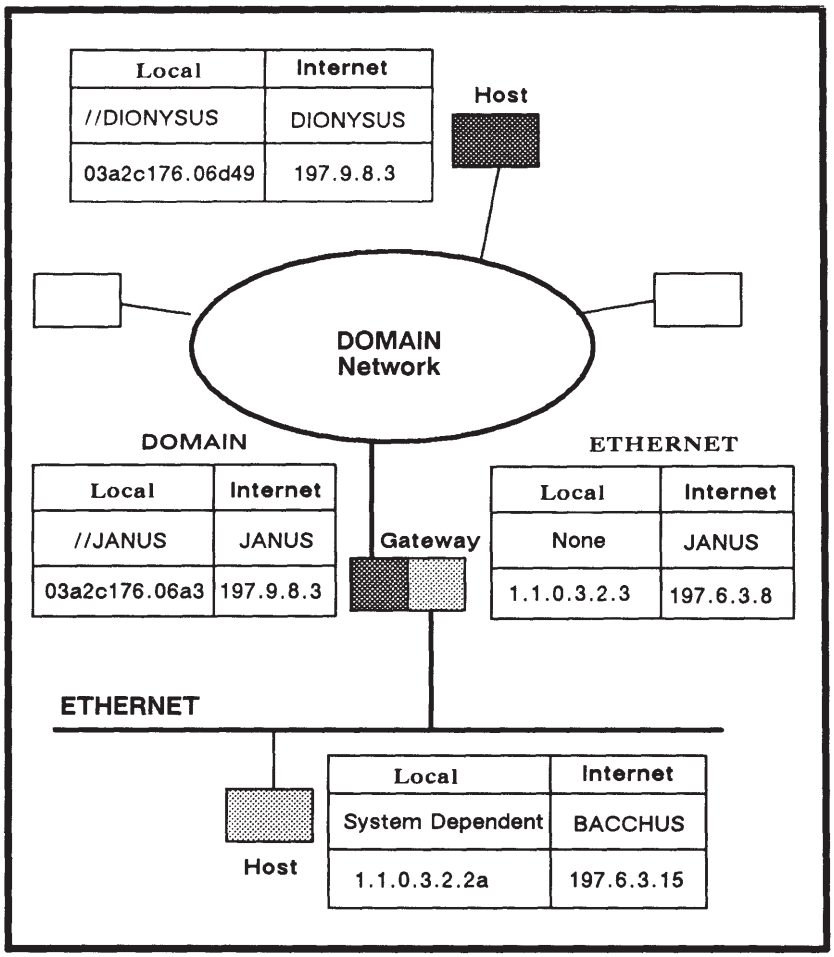



*Figure 1-1. Relationships Among Names and Addresses*

## Names

Since people generally prefer to use names rather than Internet addresses, TCP/IP requires you to associate an ASCII name with each host and gateway with which you communicate. While TCP/IP does not require you to do so, people usually use the node or computer names used in their local network as the TCP/IP names. For example, if your node entry name is //JANUS, you'd name the host JANUS.

The Internet model allows you to use more than one name for a host or gateway. The additional names are called aliases. Therefore, you could give //JANUS not just the Internet name JANUS, but also the Internet aliases ODIN and THOR.

Internet names for a gateway do not depend upon the local network. Therefore, if JANUS is a gateway between a DOMAIN ring and an ETHERNET LAN, you can use JANUS, ODIN, or THOR from hosts on either network.

Local names depend upon the operating system that runs on the host. Because of the design of the DOMAIN system, you use DOMAIN names on a ring-wide basis. This is not necessarily true on the ETHERNET LAN, where one host may be a VAX running UNIX and another host may run a completely different operating system.

## Local Network Addresses

A DOMAIN address consists of one or two parts, depending upon whether you have a multiple-ring DOMAIN network.

If your installation consists of a single ring, your DOMAIN address may consist of a 20-bit **Node ID**. This number is firmware-encoded for each DOMAIN node and always appears as a hexadecimal number with up to five digits. For example, the node ID for //JANUS is 006a3. You can determine any node's ID by using the NETSTAT Shell command.

If your installation uses the DOMAIN/BRIDGE to connect multiple rings, a **Network number** precedes the Node ID and is separated from the node ID by a period. The network number is an up-to eight-digit hexadecimal number that identifies the host's ring. Network numbers are assigned by the Apollo Response Center. In our example, //JANUS is on a ring with a network number of 03a2cl76. Therefore, //JANUS' full DOMAIN address is 03a2cl76.6a3.

An ETHERNET address is a 48-bit number. Three bytes are reserved for the manufacturer's identifiers. Three bytes are assigned by the XEROX Corporation.

## Internet Addresses

Each Internet address is 32 bits long, and contains two variable-length fields that identify the local network and the host within that network. The network number, the left field, identifies the local network to the Internet. The host number, the right field, identifies the host within the local network.

You normally represent Internet addresses in **Internet format**. In the Internet format, you specify addresses as follows:

```
W.X.Y.Z
```

where W, X, Y and Z are decimal numbers between 0 and 255. Each of the decimal numbers represents one byte in the Internet address.

Internet addresses have variable-length fields for the network numbers and host numbers. You choose a length for your network numbers by choosing Type A, B, or C Internet addresses. Figure 1-2 illustrates the differences. It also shows how the most significant bits (MSB) in each network number serve as tag bits to identify the address type as A (MSB of 0), B (MSB of 10) or C (MSB of 110).

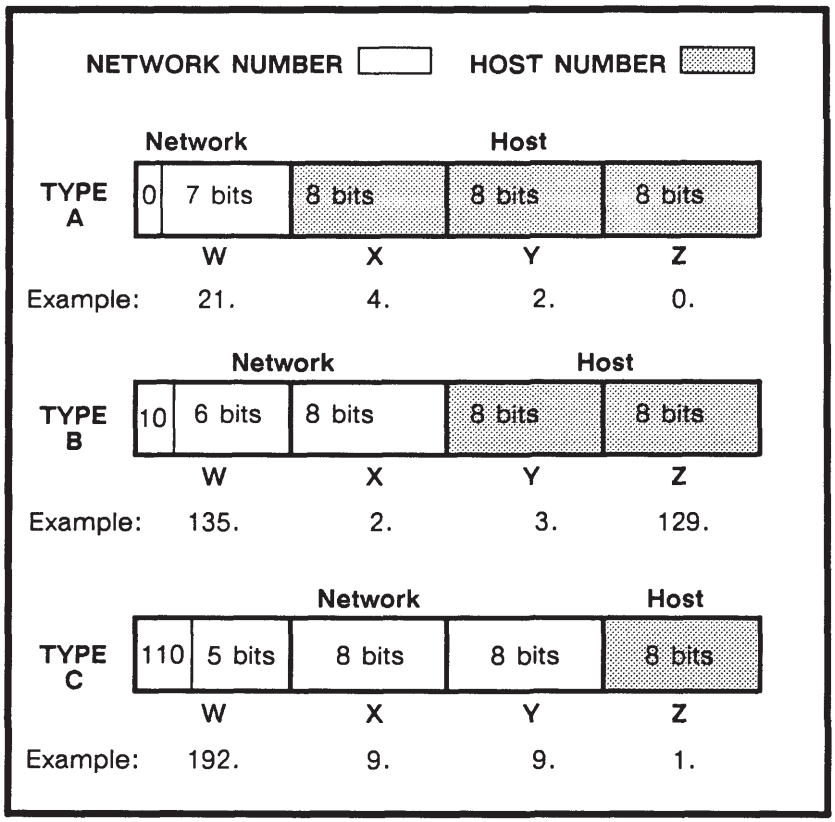



*Figure 1-2. Type A, B, and C Internet Addresses*

For example, the Type A address in the figure has a network number of 21. Contrast this with the network number in the Type B address (135.2) and in the Type C address (192.9.9). In the Type C address, you may only assign host numbers 0 - 255, whereas in the Type A address, you can assign host numbers 0.0.0 through 255.255.255. We discuss considerations for assigning Internet addresses later in this appendix.

# Physical Layer Interface Descriptions

The Internet model's layered approach allows communication between networks with varied Physical Layer protocols. The hardware that supports DOMAIN TCP/IP-based communications translates protocols between the ETHERNET LAN and the DOMAIN ring. You must indicate to TCP/IP the **physical interface** used by each host and by the gateway.

For hosts in the DOMAIN ring, the interface is the DOMAIN ring. The DOMAIN ring always has physical interface identifier of **dr0**. Gateways have two physical interfaces; the DOMAIN ring and the ETHERNET LAN. These are identified as **dr0** and **il0**, respectively.

A DOMAIN node that is a router for the DOMAIN BRIDGE is treated as a TCP/IP gateway node. Its physical interfaces are **dr0**, for the DOMAIN ring on which it resides, and **dr1**, for the DOMAIN ring to which it is connecting via TCP/IP. Like any other DOMAIN TCP/IP gateway, a DOMAIN BRIDGE router must run the **tcp_server** process and **routed**.

# Daemons, Servers, and Helpers

DOMAIN/IX TCP/IP-based communications software uses several processes that support communications in various ways. These processes respond to requests for some form of service. They are generally called **servers** or **daemons**. Table 1-1 lists these processes, describes their purposes, and indicates the nodes that require them.

**Table 1-1. DOMAIN/IX TCP/IP Server Processes**

| Name | Purpose | Location |
|---|---|---|
| tcp_server | Enables TCP/IP communications on the node | All TCP/IP hosts |
| routed | Manages gateway network routing tables | One per gateway node |
| rwhod | System status server, maintains database used by **rwho** and **ruptime** | One per gateway node |
| sendmail | Handles mail received over the Internet | One per network |
| tftpd | Enables TFTP access to the host | Hosts that accept TFTP connections |
| inetd | Starts the following daemons as needed: (in */etc/inetd.conf*) | All BSD4.2 nodes |
| ftpd | Enables direct **ftp** access to the host | Hosts that accept **ftp** connections |
| telnetd | Enables inbound **telnet** access to the host | Hosts that accept inbound **telnet** |
| rexecd | Enables remote execution of commands on this node | Hosts that accept **rexec** routine |
| rlogin | Enables remote login to this node | Hosts that accept **rlogin** program |
| rshd | Enables remote execution of commands on this node with user authentication | Hosts that accept **rsh** program |

## Starting Server Processes

As a general rule, these processes should execute whenever the node is running; therefore you include commands to start them in startup files. The files you use depend upon the processes.

### Configuring Servers and Helpers

You start some of the server processes in the node startup file. The name and location of this file depends upon the type of node that you are using. If you are configuring a disked node, use the */sys/node_data/startup[.node_type]* file. If you are configuring a diskless node, use the *//partner_node/sys/node_data.nodeid/startup [.node_type]* file. In these pathnames *.node_type* indicates the model node you are using; *.nodeid* is the node identification number. *Administering your DOMAIN System* describes node startup files in detail.

You use the node startup file to specify the following process:

- tcp_server

To specify the tcp_server, uncomment or enter the following line:

```
cps /sys/tcp/tcp_server -n tcp_server
```

### Configuring the BSD4.2 Daemons

**NOTE:** In most installations the */etc* directory is located on a single administrative node, and all other nodes use links to access this directory. In such cases, */etc/rc* on the administrative node *must* be a link to the file *'node_data/etc.rc*, and /etc/inetd.conf must be a link to *'node_data/etc.inetd.conf*. The *'node_data* is automatically interpreted as either */sys/node_data* or *//partner_node/sys/node_data.nodeid* of the node that is accessing the file.

The BSD4.2 daemons are started by the **/etc/run_rc** program, which starts the daemons specified in the */etc/rc* file. You start **/etc/run_rc** at login time, in the node's startup file. Put the following line in either the */sys/node_data/startup[.node_type]* or *//partner_node/sys/node_data.nodeid/startup[.node_type]* file.

```
cps /etc/run_rc
```

Use the */etc/rc* file to specify any of the following processes:

- **inetd**
- **routed**
- **rwhod**
- **sendmail**
- **tftpd**

Use the */etc/inetd.conf* file to specify the following processes. (You must, of course, also specify **inetd** in the */etc/rc* file.)

- **ftpd**
- **rexecd**
- **rlogind**
- **rshd**

- **telnetd**

Specify each required daemon by uncommenting the lines in the file that start the process. For example, to specify **inetd**, uncomment (remove the # characters from the beginning of) the following lines in */etc/rc*:

```
if [ -f  /etc/inetd ]; then
        /etc/inetd &
fi
```

## The TCP_SERVER

The **tcp_server** process ensures that all TCP/IP data are reliably transmitted between the end-user processes such as **ftp** and **telnet**. It also performs routing services, and maintains mapping tables that relate Internet addresses to local addresses.

**Rules:** The **tcp_server** must run on each node that uses any form of TCP/IP based communications.

## BSD4.2 Daemons

You must run the BSD4.2 daemons to enable various BSD4.2 commands and utilities. The following sections describe these daemons and indicate the nodes where they should execute. See the *DOMAIN/IX Programmer's Reference for BSD4.2* for detailed descriptions of each daemon.

### routed

The **routed** daemon maintains the gateway network routing tables. It maintains an internal database of gateways that are directly accessible from the local gateway. It also broadcasts its routing tables every 30 seconds. This dynamic operation eliminates the need to update static gateway tables each time the network configuration changes.

If **routed** does not execute on the gateway, the gateway does not transmit packets with routing information to the ETHERNET. Because **routed** purges its database of gateway entries for gateways that have not sent a network information packet within three minutes, remote BSD4.2 gateways that run **routed** may lose knowledge of any DOMAIN gateway that does not use **routed**.

