

HP-UX Selected Articles

HP-UX
HP-UX
HP-UX
HP-UX
HP-UX
HP-UX

HP-UX Selected Articles

for the HP 9000 Series 200/500

Manual Part No. 97089-90003

© Copyright 1983, Hewlett-Packard Company.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

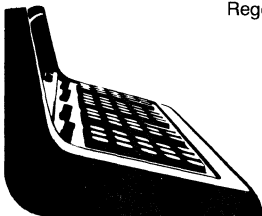
Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

© Copyright 1980, Bell Telephone Laboratories, Inc.

© Copyright 1979, 1980, The Regents of the University of California.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.



Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

November 1983...First Edition

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Fort Collins Systems Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

* For other countries, contact your local Sales and Support Office to determine warranty terms.

Preface

The articles contained in this manual are provided to help you use the commands and utilities provided with HP-UX. The articles have several sources. Some were written at Hewlett-Packard specifically for the HP 9000 family of computers. Others were written at Bell Laboratories or the University of California at Berkeley (UCB), and remain in their original state. Thus, it is possible that some options and/or descriptions of command behavior may not apply to your system.

The articles included are:

1. System Overview
2. Edit: A Tutorial
3. Ex Reference Manual
4. The Vi Editor
5. The Ed Editor
6. Sed – A Non-Interactive Text Editor
7. Awk – A Pattern Scanning and Processing Language
8. Shell Programming
9. UNIX* Programming
10. Make – A Program for Maintaining Computer Programs
11. Source Code Control System User's Guide
12. Using C on the HP 9000 Series 500 Computers
13. Lint – C Program Checker
14. Nroff/Troff User's Manual
15. Memorandum Macros
16. Lex – A Lexical Analyzer Generator
17. Yacc: Yet Another Compiler-Compiler
18. Uucp Implementation Description
19. Using the System Console with HP 9000 Series 200 Computers
20. HP-UX and the HP 9000 Model 520 as System Console
21. MC68000 Assembler on HP-UX

*UNIX is a trademark of Bell Laboratories, Inc.

Table of Contents

Introduction and System Overview for Series 500 Computers

The Pieces	1
Where are the Pieces Located?.....	3
How Do the Pieces Fit?	4
Where Can I Learn More?.....	5

Introduction and System Overview for the Series 500 Computers

HP-UX is a powerful and flexible operating system, providing many tools for developing and running application programs. Supplied with HP-UX is, seemingly, a mountain of material. Where do you start? How do you learn to use the system?

This section provides an overview of the system by describing its parts, their functions, and the methods in which they interact. It also provides a guide through the documents supplied with HP-UX, explaining where you may find detailed information about its various components.

The Pieces

HP-UX is composed of several functional “pieces”, each of which is described in the following paragraphs.

- **The kernel** - the heart of the operating system. It is a program that controls the allocation of system resources. For example, it allocates memory to run programs, schedules programs for execution, allocates computer (CPU) time among running programs, and takes care of the technical intricacies of communicating with peripheral devices.

The kernel is automatically loaded and run when the computer is powered-up (generally the responsibility of the system administrator). Unless you are the system administrator, this should already be done for you by the time you are ready to use the system.

- **Commands** - executable programs performing specific tasks. This includes the programs supplied with HP-UX as well as the ones that you create.

Some commands are simple, performing a specific function with little or no user interaction. For example, the *who* command, when executed, prints the user name of each user currently logged in (accessing the system). Other commands are more complex, continually interacting with you to perform their tasks. For example, the *ed* command, when executed, allows you to create a text file from the characters entered from the keyboard. It has several subprograms (such as *append*, *replace*, *insert*, etc.) that are accessed by supplying certain “key characters” to the main program.

- **Shell** - a program (command) that acts as your interface to HP-UX. It is automatically run when you successfully log in (gain access to the system). The shell’s main purpose is to wait for information to be typed on the keyboard. Once the information is entered, it passes the information on to the kernel as the name of a program to be executed. It also transfers any optional data and parameters entered with the program name.

Beyond its job as a simple command interpreter, the shell also provides its own programming language. This language includes control-flow constructs (such as *for - next*, *while*, and *if - then - else*). The shell’s programming language is used primarily for writing shell scripts (described next).

- **Script or shell script** - text file containing a series of program names and shell programming language constructs. When executed, a shell script can replace typing from the keyboard (and thus free your time for other activities) by providing command names and parameters to the shell for execution. Since its language provides control-flow statements, it can examine the output of one command and decide which command(s) to execute next.

For example, suppose that you want to see both a list of everyone using the system and a list identifying the tasks that each user is performing. You could execute the individual commands *who* (prints a list of everyone using the system) and *ps* (prints a list identifying each task being

performed on the system). However, if this operation is to be performed often, you would spend more time typing than necessary. By creating a shell script (described in the Shell Programming section later in this manual) that contains both commands, you would only have to execute the script to obtain the desired information. The *whodo* command is a script performing exactly this function. It also includes many commands to format and present the data in a more concise form.

- **Subroutine** - a sequence of computer instructions for performing a specific task. A subroutine can only be used by a program (that is, it cannot act as a free-standing program). A subroutine can be used repeatedly in one or more programs, thus allowing you to use the same subroutine each time the function is needed. There is no need to write or even enter the program code to perform the task; you need only use the name of the existing subroutine to obtain its function.

Subroutines are provided with your system to keep you from having to re-invent the code that performs a function. For example, suppose that while writing a program, you found that you needed an arctangent function. You could write your own arctangent function using the math functions available in your programming language. Alternately, you could simply call the **atan** subroutine (provided with HP-UX) from your program.

- **Library** - a collection of related subroutines. For example, a math library might include trigonometric functions, logarithmic functions, and numeric base conversion functions. A math library and an I/O library are included with the standard libraries supplied with HP-UX. Other libraries may be supplied with the various programming languages and application packages you purchase.
- **System call** - a “link” or “hook” into the capabilities provided by the HP-UX kernel. Many of the base level capabilities used in the kernel are available for use in your programs. Functionally, the system call is similar to the subroutine. Both are previously written routines which you may use in your application programs. Subroutines are stored in disc files while system calls reside in the kernel.
- **Language** - a programming language, such as C, FORTRAN, and Pascal. Each language actually consists of at least one command (a compiler for the language) and usually one or more libraries. The compiler translates a text file (assuming it has the expected form and syntax) into binary code which the computer can understand and execute.
- **Ordinary file** - a file containing a program or data (binary code or ASCII text) such as a command or shell script.
- **Special file** - a file defining the attributes of a peripheral device, such as the communication protocol and the location of a disc drive or a printer. When output is directed to or input is directed from a special file, the kernel uses the information in the special file to communicate with the peripheral device. The complexities of actual device to device communication are left to the kernel. The program operates independently of the device(s) with which it communicates.
- **Directory (file)** - an ordinary file containing a list of ordinary files, special files, and other directories. Directories are used to organize the files that form the HP-UX hierarchical file system.

Where are the Pieces Located?

The pieces that form HP-UX (previously described) are located at various places in the system. The other documents supplied with your system describe the organization of the HP-UX file system and teach its use and methods of access. This section simply identifies the location of the major pieces that form HP-UX.

The kernel is stored in a special part (called the **boot area**) of a mass storage medium. At computer power-up, it is automatically loaded by the loader - a program that permanently resides in the computer.

The commands and libraries are stored in ordinary files on the disc. Their location is specified by the directory in which they are located (as described by the text and diagram that follow).

Commands are distributed between three directories. The most commonly used commands (such as *ls*, *who*, and *ed*) are stored in the */bin* directory. Less frequently used commands (such as *get*, *delta*, and *lint*) are stored in the */usr/bin* directory.