**Rules:** **routed** *must* execute on each gateway.

**routed** is not required if you only use BSD4.2 TCP/IP communications on the DOMAIN network.

## rwhod

The **rwhod** daemon is the Internet system status server. It maintains the database of status information that is used by the **rwho**(1) and **ruptime**(1) programs.

**Rules:** **rwhod** should execute on each gateway. In this case, it can provide information on both the DOMAIN network and the ETHERNET.

### sendmail

The **sendmail** program routes mail messages that you send using BSD4.2 or DARPA mail commands over the Internet. When it runs as a daemon, it enables the DOMAIN network to send and receive **mail** messages from the ETHERNET. **Sendmail** *is not* a user-level interface.

**Rules:** If you use mail between the DOMAIN network and any other network, **sendmail** must run as a daemon on one node on your network. You execute **sendmail** as a daemon by including the **-bd** flag in the **sendmail** command.

### tftpd

The **tftpd** daemon is a server that supports the DARPA Trivial File Transfer Protocol. It listens for and accepts **tftp** requests. You can establish a **tftp** request *only* to a node that runs the **tftpd**. That is, you must specify a host that runs **tftpd** in the Shell tftp command. However, the **tftpd** daemon provides access to files on all nodes on the DOMAIN network. You do *not* need to run **tftpd** to issue the **tftp** command.

**Rules:** You should have one or more **tftpd** processes per DOMAIN network if you wish to support the TFTP protocol.

### inetd

The **inetd** is a server-manager that invokes Internet services such as **ftpd** or **rlogind** as necessary. Since it is a single process, **inetd** can efficiently manage many types of Internet connections

You *must* use inetd to invoke the following servers and daemons:

- **ftpd**
- **rexecd**
- **rlogind**
- **rshd**
- **telnetd**

**Rules:** You must have an **inetd** process on each node that requires **ftpd**, **rexecd**, **rlogind**, **rshd**, or **telnetd**.

The file **/etc/inetd.conf** specifies the daemons that **inetd** invokes. You only include those daemons in this file for the services that are supported by the server node. A template file, located in */sys/node_data/etc.inetd.conf*, is automatically installed with DOMAIN/IX BSD4.2. You include daemons by removing the comment marks (#) at the beginning of the lines that specify the required daemons.

### ftpd

The **ftpd** daemon accepts **ftp** connections and services **ftp** requests. You can establish an **ftp** connection only to a node that can run the **ftpd**. That is, you must specify a node that can run **ftpd** in the Shell **ftp** command or in response to the **ftp Host:** prompt. However, the **ftpd** provides access to files on all nodes on the DOMAIN network. You do *not* need the **ftpd** to issue the **ftp** command.

**Rules:** You should have one or more **ftpd** processes per DOMAIN network.

You must use the **inetd** daemon to invoke the **ftpd**.

**ftpd** and the DOMAIN FTP_SERVER cannot execute on the same node.

### rexecd

The **rexecd** daemon services requests from the **rexec**(3X) library function. It allows you to remotely execute UNIX commands on the server node. **Rexecd** must receive a valid user id and password from **rexec**.

**Rules:** **rexecd** must execute on each node that supports invocation of commands from a remote host that uses **rexec**.

You must use the **inetd** daemon to invoke the **rexecd**.

### rlogind

The **rlogind** daemon services requests from the **rlogin**(1) program. It allows you to log in remotely on the server node. **Rlogind** requires pseudo-ttys. These are normally created when you install DOMAIN/IX, but can be created by the **crpty**(8) command. **Rlogind** does not request a password, if the remote host is listed in the server's */etc/hosts.equiv* file.

**Rules:** **rlogind** must execute on each node that supports login from a remote host using **rlogin**.

You must use the **inetd** daemon to invoke the **rlogind**.

### rshd

The **rshd** daemon is the remote shell server. It services requests from the **rsh** program and **rcmd** library function. It allows you to remotely execute UNIX commands on the server node. **Rshd** does not request a password, if the remote host is listed in the server's */etc/hosts.equiv* file.

**Rules:** **rshd** must execute on each node that supports invocation of commands from remote hosts using **rsh** or **rcmd**.

You must use the **inetd** daemon to invoke the **rshd**.

### telnetd

The **telnetd** daemon accepts **telnet** connections. You must run the **telnetd** on each node that accepts inbound **telnet** sessions. You do *not* need the **telnetd** to issue the **telnet** command.

**Rules:** **telnetd** must execute on each node that accepts remote login using **telnet**.

You must use the **inetd** daemon to invoke the **telnetd**.

**telnetd** and the DOMAIN TELNET_SERVER can not execute on the same node.

# Service Nodes

Before we can discuss TCP/IP Mapping information files, we must introduce an additional concept that is *not* part of standard TCP/IP, but reflects the distributed nature of the DOMAIN system. While TCP/IP communications distinguishes between gateways and hosts, we add a third type of node, the service node, for the purposes of configuration.

Defined broadly, a service node is any node that provides some form of support service to other users. A service node does not necessarily have to use TCP/IP communications itself. Conversely, a service node

can also be either a host or a gateway. The TCP/IP service node is that node which contains the TCP/IP name and address mapping files.

The DOMAIN/IX administrative node, which contains the /*etc* directory, is also considered a service node, since the BSD4.2 mapping files are located in that directory. The TCP/IP service node and the DOMAIN/IX administrative node may be the same machine.

# TCP/IP Mapping Information Files

You provide the information about a host or gateway's names, addresses, and physical interfaces, and about the relations among them during the configuration process by editing several files. However, before you edit the files you should understand the contents of each file. This section describes each file, and the following sections will help you prepare for configuration. Table 1-2 lists the files that you edit and indicates their functions and locations.

Note that there are two files ( /*etc*/*networks* and /*etc*/*hosts* ) that you must manage *only* if you are using DOMAIN/IX BSD4.2 TCP/IP on a DOMAIN network that does *not* communicate with any other networks. These files are automatically generated (by **makehost.sh**) if you are using BSD4.2 and communicating across a gateway to another Internet network, but if you are using BSD4.2 TCP/IP on a DOMAIN network only, you must edit them manually.

**Table 1-2. TCP/IP Information Files that you Edit**

| Name | Purpose | Location |
|---|---|---|
| *On all Hosts* | | |
| *thishost* | Lists the Internet name of the local host. | */sys/node_data [node.id]/thishost* |
| *networks* | Lists the Internet addresses and corresponding physical interfaces for the local host. | */sys/node_data [node.id] /networks* |
| *On the Service Node* | | |
| *local.txt* | Contains information on locally defined Internet addresses. | */sys/tcp/hostmap/ local.txt* |
| *hosts.txt* | Contains information on Internet addresses defined by the Network Information Center (NIC). | */sys/tcp/hostmap/ hosts.txt* |
| *On the Gateway Node* | | |
| *host_addr* | Associates Internet and local addresses of non-DOMAIN hosts that do not use the Address Resolution Protocol (ARP). | */sys/tcp/host_addr* |
| *On the BSD4.2 Administrative Node* | | |
| *hosts.equiv* | Contains the names of all hosts that you can access using **rlogin**, **rsh** and **rcp** without password authentication. | */etc/hosts.equiv* |
| *On the BSD4.2 Administrative Node if There is No Gateway* | | |
| *networks* | Contains the Internet network number of the DOMAIN ring. This file is different from the */sys/node_data[node.id]/networks* file. | */etc/networks* |
| *hosts* | Contains the names and Internet addresses of all hosts on the DOMAIN ring | */etc/hosts* |

## Links and File Locations

The service node files and the BSD4.2 configuration files usually are located on only one or a few nodes on the DOMAIN network. You access them through links from each host and gateway. This technique limits the replication of information, and therefore the number of changes you must make when the network configuration changes. It also provides a measure of security.

When you install DOMAIN/IX BSD4.2 or TCP/IP on a host node the install procedure creates the required links between your node and the TCP/IP service node and BSD4.2 administrative node. However, you should be familiar with these links, so we list them in Table 1-3.

**Table 1-3. TCP/IP Links**

| **Pathname on Host** | **Links to:** |
|---|---|
| On all Hosts | |
| /*sys*/*tcp*/*hostmap* | //*service_node*/*sys*/*tcp*/*hostmap* (directory) |
| /*sys*/*tcp*/*gateways* | //*service_node*/*sys*/*tcp*/*gateways* |
| /*sys*/*tcp*/*hosts.hst* | //*service_node*/*sys*/*tcp*/*hosts.hst* |
| /*etc (directory)* | //*administrative_node*/*etc* |
| On BSD4.2 Administrative Nodes | |
| /*etc*/*rc* | *'node_data*/*etc.rc* |
| /*etc*/*inetd.conf* | *'node_data*/*etc.inetd.conf* |

## Some Notes on Pathnames

The following notes describe how we refer to certain files in this manual.

- In this manual we refer to pathnames in the /*sys* directory. This is a relative pathname, and is accurate only if you are working at the node you are configuring. If you are manipulating the /*sys* directory of a remote node you *must* use an absolute pathname. For a disked node this is //*nodename*/*sys*; for a diskless node it is //*partner_node*/*sys* (where *partner_node* is the node name of the diskless node's partner).

- Files in the /*sys*/*node_data* directory are specific to the node on which the directory is located. Therefore, the corresponding files for a diskless node are located in a special directory, //*partner_node*/*sys*/*node_data.nodeid*, where *nodeid* is the node number of the diskless node. We refer to these directories by using the convention /*sys*/*node_data[.nodeid]*

- The pathname *'node_data* is a relative pathname. It always refers to the node_data directory of the node *that is making the reference*.

## THISHOST File

The */sys/node_data[.nodeid]/thishost* file defines the Internet *name* of the local host. You must have a *thishost* file on each node that is a TCP/IP host, including the gateway. The file consists of the host's Internet name on a single line. For example, the *thishost* file for JANUS is simply:

```
janus
```

## NETWORKS File

The *networks* file defines the *Internet addresses* and the *physical interface name* of the local host. On disked nodes, it is located in */sys/node_data/networks*. On diskless nodes it is located in */sys/node_data.nodeid/networks*, where *nodeid* is the node number of the diskless node. You must have a networks file for each node that is a TCP/IP host, including the gateways. The *networks* file format is:

**internet_address ON physical_interface [; comment]**

A host node's *networks* file consists of a single line with the host's Internet address on the DOMAIN ring and the physical interface identifier **dr0**. A gateway's *networks* file also must have the gateway's Internet address on the ETHERNET, with the interface identifier **il0**. For example, the *networks* file for the gateway //JANUS looks like:

```
197.9.8.1 on dr0
197.10.9.1 on il0
```

The physical interface for the second DOMAIN network that a DOMAIN BRIDGE router is connected to is represented as **dr1**. If //JANUS were a DOMAIN BRIDGE router, its *networks* file would look like this:

```
197.9.8.1 on dr0
X.X.X.X   on dr1
```

where X.X.X.X was the Internet address for JANUS on the other DOMAIN ring.

## Service Node Configuration Files

### The *hosts.txt* File

The */sys/tcp/hostmap/hosts.txt* file is the official Department of Defense Internet Host table from the Network Information Center (NIC). This file contains the names and addresses of all the hosts on the ARPANET, as well as many other networks in the Internet. You must have a copy of this file on your service node if you will communicate over ARPANET or any other network that is listed by the NIC.

If do *not* plan to connect your DOMAIN ring network to the DoD Internet, you should replace *hosts.txt* with an empty file or rename the file. Doing so will make the Shell script **makehost.sh** run faster.

We include a copy of *hosts.txt* with the TCP/IP software. However, the copy we send may not be the most current copy of the *hosts.txt* file. We suggest that you use the copy we ship you initially to create your host table, and then retrieve the latest version by executing the **gettable**(8) program. See the *DOMAIN/IX Programmer's Reference for BSD4.2* for a detailed description of this utility.

### The *local.txt* File

The */sys/tcp/hostmap/local.txt* file contains network information that is not provided by the Network Information Center in the *hosts.txt* file. You should use this file, and not *hosts.txt* if you assign your own network number and addresses. You usually do this if your hosts do not use any network listed by the Network Information Center. You should also add to this file the names and Internet addresses of new computers and networks that are not yet in The NIC-supplied version of *hosts.txt*.

To add these names and Internet addresses, edit the version on the service node. You'll keep only one version of the *local.txt* and *hosts.txt* file in your network, on the service node. Use the following information when you edit the *local.txt* file.

**NOTE:** The format of the *local.txt* file is the same as the format for the *hosts.txt* file. The format of these files is defined in RFC 810, "DoD Internet Host Table Specification" and is available online at NIC as the file:

[SRI-NIC] <NETINFO>RFC810.TXT

You can retrieve this file through FTP using username ANONYMOUS with any password. See *Using FTP and Telnet* for information about FTP.

The *local.txt* file has three categories of entries called **NET**, **GATEWAY**, and **HOST**.

- The **NET** entries define the networks that you can access. You list the network number and a name for each network linked in your TCP/IP implementation. For DOMAIN TCP/IP implementations, you'll be listing a DOMAIN network (or, if you use DOMAIN/BRIDGE connections, a DOMAIN Internet) and an ETHERNET LAN.
- The **GATEWAY** entries specify the gateways between the networks that you can access. In this entry or entries, you list both Internet addresses of each gateway node, its name, and certain other information.
- The **HOST** entries specify the TCP/IP hosts that you can access. In these entries, you list each host, its name, and other information. You have as many HOST entries as there are hosts that you wish to access, including remote hosts. (If you know the remote hosts are listed in *hosts.txt*, you need not add them to *local.txt*; however, it's fine to have entries listed in both files.)

```
; NET : NET-ADDR : NETNAME :
; GATEWAY : ADDR, ADDR : NAME : CPUTYPE : OPSYS : PROTOCOLS:
; HOST : ADDR, ALTERNATE-ADDR (if any): HOSTNAME,NICKNAME:
; CPUTYPE : OPSYS : PROTOCOLS :
;
NET : 197.6.3.0 : OFFICE-ETHER :
NET : 197.9.8.0 : DOMAIN-RING :
GATEWAY : 197.9.8.1,197.6.3.8 : JANUS, APOLLO : DN460 : AEGIS : IP/GW, GW/DUMB :
; The following are remote hosts on the ETHERNET
HOST : 197.6.3.15 : BACCHUS : VAX/11-750 : VMS : TCP/TELNET, TCP/FTP :
HOST : 197.6.3.18 : ODIN, WOTAN : VAX/8650 : UNIX : TCP/TELNET, TCP/FTP :
HOST : 197.6.3.22 : TESTBED : VAX/11-785 : UNIX :TCP/TELNET, TCP/FTP :
; The next host entry is for the gateway node. (The gateway
; needs a host entry, also, since it's also a host.)
HOST : 197.9.8.1, 197.6.3.8 : JANUS, : DN460 : DOMAIN : TCP/TELNET,TCP/FTP :
; The rest of the file lists hosts on the DOMAIN network.
HOST : 197.9.8.3 : DIONYSUS : DN550 : DOMAIN/IX : TCP/TELNET, TCP/FTP:
HOST : 197.9.8.5 : PAN : DN460 : DOMAIN/IX : TCP/TELNET,TCP/FTP :
HOST : 197.9.8.8 : ATHENA : DN300 : DOMAIN/IX : TCP/TELNET, TCP/FTP:
HOST : 197.9.8.9 : APOLLO : DN460 : DOMAIN/IX : TCP/TELNET, TCP/FTP:
```

*Figure 1-4. A local.txt File*

Each entry consists of the keyword **NET**, **GATEWAY**, or **HOST** followed by from two to five fields. You should start the *local.txt* file with all NET entries, followed by all GATEWAY entries, and then all HOST entries.

The following rules apply to each entry:

- A semicolon (:) terminates each field.
- Double colons (::) indicate a null field.
- A comma (,) separates values within a field
- A semicolon (;) begins a comment. Any text on a given line following the semicolon is not part of the host table. A comment can follow an entry on the same line.
- Spaces are optional before and after commas and colons.

Each **NET** entry in the *local.txt* file has the following format:

**NET : net-addr : netname :**

Where:

**net-addr** is the network's Internet network number, followed by a 0; for example 197.9.8.0

**netname** is the name of the network; for example OFFICE-ETHER.

Each **GATEWAY** entry in the *local.txt* file has the following format:

**GATEWAY : addr1,addr2 : name : [cputype :] [opsys :]**
**[protocols :]**

Where:

**addr1** is the address of the gateway on one of the networks that it connects; for example 197.9.8.1

**addr2** is the address of the gateway on the other network that it connects; for example 197.6.3.8

**name** is the Internet name of the gateway; for example, JANUS.

**cputype** This optional field describes the gateway processor. This field provides you with information; TCP/IP software does not use it. To ensure consistency you should use workstation model numbers for DOMAIN nodes, for example DN330.

**opsys** This optional field describes the gateway processor's operating system. This field provides you with information; TCP/IP software does not use it. To ensure consistency, you should use **DOMAIN/IX** for nodes that run DOMAIN/IX and **AEGIS/DOMAIN/IX** for nodes that run both.

**protocols** This optional field describes the Internet protocols that the gateway supports. For DOMAIN/ETHERNET Gateways, specify:

IP/GW Internet Gateway

and either:

GW/DUMB Non-routing gateway, DOMAIN gateways are normally non-routing.

or:

GW/PRIME Prime gateway. Specify this protocol *only* if you wish to access hosts on other networks that are connected (directly or indirectly) to the ETHERNET network. *Do not* specify both GW/PRIME and GW/DUMB.

Each HOST entry in the local.txt file has the following format:

**HOST : addr [,alt-addr] : name [,nickname] : [cputype :]**
**[opsys :] [protocols :]**

Where:

**addr** is the Internet address of the host; for example, 197.9.8.5

**alt-addr** is one or more alternate Internet addresses for the host, separated by commas; for example, 197.6.3.8

**name** is the Internet name of the host; for example, odin.

**nickname** is one or more alternate names that you can use to access the host. All names must be separated by commas; for example, wotan, snaer.

**cputype** This optional field describes the host processor. This field provides you with information; TCP/IP software does not use it. To ensure consistency, you should use workstation model numbers for DOMAIN nodes, for example, DN330.

**opsys** This optional field describes the host operating system. This field provides you with information; TCP/IP software does not use it. To ensure consistency, you should use **DOMAIN/IX** for nodes that run DOMAIN/IX and **AEGIS/DOMAIN/IX** for nodes that run both.

**protocols** This optional field describes the Internet protocols that the host or gateway supports. This field provides you with information; TCP/IP software does not use it. For DOMAIN hosts, including gateways, specify both of the following. If your node supports other protocols, specify them as defined in RFC 810.

TCP/FTP FTP file transfer protocol

TCP/TELNET Telnet terminal emulator protocol

### HOST_ADDR File

If you will be communicating with any systems on the ETHERNET LAN that do not support the Address Resolution Protocol (ARP), you must put information about the system's ETHERNET address in the */sys/tcp/host_addr* file and run the **maphost** program on the gateway (from the */sys/node_data[node.id]/startup[node_type]* file, after the **tcp_server** is started. See *Managing TCP/IP-Based Communications Products* for details.

## DOMAIN/IX Files

DOMAIN/IX BSD4.2 uses the following files:

- */etc/hosts.equiv*
- */etc/networks*
- */etc/hosts*

You must create the */etc/networks* and */etc/hosts* files *only* if you are using DOMAIN/IX BSD4.2 TCP/IP-based communications on a DOMAIN network that *does not* have a gateway. You must have these files if you use **rlogin**, **lpr**, **rcp**, **rsh**, and **rexec**, as well as **ftp** and **telnet**. You do not have to create these files on DOMAIN networks that use a TCP/IP gateway, because the **/sys/tcp/hostmap/makehost.sh** Shell script creates them from information in the *local.txt* file.

#### *hosts.equiv*

The */etc/hosts.equiv* file lists hosts that are equivalent to your host for login purposes. That is, if a host is on the */etc/hosts.equiv* file for you node it can execute any of the following programs or functions using your node:

- **lpr**(1)
- **lprm**(1)
- **lpq**(1)
- **rcmd**(3X)
- **rcp**(1)
- **rlogin** (1) (without entering a password)

- **rsh**(1)

Note that the node that runs the line printer daemon lpd must be configured for TCP/IP communications and must have the names of all nodes that will print files using this daemon in its */etc/hosts.equiv* file.

The */etc/hosts.equiv* file contains the name of each equivalent TCP/IP host, one name per line. For example:

```
janus
dionysys
bacchus
pan
athena
```

#### *networks* File

The */etc/networks* file consists of a single line with the following form for each Internet network:

```
domain-ring          network_number
```

Where **domain-ring** is a keyword (all lower case) and **network_number** is the Internet network number of the DOMAIN network, and the two values are separated by one or more blanks or TAB characters.

In a DOMAIN network where there is no gateway to additional (non-DOMAIN) networks, this file consists of a single line. For example:

```
domain-ring          197.9.8
```

#### The */etc/hosts* File

The */etc/hosts* file contains the name and Internet address of each TCP/IP host. Each line has the following form:

```
Internet_address     Host_name
```

The address and name must be separated by one or more blanks or TAB characters.

For example:

```
197.9.8.1            JANUS
197.9.8.3            DIONYSUS
197.9.8.5            PAN
197.9.8.8            ATHENA
197.9.8.9            APOLLO
```

# Defining the Configuration

This section describes how you can define the configuration for a DOMAIN network that uses TCP/IP communications. It is helpful if you have never configured TCP/IP and must configure a complete network. You can skip this information and begin the configuration procedure (in Part 2) if you are familiar with TCP/IP or if you are configuring a single node.

To define the configuration you:

1. Select the network, gateway and host Internet Addresses.
2. Define the host mapping files.

3. Select the nodes that execute special server processes.

## Selecting Internet Addresses

You must select Internet addresses for your hosts and gateways before you can configure TCP/IP-based communications.

- You must assign an Internet address to each node that contains TCP/IP software and acts as a host. All the Internet addresses you assign to hosts on a DOMAIN network *must* have the same network number, but different host numbers.
- You must assign *two* internet addresses to a gateway node. Since the gateway node belongs to two local networks, its two Internet addresses have *different* network numbers.

As described before, Internet addresses have variable-length fields for the network numbers and host numbers. You choose a length for your network numbers by choosing Type A, B, or C Internet addresses. Type A, B, and C Internet addresses differ in the size of the network number and host number fields. Table 1-4 summarizes these differences:

**Table 1-4. Type A, B, and C Internet Address Comparison**

| Type | Format | | Description | Example |
|---|---|---|---|---|
| A | W.<br>(network number) | X.Y.Z<br>(host number) | One-byte network number, three-byte host number. | 48.1.2.1<br>(decimal) |
| B | W.X.<br>(network number) | Y.Z<br>(host number) | Two-byte network number, two-byte host number. | 139.2.9.2<br>(decimal) |
| C | W.X.Y.<br>(network number) | Z<br>(host number) | Three-byte network number, one-byte host number. | 192.9.1.2<br>(decimal) |

For example, the Type A address in the table has a network number of 48. Contrast this with the network number in the Type B address (139.9) and in the Type C address (192.9.1). In the Type C address, you may only assign host numbers 0 - 255, whereas in the Type A address, you can assign host numbers 0.0.0 through 255.255.255.

You should choose an address type, network numbers, and host numbers to suit the number of hosts at your site. If you plan to use DOMAIN TCP/IP to communicate on the ARPANET, you must apply to the Network Information Center (NIC) for a network number.

Use these guidelines to select Type A, B, or C addressing:

- Type A addresses allow for large numbers of hosts, up to 16,777,216, on a single network. Choose Type A addressing ONLY if you plan never to access the ARPANET, because many Type A addresses are already assigned by the Network Information Center (NIC).

  Type A network numbers must be in the range 0 - 127. Type A host numbers must be in the range 0.0.0 through 255.255.255. Therefore, Type A addresses range from 0.0.0.0 through 127.255.255.255.

- Type B addresses allow for up to 65,535 hosts. Choose Type B addresses if you plan a very large number of hosts. The NIC will sometimes assign you a Type B network number.

  Type B network numbers must be in the range 128.0 - 191.255. Type B host numbers must be in the range 0.0 through 255.255. Therefore, Type B addresses range from 128.0.0.0 through 191.255.255.255.

- Type C addresses allow for only 256 hosts, but allow network numbers between 192.0.0 and 255.255.255 (three-byte network numbers). Therefore, Type C addresses range from 192.0.0.0 through 255.255.255.255. If you apply to the NIC for a network number, you will usually receive a Type C network number.

In most cases, you should use Type C addresses even if you don't plan to register with NIC and access the ARPANET immediately. We recommend Type C addresses because you can choose a number not yet used by any other network. (To see the reserved network numbers, read the file */sys/tcp/hostmap/hosts.txt*.) Then, if you wish to access the ARPANET in the future, you will not need to change network numbers in your Internet addresses.

## Defining the Mapping Files

Once you have defined the Internet addresses, you can determine the location of the network-wide mapping files that you will need, and their contents. You should:

1. Determine the service node (or nodes) that will have the host mapping tables.
2. Determine the name of the BSD4.2 administrative node (or nodes) that will have the */etc* directory.
3. If you are configuring TCP/IP that uses a gateway, define the contents of the *//service_node/sys/tcp/local.txt*.

   If you are configuring BSD4.2 TCP/IP on a network that does not have an ETHERNET gateway, define the contents of the */etc/networks* and */etc/hosts* files.

### Determining the Service and Administrative Nodes

You may wish to have one or a few service nodes. If you have a single such node, then you only have to maintain a single database. If you have a large network or wish to ensure the availability of mapping information, additional nodes can be helpful. However, each host is linked to a single node, and you will have to manually change the links in order to change the host's service node. Also, you must update each service node whenever mapping information changes. The same considerations apply to selecting administative nodes.

The service node *does not* have to be a TCP/IP host or gateway. Because the DOMAIN/IX administrative node uses BSD4.2, it is usually a TCP/IP host.

### Defining the *local.txt* file

Define the *local.txt* file by determining the information required for each network, gateway, and host in your internet. If you allow connection to the DARPA internet, do not include any information that is already in the *hosts.txt* file.

If you are connecting the DOMAIN network to an Internet, that is to more than just a single ETHERNET, you should apply the following considerations when you define the *local.txt* GATEWAYS entries.

- The order that you use for the GATEWAY entries affects the gateway that is used when TCP/IP software establishes a connection. TCP/IP always tries to use the first applicable gateway on the list.

### Defining the */etc* files

You need to define the */etc* directory files if you are configuring a DOMAIN/IX BSD4.2 network that does not use gateways. In this case, the */etc/networks* file consist of a single line with the network name and address, for example,

```
our-net-ether       197.9.8
```

The */etc/hosts* file must contain the Internet address and name of each BSD4.2 node on the network.

## Determining Server Processes

Before you begin configuring a network that uses TCP/IP you should determine which nodes use which server processes. Table 4-1 lists these processes, and the "Servers and Daemons" section describes where they are required.

# Part 2 - Configuring TCP/IP

This part describes how to configure a DOMAIN network or node that uses TCP/IP communications or supports TCP/IP based communications. It also briefly describes requirements for configuring TCP/IP on non-DOMAIN hosts.