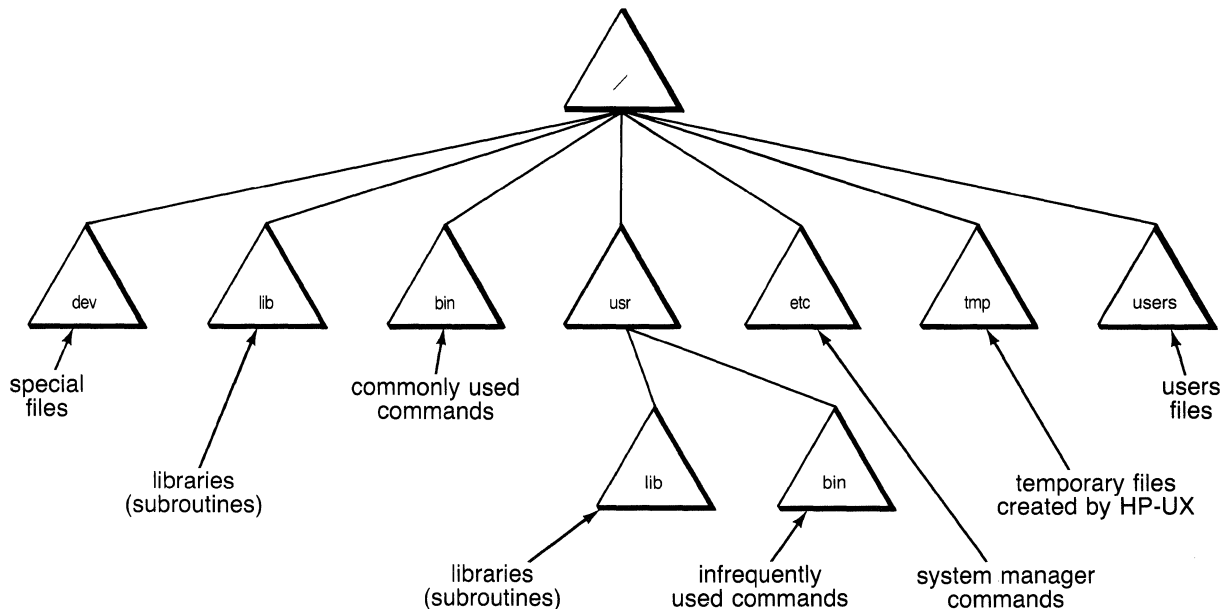
Commands located in the */etc* directory are typically used for system management or system maintenance. Often, these commands can only be accessed by the system manager or super-user. (The super-user is a system user with special capabilities; he is identified by a special user ID.)

The libraries supplied with HP-UX are located in the directories */lib* and */usr/lib*.

Special files are located in the directory */dev*.

The files and directories that you and other users create are usually stored in the directory */users*. Typically, each user has his own directory in */users*. That directory, in turn, contains the directories and ordinary files that he has created.

Location of Features in the HP-UX File System



How Do the Pieces Fit?

Now that you are familiar with the various pieces of HP-UX, you are probably saying to yourself “That’s nice. But how do all of the pieces fit together?”. The easiest way to show this is through an example. Let’s follow Joe Programmer through his task of creating and running a program:

1. If the system is not loaded and running, Joe (or his system administrator) switches on power to the computer. The computer’s system loader program finds and loads the HP-UX kernel.
2. Once the kernel is loaded, some initial set-up is performed and then the *login* program is run. When the program is ready, it displays the prompt:

```
login:
```

Joe types in his user name, which he obtained from the system administrator. He is then prompted for his password (assuming Joe has been wise enough to assign himself a password). Once his user name and password have been correctly entered, Joe is logged in.

3. The system displays some information (such as the message of the day) and then automatically runs the shell program. This allows Joe to execute any program he wishes. He executes the *mail* command to send his boss a message informing him that he is three days ahead of schedule “because of the incredible power of HP-UX”. He knows that the message will be received when his boss logs in.

Next, Joe executes the *ls* command to see what files are present in his directory. He executes the *rm* command to remove a file containing an early version of his project report that he no longer needs.

4. Now Joe starts to write a Pascal program to plot test data (for quality assurance) from last month’s Widget production. He executes his favorite text editing program (such as *vi* or *ed*) to create the new program. He enters the text which forms the program from his terminal:

```
PROGRAM PLOT_DATA (INPUT,OUTPUT);
(*PLOTS Q/A TEST DATA FROM WIDGET PRODUCTION*)

VAR
    NUM_PRODUCED, NUM_FAILED : INTEGER;

    ...

END.
```

His program probably includes calls to HP-UX libraries and to the libraries supplied with the Pascal Programming Language (possibly to provide math functions). Additionally, it may include HP-UX system calls (calls to the intrinsics).

5. Once he is satisfied with the program, he instructs the text editing program to save (write) his program in a file named “plot_data.p”. He then terminates the text editing program.
6. Next, Joe converts the text file into executable code by executing the *pc* command (the Pascal compiler). The compiler checks the file for correct syntax and, catching an error, informs Joe that his program contains an error. Joe again executes the text editing program, instructing it that he wishes to edit the text contained in the file “plot_data.p”. Once the error is corrected, he re-saves the file and terminates the editing program.
7. Once more Joe executes the compiler and this time finds that no syntax errors are found. Executing the *mv* command, he moves the compiled code to a file named “plot_it”.

8. Finally, Joe executes his program, directing its output to the special file `/dev/plt9872` (a special file created by the system manager for accessing the HP 9872 Plotter). He enters:

```
Plot_it > /dev/plt9872 
```

Assuming Joe is a good programmer, he finds his data plotted on the HP 9872. Pleased with himself, he logs off (terminates his login session) by pressing while holding the key depressed.

The following prompt verifies that he is no longer accessing the system and that the terminal is ready for the next user (possibly you?) to login:

```
login:
```

Running More Than One Task

Although HP-UX provides virtual memory support for both code and data, this support does not allow a task to be completely swapped from main memory. A minimum of 16 Kbytes of main memory is consumed by each task until it terminates. Therefore, it is possible when multiple tasks are competing for main memory to get an out-of-memory message (usually: “not enough memory” or “MEMORY FAULT”). The task that failed to get the needed memory is killed, while other tasks proceed normally. A task that was killed by the system can be restarted at any future time, but preferably when the system is less loaded.

If You Get an Error

The `err` command has been included to help you obtain more information about why an error is occurring. If you encounter an error that you believe is a bug, execute `/bin/err` immediately after the error occurs. This will list three numbers that you should record and use in describing the problem to your support engineer. Note that the numbers are updated as each occurs. Be sure to record them before another error happens.

Where Can I Learn More?

Now that you are acquainted with the pieces that form HP-UX and have seen how the pieces are used in a typical application, you are probably wondering where you can learn more about HP-UX. The best way to learn HP-UX is to attend an HP-UX training class. Contact your HP sales representative for more information about the courses.

Whether or not you attend a training course, you should read the documents supplied with HP-UX. Where do you start? The following list describes the documents shipped with HP-UX and indicates the order in which they should be read.

1. If your computer has not yet been installed or if you want to verify that the computer is operating properly, read the **Installation and Test** manual supplied with your computer. A list of the materials supplied with your computer is supplied in the **Unpacking Instructions for the HP 9000 Series 500 Computers** (HP part number 97080-90092), as well as a “roadmap” briefly describing the documents supplied with the system.
2. If you are responsible for installing HP-UX, read the **System Administrator Manual** (HP part number 97089-90047). This manual provides instructions for installing HP-UX.

3. For a basic understanding of HP-UX, read the article entitled "Introduction and System Overview" in the **HP-UX Selected Articles** manual (HP part number 97089-90003). It introduces you to some fundamental terms and provides an overview of the system. Additionally, it includes a detailed description of the documents provided with HP-UX and indicates the order in which they should be read.
4. To start learning how to use HP-UX, read the softcover text **Introducing the UNIX¹ System** (HP part number 98680-90025). You will need to obtain a user name (and optionally, a password) from your system administrator so that you can try the interactive examples in the text. Working with the system is the easiest way to learn its use.

If you are the system administrator and are the first user on the system, use the user name `root` when logging in (when you are so directed by the text). Otherwise, obtain a user name and password from your system administrator before working the examples in the text.