**NOTE:** Before you configure a node you should be familiar with the terms and concepts that are described in Part 1.

If you are configuring a DOMAIN network to use TCP/IP-based communications for the first time, you must configure all of the nodes that use or support TCP/IP communications.

## Configuring TCP/IP on an Internet

If you are configuring a DOMAIN network that uses an ETHERNET gateway to communicate with other networks, use the procedures described in this appendix in the following order:

1. Use Procedure 1 to configure the service node or nodes.
2. Use Procedure 2 to configure each node that uses DOMAIN/IX TCP/IP, that is, each host and gateway.
3. Use Procedure 4 to configure the remote (non-DOMAIN/IX) hosts that will communicate with the DOMAIN network.

## Configuring DOMAIN-Only BSD4.2 TCP/IP

If you are configuring a DOMAIN network that supports DOMAIN/IX BSD4.2, but does not have access to any non-DOMAIN networks, use procedure 3 to configure TCP/IP on each BSD4.2 node.

# Configuring DOMAIN/IX Nodes

For purposes of installing and configuring TCP/IP, DOMAIN/IX nodes can be divided into three classes:

| | |
|---|---|
| Host | Nodes that use TCP/IP communications to communicate with other hosts |
| Gateway | Nodes that connect two networks (Gateways are also TCP/IP hosts) |
| Service | Nodes that aid TCP/IP communications but do not necessarily act as hosts. Service nodes provide host mapping tables and contain the */etc* directory. |

Additionally, the exact procedures that you use depend on whether your node:

- Has a disk or is diskless
- Is a DOMAIN/IX BSD4.2 node on a DOMAIN network that is not connected to any other networks.

This part of the appendix includes procedures for all types of nodes.

Table 2-1 lists the procedures in this part and types of hosts that they can configure. The rest of this part consists of the configuration procedures.

**NOTES:** These procedures refer to the */sys/node_data[.nodeid]* and *'node_data* directories. These two terms are not interchangeable; the first is an absolute reference, the second is relative. Using the wrong name in a link can result in circular file references.

*Always* configure the service node before you configure hosts and gateways. This order insures that all tables are up-to-date and eliminates any duplication of effort

**Table 2-1. Configuration Procedures**

| *Procedure* | *System* | *Type* |
|---|---|---|
| 1 | | Service node |
| 2 | DOMAIN/IX | Internet Host or Gateway |
| 3 | DOMAIN/IX | Host on a single DOMAIN network |
| 4 | Various | Remote Host |

All of these procedures describe the steps required for either a disked or diskless node. (However, you should not have a diskless service node, as the node's main function is to provide mapping files information.) Similarly, procedures 2 and 3 describe procedures for hosts and gateways. Some steps in these procedures are required for only disked, or only diskless nodes or for gateways only, and they are marked as such.

## PROCEDURE 1. Configuring the Service Node

**NOTE:** This procedure assumes that you are using a DOMAIN/IX BSD4.2 C Shell.

### ❑ Task 1: Select the Internet Addresses

If you have not already done so, determine the Internet addresses for all hosts, including the remote hosts that you can access across the gateway.

### ❑ Task 2: Install the Software

If you have not already done so, use the procedures described in the associated Release Notes to install software *in the following order*:

**NOTES:** See the TCP/IP Release Notes to determine the revision level required for each of the following software.

If you are configuring a diskless host, you must install the following software on the hosts's partner node.

1. DOMAIN
2. C, DOMAIN/IX

   **Note:** When you install BSD4.2 DOMAIN/IX you must give **/etc/run_rc** root ownership. Otherwise, processes required for TCP/IP will not execute properly.

3. TCP/IP

   A service node can be a host or gateway, but does not have to be either. The files that are installed when you install TCP/IP depend upon the options you select.

   The Release Notes list the files are installed in each case. Use the **ls** command to make sure that all the required files are installed. For example, if this node is a service node only:

```
% ls /sys/tcp
host_addr   hostmap
% ls /sys/tcp/hostmap
hashnic     hosts./txt      htable          local.txt
makehdb     makehost.sh     sortnic
```

### ❑ Task 3: Configure the hosts.txt File

If you are connecting your DOMAIN network to a Department of Defense (DoD) Internet, then you should use a */sys/tcp/hostmap/hosts.txt* file. This file contains the Internet mapping information for all the hosts, gateways, and networks on the ARPANET and other networks on the Internet.

**NOTE:** The copy of *hosts.txt* that we supply may not be the most current version available. After you have configured at least one host, you can obtain the current *hosts.txt* file and update the mapping tables by running the **gettable**(8) command.

If you are not connecting your DOMAIN network to the DoD Internet, replace this file with an empty file. If you wish to save the information, rename the file to */sys/tcp/hostmap/hosts.txt.SRn*, where *n* is the software release number. Using an empty file prevents the configuration procedure from adding information in the *hosts.txt* file to the mapping tables, and speeds up Task 5.

### ❑ Task 4: Configure the local.txt File

Edit the */sys/tcp/hostmap/local.txt* file.

1. Add a **NET** entry to the file for *each* network that you can access, including your DOMAIN network. Do not add any entries that are already in the *hosts.txt* file. The **NET** entries should be the first entries in the file. **NET** entries have the following format:

   **NET** : netaddr : netname

   Where netaddr is the network's Internet network number followed by **.0**, **.0.0**, or **.0.0.0**, depending on whether the address is Type A, B, or C, and netname is the name that you have assigned to the network.

   For example, if your DOMAIN network has an Internet network number of 197.9.8 and is named SAMPLE-NET, enter the following in the *local.txt* file:

   ```
   NET : 197.9.8.0 : SAMPLE-NET
   ```

2. Add a **GATEWAY** entry to the file for *each* gateway between networks that you can access. Do not add any entries that are already in the **hosts.txt** file. All **GATEWAY** entries should follow the **NET** entries in the file. **GATEWAY** entries have the following format:

   **GATEWAY** : addr1, addr2 : name : [cputype :] [opsys :] [protocols :]

   Where addr1 is the gateway's Internet address on one network, addr2 is the gateway's Internet address on the second network, and name is the gateway's host name.

   For example, if your gateway is named JANUS, is a DSP90 running AEGIS, and has an Internet address of 197.9.8.1 on the DOMAIN network and an address of 197.6.3.8 on the ETHERNET, enter the following in the local.txt file:

   ```
   GATEWAY : 197.9.8.1, 197.6.3.8 : JANUS : DSP90 : DOMAIN : IP/GW , GW/DUMB
   ```

3. Add a **HOST** entry to the file for each host *and* each gateway on *all* networks that you can access. Do not add any entries that are already in the **hosts.txt** file. All **HOST** entries should follow the **GATEWAY** entries in the file. **HOST** entries have the following format:

   **HOST** : addr[,alt-addr] : name [.nickname] : [cputype :] [opsys :] [protocols :]

   For example, if a host is named DIONYSUS, is a DN550 running AEGIS, and has an Internet address of 197.9.8.3, enter the following in the local.txt file:

   ```
   HOST : 197.9.8.3 : DIONYSUS : DN550 : DOMAIN :
     TCP/TELNET, TCP/FTP:
   ```

### ❑ Task 5: Create the Host Tables

Run the **/sys/tcp/hostmap/makehost.sh** Shell script from any node. This script converts the *local.txt* file into a format that TCP/IP software can use.

For example:

```
% /sys/tcp/hostmap/makehost.sh
Formatting tables
Sorting Tables
Formatting tables (pass 2)
Hashing
input: hosts.tmp1, 1466 lines
output: hosts.hst, 215519 bytes
hash: 1542 bytes, 257 entries, 0 holes
keys: 9794 bytes 534 entries (largest 662 longest 37)
data: 116019 bytes, 1466 entries (largest 740)
(file) "hosts.hst" moved.
(file) "gateways" moved.
(file) "hosts.tmp" deleted.
(file) "hosts.tmpl" deleted.
(file) "nets.tmp" deleted.
(file) "hosts.txt1" deleted.
(file) "nets.txt1" deleted.
```

### ❑ Task 6: : Edit the /etc/hosts.equiv File

The */etc/hosts.equiv* may or may not be physically located on the service node. We include the file here because it performs a service function. Also, if you edit */etc/hosts.equiv* at this point when you configure a network for the first time, you do not have to edit it when you configure individual hosts.

As a general rule, the */etc* directory resides on the DOMAIN/IX administrative node and all other nodes access it through links. In most cases, only the system administrator may be able to edit this file.

Enter the name of each host that you will allow to execute the **rsh**(1) program or **rcmd**(3) routine, or to use **rlogin**(1) without entering a password in the */etc/hosts.equiv* file. Each entry in this file consists of a single line with the host name, for example:

```
lilly
```

### ❑ Task 7: Configure Other Services

The service node can also be a TCP/IP host or gateway. You should now configure that facility.

- If the Service is a TCP/IP host or gateway, continue with Procedure 2, skipping steps 1 through 4.

## END OF PROCEDURE 1.

## PROCEDURE 2. Configuring A DOMAIN/IX BSD4.2 Host or Gateway Node

### ❑ Task 1: Select an Internet Address

If you have not already done so, select an Internet address for the host.

### ❑ Task 2: Stop the TCP_SERVER

If you are already using TCP/IP on the node, find the process ID of the **tcp_server** and stop that process by entering the following commands:

```
% ps aux
USER      PID    SZ     STAT   TIME         COMMAND

root      2      0      R      7693:05      null
root      34     0      S      3425         tcp_server
.
.
.

% kill -9 34
```

### ❑ Task 3: Install the Software

If you have not already done so, use the procedures described in the associated Release Notes to install software in the following order:

**NOTES:** See the appropriate Release Notes to determine the revision level required for each of the following software.

If you are configuring a diskless host, you must install the following software on the host's partner node.

1. DOMAIN
2. C, DOMAIN/IX BSD4.2

   **Note:** When you install the BSD4.2 you must give **/etc/run_rc** root ownership. Otherwise, processes required for TCP/IP will not execute properly.

3. TCP/IP

   After you install TCP/IP, execute the **ls** command to check that the TCP/IP software was properly installed and that the four required links from the /*sys*/*tcp* directory to the mapping service node exist. In the following example, **edny** is the mapping service node:

```
% ls -l /sys/tcp
total 452
-rwxrwxrwx  1 root    9824  Sep 19 13:42 ether_diag
-rwxrw-rwx  1 root   78664  Sep 19 13:42 ftp_server
lrwxrwxrwx  1 root      27  Jan  8 10:06 gateways -> //edny/sys/tcp/gateways
lrwxrwxrwx  1 root      28  Jan  8 10:06 host_addr ->//edny/sys/tcp/host_addr
lrwxrwxrwx  1 root      26  Jan  8 10:06 hostmap -> //edny/sys/tcp/hostmap
lrwxrwxrwx  1 root      28  Jan  8 10:06 hosts.hst-> //edny/sys/tcp/hosts.hst
drwxrwxrwx  1 root    1024  Sep 19 13:42 lib
-rwxrwxrwx  1 root   27050  Sep 19 13:42 makegate
-rwxrwxrwx  1 root    3122  Sep 19 13:42 maphost
-rwxrwxrwx  1 root      19  Sep 19 13:57 networks
-rwxrwxrwx  1 root  103214  Sep 19 13:42 tcp_server
-rwxrwxrwx  1 root   91640  Sep 19 11:02 tcp_server.sr8.gateway
-rwxrwxrwx  1 root    4722  Sep 19 13:42 tcpinit
-rwxrwxrwx  1 root    1662  Sep 19 11:02 tcpreset
-rwxr-xr-x  1 root   21648  Dec 17 15:27 telnet_server
-rwxrwxrwx  1 root       7  Sep 19 13:56 thishost
```

### ❑ Task 4: Update the Mapping Service Node Host Tables

**NOTE:** If you are configuring TCP/IP for the first time on your network you should configure your mapping service nodes before you configure your host nodes. The mapping service node files will then be up-to-date. If you've already configured your service nodes, skip this task.

If you are configuring a single node for the first time, or if you are changing the node's Internet address, use the following steps to update the TCP/IP mapping service node's host mapping tables:

**NOTE:** The install procedure creates the pathname /*sys*/*tcp*/*hostmap* on your local node as a link to the directory //*mapping_service_node*/*sys*/*tcp*/*hostmap*.

1. Add (or change) the node's **HOST** entry in the /*sys*/*tcp*/*hostmap*/*local.txt* file. Each host *and* gateway must have an entry with the following format:

   **HOST** : addr[,alt-addr] : name [.nickname] : [cputype :] [opsys :] [protocols :]

   For example, if your host is named DIONYSUS, is a DN550 running DOMAIN/IX, and has an Internet address of 197.9.8.3, enter the following in the *local.txt* file:

   ```
   HOST : 197.9.8.3 : DIONYSUS : DN550 : DOMAIN/IX : TCP/TELNET, TCP/FTP:
   ```

2. **Gateways Case:** If you are configuring a gateway it must also have a **GATEWAY** entry in the *local.txt* file. All **GATEWAY** entries must precede the **HOST** entries in the file.

   **GATEWAY** : addr1, addr2 : name : [cputype :] [opsys :] [protocols :]

   Where addr1 is the gateway's Internet address on one network, and addr2 is the gateway's Internet address on the second network.

   For example, if your gateway is named JANUS, is a DSP90 running DOMAIN/IX, and has an Internet address of 197.9.8.1 on the DOMAIN network and an address of 197.6.3.8 on the ETHERNET, enter the following in the *local.txt* file:

   ```
   GATEWAY : 197.9.8.1, 197.6.3.8 : JANUS : DSP90 :
        DOMAIN/IX : IP/GW , GW/DUMB
   ```

3. Run the **/sys/tcp/hostmap/makehost.sh** Shell script from any node. This script converts the *local.txt* file into a format that TCP/IP software can use.