Once you know how to use the system, the order in which the remaining manuals are read depends on the task(s) you need to accomplish and the options (such as programming languages) purchased with HP-UX. Select and read only the documents that apply.

5. If you are responsible for installing, managing and maintaining HP-UX (for example, adding users to the system, backing-up the file system, and adding peripherals to the system) read the **System Administrator Manual** (HP part number 97089-90047). It describes the system administrator's job and his responsibilities. It also explains the concepts of HP-UX that are needed to manage and maintain the system. Additionally, it provides instructions for performing the specific tasks that are the system administrator's responsibility (in the chapter entitled "System Administrator's Toolbox").
6. To learn how to use a specific application program (such as the *vi* or *ed* text editor programs), read the appropriate articles in the manual, **HP-UX Selected Articles** (HP part number 97089-90003).
7. The **HP-UX Reference** (HP part number 09000-90006) provides syntax and semantic information about the HP-UX commands, system calls, subroutines, and data files. It also provides some limited tutorial information about device access and complex procedures. Be sure to read the section entitled "Introduction". It describes in detail the contents of the manual and provides tutorial information about accessing HP-UX from your terminal or computer keyboard.
8. To learn how to write shell scripts with the shell's programming language, read the article entitled "Shell Programming" in the manual, **HP-UX Selected Articles** (HP part number 97089-90003).
9. To learn how to write programs in the C Programming Language, read the softcover text, **C Programming Language** (HP part number 97089-90000).
 - a. Once you know how to program in C, you may want to access the power of HP-UX directly from C. The article entitled "UNIX Programming" in the **HP-UX Selected Articles** manual (HP part number 97089-90003) shows you how to access the HP-UX system calls and subroutines from a C-Language program.
10. For a description of the features provided with the FORTRAN Programming Language, read the **FORTRAN Reference Manual** (HP part number 97081-90001).

¹ UNIX is a Trademark of Bell Telephone Laboratories, Inc.

11. For a description of the features provided with the Pascal Programming Language, read the **Pascal Reference Manual** (HP part number 97082-90001).
12. To learn how to access the Device-independent Graphics Library from a program, you should read:
 - **DGL Device Handlers Manual** (HP part number 97085-90005)
 - **GRAPHICS/9000 DGL Programmer's Reference Manual** (HP part number 97084-90000)
 - **GRAPHICS/9000 DGL Supplement for HP-UX Systems** (HP part number 97084-90002).
13. To learn how to configure or use the data communications utilities, read **HP-UX Asynchronous Communications Guide** (HP part number 97076-90001).

Table of Contents

Edit: A Tutorial

Abstract	1
Session 1: Creating a File of Text	4
Asking for Edit	4
The "Command not found" Message	5
A summary	5
Entering Text	5
Messages from Edit	5
Text Input Mode	6
Writing Text to Disk	6
Logging Off	7
Session 2	8
Adding More Text to Your File	8
Interrupt	8
Making Corrections	8
Listing What's in the Buffer	9
Finding Things in the Buffer	9
The Current Line	10
Numbering Lines (nu)	10
Substitute Command (s)	10
Another Way to List What's in the Buffer	12
Saving the Modified Text	12
Session 3	13
Moving Text in the Buffer	13
Copying Lines (copy)	14
Deleting Lines (d)	14
A Word or Two of Caution	15
Undo (u) to the Rescue	15
More About the Dot (.) and Buffer End (\$)	16
Moving Around in the Buffer (+ and -)	16
Changing Lines (c)	17
Session 4	18
Making Commands Global (g)	18
More About Searching and Substituting	19
Special Characters	19
Issuing UNIX Commands From the Editor	20
Filenames and File Manipulation	20
The File (f) Command	20
Reading Additional Files (r)	21
Writing Parts of the Buffer	21
Recovering Files	21
Other Recovery Techniques	21
Further Reading and Other Information	22
Using Ex	22



Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex*, was designed to provide an informative environment for new and casual users.