   For example:

```
% /sys/tcp/hostmap/makehost.sh
Formatting tables
Sorting Tables
Formatting tables (pass 2)
Hashing
input: hosts.tmp1, 1466 lines
output: hosts.hst, 215519 bytes
hash: 1542 bytes, 257 entries, 0 holes
keys: 9794 bytes 534 entries (largest 662 longest 37)
data: 116019 bytes, 1466 entries (largest 740)
(file) "hosts.hst" moved.
(file) "gateways" moved.
Updating 4.2bsd host tables
(file) "/etc/hosts" moved.
(file) "/etc/gateways" moved.
(file) "/etc/networks" moved.
(file) "hosts.tmp" deleted.
(file) "hosts.tmp1" deleted.
(file) "nets.tmp" deleted.
(file) "hosts.txt1" deleted.
(file) "nets.txt1" deleted.
```

### ❑ Task 5: Update the /etc/hosts.equiv File

**NOTE:** In most installations the */etc* directory resides on an administrative node and all other nodes access it through links. In this case, only the system administrator may be able to edit this file.

If you will allow this host to execute the **rsh**(1) program or **rcmd**(3) routine, or to use **rlogin**(1) without entering a password, then enter the host name in the */etc/hosts.equiv* file. Each entry in this file consists of a single line with the host name, for example:

```
lilly
```

### ❑ Task 6: Edit the Node Startup Files

Use the following steps to update your node startup files.

1. Edit the */sys/node_data[.node_id]/startup[.type]* file to include the following commands:

   REQUIRED: These commands **must** be in the following order. Otherwise, processes that are initialized by **run_rc** will not execute. However, you can include other commands between them.

```
cps /sys/tcp/tcp_server -n tcp_server
env SYSTYPE     'bsd4.2'
cps /etc/run_rc
```

2. Edit the */etc/rc* file by removing the comment character (#) from the lines that contain processes that you want to run *on this host*. The "BSD4.2 Daemons" section in Part 1 describes the processes used for TCP/IP-based communications in detail.

In most cases you will want to run **inetd**, which can start several TCP/IP-related daemons. In this case, uncomment the following lines:

```
if [ -f /etc/inetd ]; then
    /etc/inetd &
fi
```

3. If you specified **inetd** in step 2, edit the */etc/inetd.conf* file by removing the comment character (#) from the lines that contain processes that you want to run *on this host*.

### ❑ Task 7: Edit the Shell Login Files

Edit your **.cshrc** file if you use the C shell or your **.profile** file if you use the Bourne shell to include the **/com** directory in the search path. The */com* directory includes the **tcpstat**, **host**, **net**, and **edip** commands that you use to monitor and manage TCP/IP communications. By including the */com* directory in the search path you eliminate the need to specify the full pathname for these commands.

The */com* directory also includes the DOMAIN versions (as opposed to BSD4.2 versions) of **telnet** and **ftp**. Therefore, the */com* directory must follow the */usr/ucb* directory in the search path.

For example, include the following line in your **.cshrc** file:

```
set path=(. /bin /usr/bin /usr/ucb /com )
```

or include the following line in your **.profile** file:

```
export PATH
PATH=:.:/bin:/usr/bin:/usr/ucb:/com:)
```

### ❑ Task 8: Edit the thishost File

**Disked Case:** If your node has a disk, edit the */sys/node_data/thishost* file to replace the word "apollo" with your host's name. This file must consist of a single line with your host's name, for example:

```
lilly
```

**Diskless Case:** If your node is diskless, copy the partner node's */sys/node_data/thishost* file to */sys/node_data.node_id/thishost*. Then edit this file to replace the host name with your host's name.

### ❑ Task 9: Edit the networks File

Edit the */sys/node_data[.node_id]/networks* file to include the host's Internet address and physical interface.

This file looks as follows immediately after you install TCP/IP:

```
0.0.0.0 on dr0
0.0.0.0 on il0
```

**Diskless Case:** If your node is diskless, copy the partner node's */sys/node_data/networks* file to */sys/node_data.node_id/networks*. Then continue with the **Host** or **Gateway** case.

**Host Case:** Hosts have a single address and a single physical interface. The host's physical interface must be **dr0**. Therefore, a host's *networks* file must have a single entry.

Edit the file by changing the address on the first line and deleting the second line. For example, if your host's Internet address is 197.9.8.3, you should edit the *networks* file to look as follows:

197.9.8.3 on dr0

**Gateway Case:** Gateways have one address and physical interface for each network. The physical interface for the DOMAIN network is **dr0**; the physical interface for the ETHERNET network is **il0**. Therefore, a host's *networks* file must have two single entries.

Edit the file by changing the address on each line. For example, if your gateway's Internet address on the DOMAIN network is 197.9.8.1 and the address on the ETHERNET LAN is 197.6.3.8, you should edit the *networks* file to look as follows:

```
197.9.8.1 on dr0
197.6.3.8 on il0
```

### ❑ Task 10: Initialize TCP/IP

You initialize TCP/IP by starting the node's **tcp_server** process and updating its host tables, if necessary.

1. Start the TCP_server.

   - Reboot the node. Because you have included the required command in the node startup file, the TCP_Server automatically initializes when the node reboots. Rebooting the node also ensures that any other TCP/IP processes are started, and that all processes are up-to-date.

     To reboot the node:

     1. Enter the DM **ex** command as follows:

        ```
        Command: ex
        ```

        All current processes stop executing, the operating system exits, and the node enters the bootshell, which prompts you with parentheses. Enter the **go** command as follows:

        ```
        ) go
        ```

        AEGIS reboots and returns you to the DM login message. You can now log in and use TCP/IP.

   **NOTE:** When the TCP_server initializes it automatically executes the following programs:

   - **/sys/tcp/tcpinit**
   - **/sys/tcp/makegate**

2. If your node will communicate with non-DOMAIN hosts that do not understand the Address Resolution Protocol (ARP), run the **/sys/tcp/maphost** program.

**END OF PROCEDURE 5-2.**

## PROCEDURE 3. Configuring a DOMAIN/IX BSD4.2 Host that Communicates Only On the DOMAIN Network

### ❑ Task 1: Select an Internet Address

If you have not already done so, select an Internet address for the host.

### ❑ Task 2: Stop the TCP_SERVER

If you are already using TCP/IP on the node, find the process ID of the **tcp_server** and stop that process by entering the following commands:

```
% ps aux
USER      PID    SZ     STAT   TIME         COMMAND

root      2      0      R      7693:05      null
root      34     0      S      3425         tcp_server
.
.
.

% kill -9 34
```

### ❑ Task 3: Install the Software

If you have not already done so, use the procedures described in the associated Release Notes to install software *in the following order*:

**NOTES:** See the appropriate Release Notes to determine the revision level required for each of the following software.

If you are configuring a diskless host, you must install the following software on the hosts's partner node.

1. DOMAIN
2. C
3. DOMAIN/IX BSD4.2

   **Note:** When you install BSD4.2 you must give **/etc/run_rc** root ownership. Otherwise, processes required for TCP/IP will not execute properly.

### ❑ Task 4: Update the /etc Directory files

If you are configuring a single node for the first time, or if you are changing the node's Internet address, use the following steps to update the TCP/IP files in the */etc* directory:

**NOTE:** In most installations the */etc* directory resides on an administrative node and all other nodes access it through links. In this case, only the system administrator may be able to edit these files.

1. Add an entry to the */etc*/*hosts* file. Each entry consists of a single line with the Internet address followed by the host name. For example, if your host's Internet Address is 197.9.8.3 and it's name is DIONYSUS, add the following line to the */etc*/*hosts* file:

```
197.9.8.3  dionysus
```

2. Make sure there is an entry in the */etc/networks* file. There should only be a single line in this file, consisting of a network name followed by the Internet network number. Therefore, you only need to edit this file *once*, when you configure the first host on your network.

   For example, if your network is named **sample-ring** and your network number is 197.9.8, the */etc/networks* file should look as follows:

   ```
   sample-ring        197.9.8
   ```

3. If you will allow this host to execute the **rsh**(1) program or **rcmd**(3) routine, or to use **rlogin**(1) without entering a password, then enter the host name in the */etc/hosts.equiv* file. Each entry in this file consists of a single line with the host name, for example:

   ```
   lilly
   ```

### ❑ Task 5: Edit the Node Startup Files

Use the following steps to update your node startup files. See Part 1 for a discussion of the processes that you can start using each of these files:

1. Edit the */sys/node_data[.node_id]/startup[.type]* file to include the following commands. The commands must be in the following order, but you can include other commands between them.

   ```
   cps /sys/tcp/tcp_server -n tcp_server
   env SYSTYPE  'bsd4.2'
   cps /etc/run_rc
   ```

2. Edit the */etc/rc* file by removing the comment character (#) from the lines that contain processes that you want to run *on this host*. The "BSD4.2 Daemons" section in Part 1 describes the processes used for TCP/IP-based communications in detail.

   In most cases you will want to run **inetd**, which can start several TCP/IP-related daemons. In this case, uncomment the following lines:

   ```
   if [ -f /etc/inetd ]; then
       /etc/inetd &
   fi
   ```

3. If you specified **inetd** in step 2, edit the */etc/inetd.conf* file by removing the comment character (#) from the lines that contain processes that you want to run *on this host*.

### ❑ Task 6: Edit the Shell Login Files

Edit your **.cshrc** file if you use the C shell or your .profile file if you use the Bourne shell to include the */com* directory in the search path. The */com* directory includes the **tcpstat**, **host**, **net**, and **edip** commands that you use to monitor and manage TCP/IP communications. By including the */com* directory in the search path you eliminate the need to specify the directory in these commands.

The */com* directory also includes the DOMAIN versions (as opposed to BSD4.2 versions) of **telnet** and **ftp**. Therefore, the */com* directory must follow the */usr/ucb* directory in the search path.

For example, include the following line in your **.cshrc** file:

```
set path=(. /bin /usr/bin /usr/ucb /com)
```

or include the following line in your **.profile** file:

```
export PATH
PATH=:.:/bin:/usr/bin:/usr/ucb:/com:
```

### ❑ Task 7: Edit the thishost File

Edit the */sys/node_data[.node_id]/thishost* file to replace the word "apollo" with your host's name. This file must consist of a single line with your host's name, for example:

```
lilly
```

### ❑ Task 8: Edit the networks File

Edit the */sys/node_data[.node_id]/networks* file to include the host's Internet address and physical interface. The physical interface for a host that is not a gateway must be **dr0**.

This file looks like this immediately after you install TCP/IP:

```
0.0.0.0 on dr0
0.0.0.0 on il0
```

If your Internet address 197.9.8.3, you should edit this file to look as follows:

```
197.9.8.3 on dr0
```

### ❑ Task 9: Initialize TCP/IP

You initialize TCP/IP by starting the node's **tcp_server** process.

- Reboot the node. Because you have included the required command in the node startup file, the TCP_Server automatically initializes when the node reboots. Rebooting the node also ensures that any other TCP/IP server processes are started, and that all processes are up-to-date.

  To reboot the node:

  1. Enter the DM **ex** command as follows:

     ```
     Command: ex
     ```

     All current processes stop executing, the operating system exits, and the node enters the boot-shell, which prompts you with parentheses.

  2. Enter the go command as follows:

     ```
     ) go
     ```

     AEGIS reboots and returns you to the DM login message. You can now log in and use TCP/IP.

**NOTE:** When the TCP_server initializes it automatically executes the following programs:

- **/sys/tcp/tcpinit**
- **/sys/tcp/makegate**

**END OF PROCEDURE 3.**

# Configuring Non-DOMAIN Hosts

In the same way that you must add the names and Internet addresses of foreign networks, gateways, and hosts to the *local.txt* file on the DOMAIN network, you must add the names and Internet addresses of the DOMAIN network, gateway, and hosts to some equivalent file or files on ETHERNET side of the connection. Each foreign host that will communicate with the DOMAIN network must have access to this information.

We can not provide procedures for all possible situations; however, we *can* provide some general rules that may be helpful if you are a first-time user of TCP/IP. The following sections cover the case if you are using a DARPA standard TCP/IP implementation and if you are using BSD4.2 UNIX TCP/IP.

## Configuring DARPA TCP/IP Hosts

If you are configuring a host that uses DARPA standard TCP/IP you must edit the hosts.txt or an equivalent file used for hosts that are not listed by the NIC to include the DOMAIN Network, gateway, and hosts. After you edit this file you may have to execute additional steps to update the host's internal mapping tables.

## Configuring BSD4.2 UNIX Hosts

If you are configuring a host that uses BSD4.2 UNIX you must do the steps described in Procedure 2 *at the non-DOMAIN host*. We assume several things in this procedure, including:

- You have the permissions required to edit such files as /*etc*/*hosts* on the non-DOMAIN host.
- You understand the /*etc*/*rc* file on the UNIX system well enough to set up the servers and daemons that are appropriate for the UNIX host.

## **PROCEDURE 4.** Configuring a Non-DOMAIN BSD4.2 Host to Communicate with a Host on a DOMAIN Network

### ❑ Task 1: Edit /etc/networks

Make sure that there is an entry in the */etc/networks* file for the DOMAIN network that you want to access. Unless you are adding the first DOMAIN host, there should already be an entry in the file. */etc/networks* file entries consist of the network's name followed by its Internet network number. For example:

```
domain-ring2      197.9.8
```

Note that in this example the three digit Internet network number indicates that the DOMAIN network uses class C Internet addresses.

The method you use to manage the */etc/networks* file depends upon your system administration procedures. For example, you might edit the */etc/networks* file directly. Or, you might edit a file similar to *local.txt* file and use the **htable**(8) command to put the entries in the */etc/networks* file. See your host's BSD4.2 documentation for more information on **htable**.