This edition documents Version 2 of *edit* and *ex*.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

August 31, 1980

†UNIX is a trademark of Bell Laboratories.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

Text editing using a terminal connected to a computer allows one to create, modify, and print text easily. A specialized computer program, known as a *text editor*, assists in creating and revising text. Creating text is very much like typing on an electric typewriter. Modifying text involves telling the text editor what to add, change, or delete. Text is printed by giving a command to print the file contents, with or without special instructions as to the format of the desired output.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which will lead you through the fundamental steps of creating and revising a file of text. After scanning each lesson and before beginning the next, you should follow the examples at a terminal to get a feeling for the actual process of text editing. Set aside some time for experimentation, and you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading "Communicating with UNIX" or one of the other tutorials which provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with your terminal and its special keys, the login procedure, and the ways of correcting typing errors. Let's first define some terms:

- program A set of instructions given to the computer, describing the sequence of steps which the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.
- UNIX UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.
- edit *edit* is the name of the UNIX text editor which you will be learning to use, a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor named *ex*.
- file Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file it is kept until you instruct the system to remove it. You may create a file during one UNIX session, log out, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, and another might contain a very long document or program. The only way to save information from one session to the next is to write it to a file, storing it for later use.
- filename Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.

disk Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to the coating on magnetic recording tape, on which information is recorded.

buffer A temporary work space, made available to the user for the duration of a session of text editing and used for building and modifying the text file. We can imagine the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1: Creating a File of Text

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure.

If the terminal you are using is directly linked to the computer, turn it on and press carriage return, usually labelled "RETURN". If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. Press carriage return once and await the login message:

`:login:`

Type your login name, which identifies you to UNIX, on the same line as the login message, and press carriage return. If the terminal you are using has both upper and lower case, be sure you enter your login name in lower case; otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types "`:login:`" and you reply with your login name, for example "susan":

`:login: susan` *(and press carriage return)*

(In the examples, input typed by the user appears in **bold face** to distinguish it from the responses from UNIX.)

UNIX will next respond with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it to prevent others from seeing it. The message is:

`Password:` *(type your password and press carriage return)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX will respond with

`Login incorrect.`

`:login:`

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % is the UNIX prompt symbol which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with *edit*, the text editor. Now is a convenient time to choose a name for the file of text which you are about to create. To begin your editing session type *edit* followed by a space and then the filename which you have selected, for example "text". When you have completed the command, press carriage return and wait for *edit*'s response:

```
% edit text (followed by a carriage return)
"text" No such file or directory
:
```

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. As we expected, it was unable to find such a file since "text" is the name of the new file that we will create. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that edit expects a command from you. You may now begin to create the new file.

The "Command not found" message

If you misspelled edit by typing, say, "editor", your request would be handled as follows:

```
% editor
editor: Command not found.
%
```

Your mistake in calling edit "editor" was treated by UNIX as a request for a program named "editor". Since there is no program named "editor", UNIX reported that the program was "not found." A new % indicates that UNIX is ready for another command, so you may enter the correct command.

A summary

Your exchange with UNIX as you logged in and made contact with edit should look something like this:

```
:login: susan
Password:
... A Message of General Interest ...
% edit text
"text" No such file or directory
:
```

Entering text

You may now begin to enter text into the buffer. This is done by *appending* text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, you are creating text. Most edit commands have two forms: a word which describes what the command does and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append" which may be abbreviated "a". Type **append** and press carriage return.

```
% edit text
: append
```

Messages from edit

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of

“append” or “a”, you will receive this message:

```
:add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new “:” appeared to let you know that edit is again ready to receive a command.

Text input mode

By giving the command “append” (or using the abbreviation “a”), you entered *text input mode*, also known as *append mode*. When you enter text input mode, edit responds by doing nothing. You will not receive any prompts while in text input mode. This is your signal that you are to begin entering lines of text. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press carriage return*. When you give this signal that you want to stop appending text, you will exit from text input mode and reenter command mode. Edit will again prompt you for a command by printing “:”.

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and carriage return.

This is as good a place as any to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

Let’s say that the lines of text you enter are (try to type **exactly** what you see, including “thiss”):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
```

The last line is the period followed by a carriage return that gets you out of append mode. If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer to “Communicating with UNIX” if you need to review the procedures for making a correction. Erasing a character or cancelling a line must be done before the line has been completed by a carriage return. We will discuss changes in lines already typed in session 2.