### ❑ Task 2: Edit /etc/hosts

Make sure that there is an entry in the */etc/hosts* file for the DOMAIN host (or hosts) that will communicate with this host. Entries in the */etc/hosts* file entries consist of the network's name followed by its Internet network number. For example:

```
197.9.8.3  dionysus
```

The method you use to manage the */etc/hosts* file depends upon your system administration procedures. For example, you might edit the */etc/hosts* file directly. Or, you might edit a file similar to *local.txt* file and use the **htable**(8) program to put the entries in the */etc/hosts* file. See your host's BSD4.2 documentation for more information on **htable**.

### ❑ Task 3: Edit /etc/hosts.equiv

Enter in the */etc/hosts.equiv* file the name of each DOMAIN host that you will allow this host to access using the **rsh**(1) program or **rcmd**(3) routine, or to use **rlogin**(1) without entering a password. Each entry in this file consists of a single line with the host name, for example:

```
dionysus
```

### ❑ Task 4: Edit /etc/rc

Edit the non-DOMAIN host's */etc/rc* file, if necessary. This file controls the services that this host provides to other hosts; it is a UNIX Shell script that executes automatically when the remote UNIX system is rebooted. Some installations use a */etc/rc.local* file for commands that are pertinent to a single site. For more details on **rc** see **rc**(8) in the *BSD4.2 UNIX Programmer's Manual*.

The */etc/rc* file must specify the **routed** routing daemon and any other daemons, such as **telnetd** and **ftpd**, that you require to enable TCP/IP communications with the DOMAIN hosts.

### ❑ Task 5: Ensure that the daemons are Executing

Make sure the routing daemon, **routed**, and any other daemons that you require for TCP/IP communications with DOMAIN hosts run on this host. You can check whether this process is running by using the UNIX **ps** command. If necessary, reboot the system or start the processes manually.

## END OF PROCEDURE 4.

# Appendix K

# Line Printer Management for DOMAIN/IX *bsd4.2*

# Appendix K: Line Printer Management for DOMAIN/IX bsd4.2

## 1. Overview

This document describes the structure and installation procedure for the line printer spooling system developed for the *bsd4.2* version of DOMAIN/IX SR9.0.

The line printer system supports:

- multiple printers,
- multiple spooling queues
- local and remote printers, and
- printers attached via serial lines that require line initialization.

The line printer system also supports raster output devices like Varian and Versatec and laser printers like Imagen.

The line printer system consists of the following files and commands:

| | |
|---|---|
| */etc/printcap* | printer configuration and capability data base |
| */usr/lib/lpd* | line printer daemon |
| */usr/ucb/lpr* | program to enter a job in a printer queue |
| */usr/ucb/lpq* | spooling queue examination program |
| */usr/ucb/lprm* | program to delete jobs from a queue |
| */etc/lpc* | program to administer printers and spooling queues |
| */usr/spool/lpd/** | spooling directories |
| */usr/spool/lpd/servername* | node on which **lpd** runs (optional) |

You must log in as the super-user to run several of the components of the line printer system; that is, you must log in using the 'root' user name and password.

## 2. How Does It Work?

Normally, only one node per DOMAIN ring will run the line printer daemon, **/usr/lib/lpd**. That node will usually start the daemon at boot time, by means of the */sys/node_data/etc.rc* file. When **/usr/lib/lpd** is started, the daemon goes through the *printcap* file and restarts any printers that have jobs in their queues. Then **lpd** listens for print requests from: nodes on your DOMAIN network, nodes on another DOMAIN network to which you're connected, and/or foreign hosts that are connected to your network.

When you submit a print request using the **lpr** command, **lpd** creates a copy of itself to process the request. (The original **lpd** process continues to listen for requests.) The **lpr** command places the actual material to be printed in the appropriate spool directory (*/usr/spool/lpd/**), and the copy of **lpd** then schedules the job's printing. If the printer you specified in the **lp** command line is unavailable for some reason, or if the machine to which it is connected is not operating, the request will remain in the spooling directory (or 'queue') until it is

removed with the **lprm** command, or until the faulty printer or machine becomes available.

The file */etc/printcap* is a data base that describes the printers that are available to machines using the **lp** commands. The manual entry **printcap**(5) defines the format of this data base, as well as default values for important items like the directory in which spooling is performed.

The DOMAIN/IX version of the *printcap* file has one additional entry, which is explained more fully on the **printcap**(5) *DOMAIN/IX Programmer's Reference* Manual pages. It is:

**pc** Provides an interface to print commands you can use instead of sending output to **lp** or **rp**. In DOMAIN/IX, this is set to use the DOMAIN **/com/prf** command sequence.

### 2.1. Prerequisites for DOMAIN/IX

From what we've said above, we can see that the **lp** system must have an **lpd** process running, and an entry in */etc/printcap* for the printer to which requests will be sent. We'll assume that the necessary **lp** commands, like **lpc** and **lpr**, have been installed in the correct places on the system.

To set up **lpd** to run at boot time, uncomment (remove the # from the beginning of) the three lines in the */sys/node_data/etc.rc* file that read:

```
#if [ -f /usr/lib/lpd ]; then
#       /usr/lib/lpd &
#fi
```

Shut down the node and restart it. The DOMAIN/IX implementation of **lpd** includes an optional file */usr/spool/lpd/servername* that can be used if you want only one machine on your DOMAIN network to run **lpd**. If the file exists, it contains the TCP host name of the one machine on the network that is allowed to run **lpd**. If you attempt to start an **lpd** process on a machine other than the one specified in */usr/spool/lpd/servername*, **lpd** will return an error message that specifies the name of the only machine that is allowed to run **lpd**. If the file does not exist, any number of machines on the network can run **lpd**, but only one should run it at a time.

The */etc/printcap* file, as installed, contains several default descriptions for printer types. You may need to create an entry for another printer type. In this case, use the discussion of *printcap* later in this document, as well as the *DOMAIN/IX Programmer's Reference* manual pages for **printcap**(5) and **termcap**(5), as your guide. Be certain not to leave a blank line between entries in the */etc/printcap* file, as this will cause the **lp** system to think that there is a valid printer on the system with no name, and it may attempt to send requests there.

The */etc/hosts.equiv* file is a list of machine names. In general, this file allows all the machines named in it to be treated as equivalent; for example, if your machine name is in this file, you can **rlogin** to any other machine named in the file, without going through the normal user and password authorization procedures. In order to use the line printer system on your network, a machine must have its TCP/IP host name in this file. There should be only one */etc/hosts.equiv* file per network. (See below for references to information about DOMAIN/IX TCP/IP.)

The DOMAIN/IX implementation of the line printer spooler runs over a TCP/IP connection. Therefore, DOMAIN/IX TCP/IP must be configured on all machines that are to use the **lp** system, and the DOMAIN *tcp_server* must be running on all nodes. See *Appendix J* to the *DOMAIN/IX User's Guide*, as well as the Release Notes for Software Release 9.2, for information on configuring TCP/IP correctly.

Note that the node to which the printer is physically connected does not have to have either TCP/IP or the *tcp_server* running, unless that node is also going to run the **lp** commands or the **lpd** process.

## 3. Commands

All of these commands, their options, and any arguments are explained fully on the *DOMAIN/IX Command Reference* Manual and the *DOMAIN/IX Programmer's Reference* Manual pages that are included with this update package.

### 3.1. lpd — line printer daemon

The program **lpd**(8), usually invoked at boot time from the */sys/node_data/etc.rc* file, acts as a master server for coordinating and controlling the spooling queues configured in the */etc/printcap* file. When **lpd** is started, it makes a single pass through the */etc/printcap* database, restarting any printers which have jobs. In normal operation, **lpd** listens for service requests on an Internet socket (under the "printer" service specification) for requests for printer access; see **socket** (2) and **services** (5) for more information on sockets and service specifications, respectively. **Lpd** spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with **lpd** using a simple transaction-oriented protocol. Remote clients are authenticated by means of the "privilege port" scheme employed by **rshd** (8C) and **rcmd** (3X). The following table shows the requests that **lpd** understands. In each request, the first byte indicates the "meaning" of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

| Request | Interpretation |
|---|---|
| ^Aprinter\n | check the queue for jobs and print any found |
| ^Bprinter\n | receive and queue a job from another machine |
| ^Cprinter [users ...] [jobs ...]\n | return short list of current queue state |
| ^Dprinter [users ...] [jobs ...]\n | return long list of current queue state |
| ^Eprinter person [users ...] [jobs ...]\n | remove jobs from a queue |

The **lpr** (1) command submits a print job to a local queue and notifies the local **lpd** that there are new jobs in the spooling area. **Lpd** either schedules the job to be printed locally, or in the case of remote printing, attempts to forward the job to the appropriate machine. If the printer cannot be opened or if the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

### 3.2. lpq — show line printer queue

The **lpq** (1) program works recursively backwards, displaying the queue of the machine directly connected to the printer and then the queue(s) of the machine(s) that lead to it. **Lpq** has two forms of output. In the default short format, it gives a single line of output per queued job (first example below). In the long format (second example), it shows the list of files which comprise a job, and their sizes.

```
%lpq
Rank    Owner     Job  Files                 Total Size
1st     cass      18   memo, manual          4997 bytes

% lpq -l
Warning: no daemon present

cass: 1st                                   [job 018apollo]
        memo                                456 bytes
        manual                              4541 bytes
```

If **lpr** is the last command in a pipeline, **lpq** cannot distinguish which files comprised the job. If you request the long format in this case, the legend "(standard input)" is displayed instead of the filenames.

### 3.3. lprm — remove jobs from a queue

The **lprm** (1) command deletes jobs from a spooling queue. If necessary, **lprm** will first kill off a running daemon that is servicing the queue, then restart it after the files are removed. When removing jobs destined for a remote printer, **lprm** acts like **lpq**, except that it first checks locally for jobs to remove and then tries to remove files in other queues off-machine. You must either be the owner of a job, or the super-user, to remove it.

### 3.4. lpc — line printer control program

The **lpc**(8) program is used by the system administrator to control the operation of the line printer system. You must log in as the super-user to use **lpc** and its associated commands. For each line printer configured in */etc/printcap*, **lpc** can:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer daemons.

## 4. Access control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The following strategy is used to maintain protected spooling areas: The spooling area is writable only by a *daemon* user and *daemon* group. The **lpr** program runs setuid *root* and setgid *daemon*. The *root* access is used to read any file required,

verifying accessibility with an **access** (2) call. The group ID is used in setting up proper ownership of files in the spooling area for **lprm**. Control files (*/usr/spool/lpd/*/cf**) in a spooling area are created by the **lpd** process, with daemon ownership and group ownership *daemon*. Their mode is 0660. This insures that control files are not modified by a user and that no user can remove files except with **lprm**. The spooling programs, **lpd**, **lpq**, and **lprm** run setuid *root* and setgid *daemon* to access spool files and printers. **Lpd** uses the same verification procedures as **rshd** (8C) in authenticating remote clients. As we mentioned earlier, the machine name on which an lp user resides must be present in the file */etc/hosts.equiv*.

## 5. Setting up

The majority of the work in setting up is to create the */etc/printcap* file and any printer filters for printers not supported in the distribution system. (The current DOMAIN/IX implementation of **lp** supports only the **lp** printer filter.)

### 5.1. Creating a printcap file

The */etc/printcap* database contains one or more entries per printer. Each printer should have a separate spooling directory; otherwise, jobs will be printed on different printers, depending only on which printer daemon starts first. This section describes how to create entries for printers which do not conform to the default printer description.

#### 5.1.1. Remote printers

Printers which reside on remote hosts should have an empty **lp** entry. For example, the following printcap entry would send output to the printer named "lp" on the machine "vax".

```
lp|default line printer:\
        :lp=:rm=vax:rp=lp:sd=/usr/spool/vaxlpd:
```

The */etc/printcap* entry **rm** is the name of the remote machine to connect to; this name must appear in the */etc/hosts* database, see **hosts** (5). The **rp** capability indicates that the name of the printer on the remote machine is "lp"; in this case, it could be left out, since this is the default value. The **sd** entry specifies */usr/spool/vaxlpd* as the spooling directory instead of the default value of */usr/spool/lpd/lp*

A remote printer, for a machine on your DOMAIN ring, is a printer on another DOMAIN ring, or a printer on a foreign host to which your ring is connected via TCP/IP.

### 5.2. Output filters

Filters are used to handle device dependencies and to perform accounting functions. The output filter **of** is used to filter text data to the printer device when accounting is not used or when all text data must be passed through a filter. It is not intended to perform accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owner's login name, identifying the beginning and ending of jobs, etc. The other filters (if specified) are started for each file printed and perform accounting if there is an **af** entry. If entries for both **of** and one of the other filters are specified, the output

filter is used only to print the banner page; it is then stopped to allow other filters access to the printer.

## 6. Output filter specifications

The filters supplied with DOMAIN/IX 4.2 BSD handle printing and accounting for printers supported by the DOMAIN print command and print server, **/com/prf** and **/com/prsvr**, respectively. For other printers or accounting methods, you may have to create a new filter.

Normally, **lpd** spawns filters, with standard input being the data to be printed, and standard output the printer. Standard error is attached to the **lf** file, for logging errors. A filter must return an exit code of 0 if there were no errors, 1 if the job should be reprinted, and 2 if the job should be discarded. When **lprm** sends a kill signal to the **lpd** process that is controlling printing, it sends a SIGINT signal to all filters and descendents of filters. If necessary, this signal can be trapped by filters that need to perform cleanup operations like deleting temporary files.

The arguments that may be passed to a filter depend on the filter's type. The **of** filter is called with the following arguments.

*ofilter* **-w**width **-l**length

The *width* and *length* values come from the **pw** and **pl** entries in the */etc/printcap* database. The **if** filter is passed the following parameters.

*filter* [ **-c** ] **-w**width **-l**length **-i**indent **-n** login **-h** host accounting_file

The **-c** flag is optional, and only supplied when control characters are to be passed uninterpreted to the printer (when the **-l** option of **lpr** is used to print the file). The **-w** and **-l** parameters are the same as for the of filter. The **-n** and **-h** parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from */etc/printcap*.

All other filters are called with the following arguments:

*filter* **-x**width **-y**length **-n** login **-h** host accounting_file

The **-x** and **-y** options specify the horizontal and vertical page size in pixels (from the **px** and **py** entries in the printcap file). The rest of the arguments are the same as for the **if** filter.

## 7. Line printer Administration

The lpc program controls line printer activity. You must be logged in as the super-user to use **lpc**. The command format and other commands are described in **lpc**(8).

**abort** and **start**

> **Abort** terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by **lpr**). This is normally used to forcibly restart a hung line printer daemon (i.e., **lpq** reports that there is a daemon present but nothing is happening). It does not remove any jobs from the queue (use the **lprm** command instead). **Abort** only operates on a machine that is running the **lpd** process.

In addition, if you run **lpc** on a different node than the one that is running **lpd**, abort may kill the wrong process. If the node running **lpc** has a process with the same process ID as the process printing on the node running **lpd**, an **abort** will kill the first, or local, process, rather than the printing one.

**Start** enables printing and requests **lpd** to start printing jobs.

**enable** and **disable**

**Enable** and **disable** allow spooling in the local queue to be turned on and off. This will allow or prevent **lpr** from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the **root** user can still use **lpr** to put jobs in the queue, but no one else can. The other common use is to prevent users from putting jobs in the queue when the printer may be unavailable for a long time.

**restart**

**Restart** allows you to restart printer daemons when **lpq** reports that there is no daemon present.

**stop**

**Stop** is used to halt a spooling daemon after the current job completes; this also disables printing. This is a clean way to shutdown a printer in order to perform maintenance, etc. Note that users can still enter jobs in a spool queue while a printer is stopped.

**topq**

**Topq** places jobs at the top of a printer queue. This can be used to reorder high priority jobs since **lpr** normally provides first-come-first-serve ordering of jobs.

## 8. Troubleshooting

There are a number of messages which may be generated by the line printer system. This section categorizes the most common and explains the cause for their generation. Where the message indicates a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer. This would be one of the names from the */etc/printcap* database.

### 8.1. LPR

**lpr:** *printer***: unknown printer**

The **printer** was not found in the */etc/printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the */etc/printcap* file.

**lpr:** *printer***: jobs queued, but cannot start daemon.**

The connection to **lpd** on your DOMAIN ring failed. This usually means the printer server started at boot time has died or is hung. Check the file */usr/spool/lpd/servername* to see which machine should be running **lpd**, then verify that **lpd** is running on that machine, with the **ps**(1) command.

If the file does not exist, at least one TCP host on the ring must be running **lpd**. Use the command **/bin/hostname** to find out the TCP host name of your node, and make sure that the name is in the */etc/hosts.equiv* file. Then start **/usr/lib/lpd** on your machine.

If the ps command shows **lpd** daemons, but they seem to be hung, do the **following**. Get a list of process identifiers of running **lpd**'s by typing

% **ps ax | fgrep lpd**

on the machine that is supposed to run **lpd**. The **lpd** to kill is the one which is not listed in any of the "lock" files. The lock file is contained in the spool directory of each printer (*/usr/spool/lpd/**). Kill the master daemon using the following command.

% **kill** *pid*

where *pid* is the process ID number of the **lpd** process, as reported by the **ps** command. Then restart the daemon (and printer) with the following command.

% **/usr/lib/lpd**

Another possibility is that the **lpr** program is not setuid *root*, setgid *daemon*. This can be checked with

% **ls -lg /bin/lpr**

**lpr:** *printer***: printer queue is disabled**

This means the queue was turned off with

% **lpc disable printer**

to prevent **lpr** from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with **lpc**.

## 8.2. LPQ

**waiting for** *printer* **to become ready (offline ?)**

The printer device could not be opened by the daemon. This can happen for a number of reasons, the most common being that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, etc. The actual reason depends on the meaning of error codes returned by system device driver. Not all printers supply sufficient information to distinguish when a printer is off-line or having trouble (e.g., a printer connected through a serial line). Another possible cause of this message is that some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with **lpc**.

*printer* **is ready and printing**

The **lpq** program checks to see if a daemon process exists for **printer** and prints the file status. If the daemon is hung, a super-user can use **lpc** to

abort the current daemon and start a new one.

**waiting for** *host* **to come up**

This indicates there is a daemon trying to connect to the remote machine named **host** in order to send the files in the local queue. If the remote machine is up, **lpd** on the remote machine is probably dead or hung and should be restarted as mentioned for **lpr**.

**sending to** *host*

The files should be in the process of being transferred to the remote host. If not, the local daemon should be aborted and started with **lpc**.

**Warning:** *printer* **is down**

The printer has been marked with **lpc** as being unavailable.

**Warning: no daemon present**

The **lpd** process overseeing the spooling queue, as indicated in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an **lpd**, use

% **lpc restart printer**

This error might also be reported if **lpq** is not run on the same machine as **lpd**.

## 8.3. LPRM

**lprm:** *printer***: cannot restart printer daemon**

This case is the same as when **lpr** prints that the daemon cannot be started.

## 8.4. LPD

The **lpd** program can write many different messages to the error log file (the file specified in the **lf** entry in */etc/printcap*). Most of these messages are about files which can not be opened and usually implicate the */etc/printcap* file or imply that the protection modes of the files are not correct. Files may also be inaccessible if people manually manipulate the line printer system (i.e., bypass the lpr program).

In addition to messages generated by **lpd**, any of the filters that **lpd** spawns may also log messages to this file.

## 8.5. LPC

**couldn't start printer**

This case is the same as when **lpr** reports that the daemon cannot be started.

**cannot examine spool directory**

Error messages beginning with "cannot ..." are usually due to incorrect ownership and/or protection mode of the lock file, spooling directory or the **lpc** program.

## 8.6. General TCP/IP Error Conditions

**Socket: I/O Error**

This is a general indication of problems with the TCP connection. Restarting */sys/tcp/tcp_server*, on either your machine or the one running **lpd** (if they are different), or both, is the first thing you should try. If this doesn't work, check that any TCP/IP links you've set up are pointing to the correct place, and that all the machines that must communicate are up.

# Appendix L

# Line Printer Management for DOMAIN/IX *sys5*

# Appendix L: Line Printer Management for DOMAIN/IX sys5

## 1.1 General

The line printer (**lp**) system is a set of commands that perform various spooling functions for the *sys5* implementation of DOMAIN/IX. Since the primary **lp** application is off-line printing, this document will focus mainly on spooling to line printers. **Lp** allows administrators to customize the system to spool to a collection of line printers of any type, as well as to group printers into logical classes. Users can:

- Queue and cancel print requests
- Prevent and allow queuing to devices
- Start and stop lp from processing requests
- Change configuration of printers
- Find status of the lp system.

This appendix describes the role of an **lp** administrator in configuring and administering **lp** for a DOMAIN/IX *sys5* **lp** system.

Throughout this appendix, any reference of the form **name**(1M) refers to an entry in the *DOMAIN/IX Programmer's Reference for System V*; the form **name**(1) refers to an entry for name in the *DOMAIN/IX Command Reference for System V*.

## 1.2 Overview

### 1.2.1 Definitions

Several terms must be defined before we present a brief summary of the **lp** commands.

**Lp** makes a distinction between printers and printing devices. A *device* is a physical peripheral device or a file and is represented by a full system pathname. A *printer* is a logical name that represents a device. At different points in time, a *printer* may be associated with different devices. A *class* is a name given to an ordered list of printers; if you specify a *class* when you submit an **lp** request, the request will print on the first available printer in that class. Every class must contain at least one printer, but classes themselves are optional. Each printer may be a member of zero or more classes.

A *destination* is a printer or a class. One destination may be designated as the 'system default destination.' The **lp**(1) command will direct all output to this destination unless the user specifies otherwise. Output directed to a specific printer will not print on another printer if the one you specified is unavailable.

Each invocation of **lp** creates an output request that consists of the files to be printed and options from the **lp** command line. An interface program which formats requests must be supplied for each printer. The **lp** scheduler, **lpsched**(1M), services requests for all destinations by routing requests to interface programs to perform the printing on devices. An **lp** configuration consists of devices, destinations, and interface programs.

### 1.2.2 Generally Available Commands

The **lp**(1) command is used to request that a file or files be printed. It creates an output request and returns a request id of the form

*dest-seqno*

to the user, where *seqno* is a unique sequence number across the entire **lp** system and *dest* is the destination to which the request was routed.

**Cancel** cancels **lp** requests. The user specifies request ids returned by **lp**, or printer names, in which case, the requests currently printing on those printers are cancelled.

**Disable** prevents **lpsched** from routing output requests to printers.

**Enable**(1) allows **lpsched** to route output requests to printers.

### 1.2.3 Commands for lp Administrators Only

Many of the restricted **lp** functions require that the administrator be logged in as either the super-user or as **lp**. All lp files and commands are owned by **lp** except for **lpadmin** and **lpsched** which are owned by **root**. The following commands will be described in more detail later in this appendix.

| | |
|---|---|
| **lpadmin**(1M) | Modifies the **lp** configuration. Many features of this command cannot be used when **lpsched** is running. |
| **lpsched**(1M) | Routes output requests to interface programs which perform the actual printing on devices. |
| **lpshut**(1M) | Stops **lpsched**. All printing activity is halted, but other **lp** commands may still be used. |
| **accept**(1M) | Allows **lp** to accept output requests for destinations. |
| **reject**(1M) | Prevents **lp** from accepting requests for destinations. |
| **lpmove**(1M) | Moves output requests from one destination to another. Whole destinations may be moved at one time. This command cannot be used when **lpsched** is running. |

## 1.3 Setup

Since many of the administrative functions require that you be logged in with the username **lp**, a log-in account for **lp** must exist on your system. If there is not already such an account, you should create one before you attempt to configure the system.

The **lp** subsystem is made up of the **lp** commands mentioned above, and a directory, */usr/spool/lp*. The administrative commands reside in the directory */usr/lib*, the user-level commands reside in */usr/bin*, and the configuration information and spool directories reside in */usr/spool/lp*. The */usr/spool/lp* directory contains the following files and directories.

```
total 64
-rwxrwxrwx   1   root   573384   Dec 9    08:14   FIFO
-rw-r--r--   1   lp          4   Dec 6    13:39   SCHEDLOCK
drwxr-xr-x   1   lp       1024   Nov 25   15:47   class
-rw-r--r--   1   lp          2   Dec 2    11:30   default
drwxr-xr-x   1   lp       1024   Dec 2    11:07   interface
-rw-r--r--   1   lp        689   Dec 9    07:25   log
drwxr-xr-x   1   lp       1024   Dec 2    11:07   member
drwxrwxrwx   1   lp       1024   Nov 25   16:18   model
-rw-r--r--   1   lp        992   Nov 30   15:42   oldlog
-rw-r--r--   1   lp        376   Dec 9    08:13   outputq
-rw-r--r--   1   lp        800   Dec 9    08:13   pstatus
-rw-r--r--   1   lp        600   Dec 9    08:13   qstatus
drwxr-xr-x   1   lp       1024   Dec 2    11:07   request
-rw-r--r--   1   lp          3   Dec 9    07:23   seqfile
```

Since you may alter the files in this directory in the course of configuring the **lp** system for your network, we suggest that, before you begin, you make a backup copy of this entire directory.

Once you've done that, add the following lines to your */sys/node_data/etc.rc* file:

```
rm -f /sys5/usr/spool/lp/SCHEDLOCK
/sys5/usr/lib/lpsched
echo "lp scheduler started"
```

This starts the **lp** scheduler each time that the system is restarted.
**NOTE:** The spooling directory (*/usr/spool/lp*) and the **lp** scheduler (*/usr/lib/llpschedFR*) must run on the same DOMAIN node.

## 1.4 Configuring lp with the lpadmin Command

If you must change the **lp** configuration, use the **lpadmin** command. **Lpadmin** will not attempt to alter the **lp** configuration while **lpsched** is running, except where explicitly noted below.

### 1.4.1 Introducing New Destinations

The following information must be supplied to **lpadmin** when you introduce a new printer to the system:

1. The printer name (**-p***printer*) is an arbitrary name which must conform to the following rules:
   - It must be no longer than 14 characters.
   - It must consist solely of alphanumeric characters and underscores.
   - It must not be the name of an existing **lp** destination (printer or class).
2. The device associated with the printer (**-v***device*). This is the pathname of a hard-wired printer, a log-in terminal, or other file that is writable by **lp**. If a DOMAIN **prf** model is specified, *device* should be set to */dev/null*.
3. The printer interface program. This may be specified in one of three ways:
   - It may be selected from a list of model interfaces supplied with **lp** (**-m***model*).
   - It may be the same interface that an existing printer uses (**-e***printer*).
   - It may be a program supplied by the **lp** administrator (**-i***interface*).

Information which need not always be supplied when creating a new printer includes:

1. The user may specify **-h** to indicate that the device for the printer is hardwired or the device is the name of a file (this is assumed by default). If the device is the pathname of a log-in terminal, then **-l** must be included on the command line. This indicates to **lpsched** that it must automatically disable this printer each time **lpsched** starts running. This fact is reported by **lpstat** when it indicates printer status. (The **a** specifies the printer name.)