Writing text to disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor’s buffer is temporary and will last only until the end of the editing session. Thus, learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command “write” (or its abbreviation “w”):

:write

Edit will copy the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a "New file" will be noted. The newly-created file will be given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines which were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit will print

No current filename

in response to your write command. If this happens, you can specify the filename in a new write command:

:write text

After the "write" (or "w") type a space and then the name of the file.

Logging off

We have done enough for this first lesson on using the UNIX text editor, and are ready to quit the session with edit. To do this we type "quit" (or "q") and press carriage return:

```
:write
"text" [New file] 3 lines, 90 characters
:quit
%
```

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal we also need to exit from UNIX. In response to the UNIX prompt of "% " type the command **logout** or a "control d". This is done by holding down the control key (usually labelled "CTRL") and simultaneously pressing the d key. This will end your session with UNIX and will ready the terminal for the next user. It is always important to logout at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Session 2

Login with UNIX as in the first session:

```
:login: susan (carriage return)
Password: (give password and carriage return)
%
```

This time when you say that you want to edit, you can specify the name of the file you worked on last time. This will start edit working and it will fetch the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

```
% edit text
"text" 3 lines, 90 characters
:
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions. In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. When append is the first command of your editing session, the lines you enter are placed at the end of the buffer. We'll soon discuss why this happens. Here we'll use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

Interrupt

Should you press the RUBOUT key (sometimes labelled DELETE) while working with edit, it will send this message to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by rubout or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text that you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

If you have read a general introduction to UNIX, such as "Communicating with UNIX", you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase. Accounts normally start out using the number sign (#) as the erase character, but it's possible for a different erase character to be selected†. We'll show "#" as the erase character in our

†UNIX accounts may be "personalized" in other ways, too. If you're using an established account, check with someone who is familiar with your account to find out if it has any other non-standard characteristics which may affect your work. Accounts for students in classes are often given class commands and other special features; the teaching assistant or instructor is the best source of information about these changes.

examples, but if you've changed your erase character to backspace (control-H) or something else, be sure to use your own erase character.

If you make a bad start in a line and would like to begin again, erasing individual characters with a "#" is cumbersome — what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex#####
```

with no room for the great text you'd like to type, or,

```
This is yukky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. (An account may select a different line erase character to use in place of @. If your line erase character has been changed, use it where the examples show "@".) You may immediately begin to retype the line. This, unfortunately, does not help after you type the line and press carriage return. To make corrections in lines which have been completed, it is necessary to use the editing commands covered in this session and those that follow.

Listing what's in the buffer

Having appended text to what you wrote in Lesson 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The "1" stands for line 1 of the buffer, the "\$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. Thus, "1,\$p" gives you:

```
This is some sample text.
And this is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may enter into the buffer a character which can't be printed, which is done by striking a key while the CTRL key is depressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidentally holding down the CTRL key while typing "a". Edit would display

```
it does illustr^Ate the editor.
```

if you asked to have the line printed. To represent the control-A, edit shows "^A". The sequence "" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the "". We'll soon discuss the commands which can be used to correct this typing error.

In looking over the text we see that "this" is typed as "thiss" in the second line, as suggested. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find "thiss" in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for "thiss" and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

`:/thiss/`

By typing `/thiss/` and pressing carriage return edit is instructed to search for "thiss". If we asked edit to look for a pattern of characters which it could not find in the buffer, it would respond "Pattern not found". When edit finds the characters "thiss", it will print the line of text for your inspection:

And thiss is some more text.

Edit is now positioned in the buffer at the line which it just printed, ready to make a change in the line.

The current line

At all times during an editing session, edit keeps track of the line in the buffer where it is positioned. In general, the line which has been most recently printed, entered, or changed is considered to be the current position in the buffer. The editor is prepared to make changes at the current position in the buffer, unless you direct it to act in another location. When you bring a file into the editor, you will be positioned at the last line in the file. If your initial editing command is "append", the lines you enter are added to the end of the file, that is, they are placed after the current position. You can refer to your current position in the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage return you will be instructing edit to print the current line:

`:.
.`
And thiss is some more text.