   ```
   $ lpstat -pa
   printer a (log-in terminal) disabled Oct 31 11:15 -
         disabled by scheduler: log-in terminal
   ```

   This is done because device names for log-in terminals can be (and usually are) associated with different physical devices from day to day. If the scheduler did not take this action, somebody might log in and be surprised that **lp** is spooling to his/her terminal!
2. The new printer may be added to an existing class or added to a new class (**-c***class*). New class names must conform to the same rules for new printer names.

The following examples will be referenced by further examples in later sections.

1. Create a printer called pr1 whose device is */dev/sio1* and whose interface program is the model hp interface:

```
$ /usr/lib/lpadmin -ppr1 -v/dev/sio1 -mhp
```

2. Add a printer called pr2 whose device is */dev/sio2* and whose interface is a variation of the model prx interface. It is also a log-in terminal:

```
$ cp /usr/spool/lp/model/prx xxx
  < edit xxx >
$ /usr/lib/lpadmin -ppr2 -v/dev/sio2 -ixxx -l
```

3. Create a printer called pr3 that will print to a Spinwriter served by the DOMAIN **prf** print facility. The pr3 will be added to a new class called cl1 and will use the *spin* interface.

```
$ /usr/lib/lpadmin -ppr3 -v/dev/null -mspin -ccl1
```

### 1.4.2 Modifying Existing Destinations

You always modify an existing destination with respect to its printer name (**-p***printer*). Modifications may be one or more of the following:

1. The device for the printer may be changed (**-v***device*). If this is the only modification, then this may be done even while **lpsched** is running. This facilitates changing devices for log-in terminals.
2. The printer interface program may be changed (**-m***model*, **-e***printer*, **-i***interface*).
3. The printer may be specified as hardwired (**-h**) or as a log-in terminal (**-l**).
4. The printer may be added to a new or existing class (**-e***class*).
5. The printer may be removed from an existing class (**-r***class*). Removing the last remaining member of a class causes the class to be deleted. No destination may be removed if it has pending requests. In that case, use **lpmove** or **cancel** to move or delete the pending requests.

These examples are based on the **lp** configuration created in the previous examples.

1. Add printer pr2 to class cl1:

```
$ /usr/lib/lpadmin -ppr2 -ccl1
```

2. Change pr2's interface program to the model prx interface and add it to a new class called cl2:

```
$ /usr/lib/lpadmin -ppr2 -mprx -ccl2
```

   Note that printers pr2 and pr3 now use different interface programs even though pr3 was originally created with the same interface as pr2. Printer pr2 is now a member of two classes.

3. Specify printer pr2 as a hard-wired printer:

```
$ /usr/lib/lpadmin -ppr2 -h
```

4. Add printer pr1 to class cl2:

   $ **/usr/lib/lpadmin -ppr1 -ccl2**

   The members of class cl2 are now pr2 and pr1, in that order. Requests routed to class cl2 will be serviced by pr2 if both pr2 and pr1 are ready to print; otherwise, they will be printed by the one which is next ready to print.

5. Remove printers pr2 and pr3 from class cl1:

   $ **/usr/lib/lpadmin -ppr2 -rcl1**
   $ **/usr/lib/lpadmin -ppr3 -rcl1**

   Since pr3 was the last remaining member of class cl1, the class is removed.

6. Add pr3 to a new class called cl3.

   $ **/usr/lib/lpadmin -ppr3 -ccl3**

### 1.4.3 Specifying the System Default Destination

You can change the system default destination even when **lpsched** is running.

1. Establish class cl1 as the system default destination:

   $ **/usr/lib/lpadmin -dcl1**

2. Establish no default destination:

   $ **/usr/lib/lpadmin -d**

### 1.4.4 Removing Destinations

You can remove a class or a printer only if it has no pending requests. Pending requests can be cancelled with **cancel** or moved to another destination with **lpmove** before the first destination may be removed. If the class or printer you remove is the system default destination, then the system will have no default destination until you explicitly define a new one. When the last remaining-member of a class is removed, then the class is also removed. Removing a class does not imply that the printers in that class were removed.

1. Make printer pr1 the system default destination:

   $ **/usr/lib/lpadmin -dpr1**

   Remove printer pr1:

   $ **/usr/lib/lpadmin -xpr1**

   Now there is no system default destination.

2. Remove printer pr2:

   $ **/usr/lib/lpadmin -xpr2**

   Class cl2 is also removed since pr2 was its only member.

3. Remove class cl3:

   $ **/usr/lib/lpadmin -xcl3**

   Class cl3 is removed, but printer pr3 remains.

## 1.5 Making an Output Request with the lp Command

Once **lp** destinations have been created, users may request output by using the **lp** command. You use the request id that is returned to see if the request has been printed or to cancel the request.

The **lp** program determines the destination of a request by checking the following list in order:

- If the user specifies **-d***dest* on the command line, then the request is routed to *dest*.
- If the environment variable LPDEST is set, the request is routed to the value of LPDEST.
- If there is a system default destination, then the request is routed there.
- The request is rejected.

1. Print two copies of file abc on printer xyz and title the output "my file":

   $ **pr abc | lp -dxyz -n2 -t"my file"**

2. Print file xxx on a Diablo* 1640 printer called zoo in 12-pitch and write to the user's terminal when printing has completed:

   lp -dzoo -ol2 -w xxx

   In this example, "12" is an option that is meaningful to the model Diablo 1640 interface program that prints output in 12-pitch mode [see **lpadmin**(1M)].

## 1.6 Finding lp Status with lpstat

The **lpstat** command is used to find status information about **lp** requests, destinations, and the scheduler.

1. List the status of all pending output requests made by this user:

   $ **lpstat**

   The status information for a request includes the request id, the logname of the user, the total number of characters to be printed, and the date and time the request was made.

---

* **Registered trademark of Xerox Corporation**

2. List the status of printers p1 and p2:

   $ **lpstat -pp1,p2**

### 1.7 Cancelling Requests with cancel

Lp requests may be cancelled using the **cancel** command. Two kinds of arguments may be given to the command—request ids and printer names. The requests named by the request ids are cancelled and requests that are currently printing on the named printers are cancelled. Both types of arguments may be intermixed.

If the user that is canceling a request is not the same one that made the request, then mail is sent to the owner of the request. **Lp** allows any user to cancel requests in order to eliminate the need for users to find **lp** administrators when unusual output should be purged from printers.

### 1.8 Allowing and Refusing Requests—accept and reject

When a new destination is created, **lp** will reject requests that are routed to it, until the **lp** administrator invokes the accept command to allow **lp** to accept requests for that destination.

Sometimes it is necessary to prevent **lp** from routing requests to destinations. If a printer has been removed or is being repaired, or if there are too many requests for it, you may wish to make **lp** reject requests for those destinations, using the **reject** command. Use the **accept** command to allow requests again.

The acceptance status of destinations is reported by the **-a** option of **lpstat**.

1. Cause **lp** to reject requests for destination xyz:

   $ **/usr/lib/reject -r"printer xyz needs repair" xyz**

   Any users that try to route requests to xyz will encounter the following:

   $ **lp -dxyz file**
   lp: can not accept requests for destination "xyz"
   -- printer xyz needs repair

2. Allow **lp** to accept requests routed to destination xyz:

   $ **/usr/lib/accept xyz**

### 1.9 Allowing or Inhibiting Printing—enable and disable

The **enable** command allows the **lp** scheduler to print requests on printers. That is, the scheduler routes requests only to the interface programs of enabled printers. Note that it is possible to enable a printer and at the same time prevent further requests from being routed to it.

The **disable** command will undo the effects of the **enable** command. It prevents the scheduler from routing requests to printers, regardless of whether **lp** is allowing them to accept requests. Printers may be disabled for several reasons including malfunctioning hardware, paper jams, and end-of-day shutdowns. If a printer is busy at the time it is disabled, then the request that was printing will be reprinted in its entirety either on another printer (if the request was originally routed to a class of printers) or on the same one when the printer is reenabled. The **-c** option causes the currently printing requests on busy printers to be cancelled, in addition to disabling the printers. This is useful if strange output is causing a printer to behave abnormally.

1. Disable printer xyz because of a paper jam:

   $ **disable -r"paper jam" xyz**
   printer "xyz" now disabled

2. Find the status of printer xyz:

   $ **lpstat -pxyz**
   printer "xyz" disabled since Jan 5 10:15 -
   paper jam

3. Now, reenable xyz:

   $ **enable xyz**
   printer "xyz" now enabled

### 1.10 Moving Requests Between Destinations with lpmove

Occasionally, it is useful for **lp** administrators to move output requests between destinations. For instance, when a printer is down for repairs, it is desirable to move all of its pending requests to a working printer. Use the **lpmove** command for this. **Lpmove** will refuse to move requests while the **lp** scheduler is running.

1. Move all requests for printer abc to printer xyz:

   $ **/usr/lib/lpmove abc xyz**

   All of the moved requests are renamed, from abc-nnn to xyz-nnn. As a side effect, destination abc will no longer accept requests.

2. Move requests zoo-543 and abc-1200 to printer xyz:

   $ **/usr/lib/lpmove zoo-543 abc-1200 xyz**

   The two requests are now renamed xyz-543 and xyz-1200.

### 1.11 Stopping and Starting the Scheduler: lpshut and lpsched

**Lpsched** is the program that routes output requests made with **lp** through the appropriate printer interface programs to be printed on line printers. Each time the scheduler routes a request to an interface program, it records an entry in the log file, */usr/spool/lp/log*. This entry

contains the logname of the user that made the request, the request id, the name of the printer that the request is being printed on, and the date and time that printing first started. In the case where a request has been restarted, there may be more than one entry in the log file that refers to the request. The scheduler also records error messages in the log file. When **lpsched** is started, it renames */usr/spool/lp/log* to */usr/spool/lp/oldlog* and starts a new log file.

No printing will be performed by the **lp** system unless **lpsched** is running. Use the command

$ **lpstat -r**

to check the status of the **lp** scheduler.

**Lpsched** is normally started at boot time as described above and continues to run until the node is shut down. The scheduler operates in the */usr/spool/lp* directory. When it starts running, it will exit immediately if a file called *SCHEDLOCK* exists. Otherwise, it creates this file in order to prevent more than one scheduler from running at the same time.

Occasionally, it is necessary to shut down the scheduler in order to reconfigure **lp** or to rebuild the **lp** software. The command

$ **/usr/lib/lpshut**

causes **lpsched** to stop running and terminates all printing activity. All requests that were actually being printed will be reprinted in their entirety when the scheduler is restarted.

To restart the **lp** scheduler, use the command

$ **/usr/lib/lpsched**

Shortly after this command is entered, **lpstat** should report that the scheduler is running. If not, a previous invocation of **lpsched** exited without removing *SCHEDLOCK*, so try the following:

$ **rm -f /usr/spool/lp/SCHEDLOCK**
$ **/usr/lib/lpsched**

The scheduler should be running now.

### 1.12 Printer Interface Programs

Every **lp** printer must have an interface program which does the actual printing on the device that is currently associated with the printer. Interface programs may be shell procedures, C programs, or any other executable program. The **lp** model interfaces are all written as shell procedures and can be found in the */usr/spool/lp/model* directory. At the time **lpsched** routes an output request to a printer P, the interface program for P is invoked in the directory */usr/spool/lp* as follows:

**interface/P** *id user title copies options file ...*
where
*id* is the request id returned by **lp**
*user* is logname of user who made the request
*title* is optional title specified by the user
*copies* is number of copies requested by user
*options* is a blank-separated list of class or printer-dependent options specified by user
*file* is the full pathname of a file to be printed

The following examples are requests made by user "smith" with a system default destination of printer "xyz". Each example lists an **lp** command line followed by the corresponding command line generated for printer xyz's interface program:

1. **$ lp /etc/passwd /etc/group**
   *interface/xyz xyz-52 smith "" 1 "" /etc/passwd /etc/group*
2. **$ pr /etc/passwd | lp -t"users" -n5**
   *interface/xyz xyz-53 smith users 5 ""*
   */usr/spool/lp/request/xyz/d0-53*
3. **$ lp /etc/passwd -oa -ob**
   *interface/xyz xyz-54 smith "" 1 "a b" /etc/passwd*

When the interface program is invoked, its standard input comes from */dev/null* and both the standard output and standard error output are directed to the printer's device. Devices are opened for reading as well as writing when file modes permit. In the case where a device is a regular file, all output is appended to the end of the file.

Several interface models have been provided to support the DOMAIN **prf** facility. These interfaces (**p**, **ge**, and **spin**) do not write to the standard output, but pipe data to the **/com/prf** command. These models may be customized to create new interfaces to support a variety of devices with the **prf** command.

Given the command line arguments and the output directed to a device, interface programs may format their output in any way they choose. Interface programs must ensure that the proper stty modes (terminal characteristics such as baud rate, output options, etc.) are in effect on the output device. This may be done in a shell interface only if the device is opened for reading:

$ **stty mode ... <&1**

That is, take the standard input for the stty command from the device.

When printing has completed, it is the responsibility of the interface program to exit with a code indicative of the success of the print job. Exit codes are interpreted by **lpsched** as follows:

| CODE | MEANING TO LPSCHED |
|---|---|

| | |
|---|---|
| 0 | The print job has completed successfully. |
| 1 to 127 | A problem was encountered in printing this particular request (e.g., too many nonprintable characters). This problem will not affect future print jobs. **Lpsched** notifies users by mail that there was an error in printing the request. |
| greater than 127 | These codes are reserved for internal use by **lpsched**. Interface programs must not exit with codes in this range. |

When problems that are likely to affect future print jobs occur (e.g., a device filter program is missing), the interface programs should disable printers so that print requests are not lost. When a busy printer is disabled, the interface program will be terminated with signal 15.