If you want to know the number of the current line, you can type `.=` and carriage return, and edit will respond with the line number:

`:.=
2`

If you type the number of any line and a carriage return, edit will position you at that line and print its contents:

`:2
.`
And thiss is some more text.

You should experiment with these commands to assure yourself that you understand what they do.

Numbering lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

`:nu
2` And thiss is some more text.

Notice that the shortest abbreviation for the number command is "nu" (and not "n" which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, "1,Snu" lists all lines in the buffer with the corresponding line numbers.

Substitute command (s)

Now that we have found our misspelled word it is time to change it from "thiss" to "this". As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append*, so *s* stands for *substitute*. We will use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

2s/thiss/this/

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them and then a closing slash mark. To summarize:

2s/ what is to be changed / what to change to /

If edit finds an exact match of the characters to be changed it will make the change **only** in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

Substitute pattern match failed

indicating your instructions could not be carried out. When edit does find the characters which you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/  
And this is some more text.  
:
```

line 2 (and line 2 only) will be searched for the characters "thiss", and when the first exact match is found, "thiss" will be changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

```
:s/thiss/this/
```

edit will assume that we mean to change the line where we are currently positioned ("."). In this case, the command without a line number would have produced the same result because we were already positioned at the line we wished to change.

For another illustration of substitution we may choose the line:

Text editing is strange, but nice.

We might like to be a bit more positive. Thus, we could take out the characters "strange, but " so the line would read:

Text editing is nice.

A command which will first position edit at that line and then make the substitution is:

```
:/strange/s/strange, but //
```

What we have done here is combine our search with our substitution. Such combinations are perfectly legal. This illustrates that we do not necessarily have to use line numbers to identify a line to edit. Instead, we may identify the line we want to change by asking edit to search for a specified pattern of letters which occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit we want to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but //". If you do not indicate the space is to be taken out, your line will be:

Text editing is nice.

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands can be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command `z`. If you type

```
:1z
```

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, give the command

```
:z
```

If no starting line number is given for the `z` command, printing will start at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as paging. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "`q`" to quit the session your dialogue with edit will be:

```
:q  
No write since last change (q! quits)  
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you have done during the editing session since the latest write command. Since in this lesson we have not written to disk at all, everything we have done would be lost. If we did not want to save the work done during this editing session, we would have to type "`q!`" to confirm that we indeed wanted to end the session immediately, losing the contents of the buffer. However, since we want to preserve what we have edited, we need to say:

```
:w  
"text" 6 lines, 171 characters
```

and then,

```
:q  
% logout
```

and hang up the phone or turn off the terminal when UNIX asks for a login name. This is the end of the second session on UNIX text editing.

Session 3

Bringing text into the buffer (e)

Login to UNIX and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you say

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by saying:

```
: e text  
"text" 6 lines, 171 characters
```

The command **edit**, which may be abbreviated "e" when you're in the editor, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move (m)** command:

```
: 2,4m$
```

This command directs edit to move lines 2, 3, and 4 to the end of the buffer (\$). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. Thus,

```
: 1,6m20
```

would instruct edit to move lines 1 through 6 (inclusive) to a position after line 20 in the buffer. To move only one line, say, line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
: 5,$m1  
2 lines moved  
it does illustrate the editor.
```

After executing a command which changes more than one line of the buffer, edit tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.

We can restore the original order by typing:

```
:4,$m1
```

or, combining context searching and the move command:

```
:/And this is some/,/This is text/m/This is some sample/
```

The problem with combining context searching with the move command is that the chance of making a typing error in such a long command is greater than if one types line numbers.

Copying lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:12,15copy $
```

makes a copy of lines 12 through 15, placing the added lines after the buffer's end (S). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is "co" (and **not** the letter "c" which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by "delete" or "d". This example deletes line 4:

```
:4d
```

```
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line which has become the current line ("").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
```

```
This is text added in Session 2.
```

```
:d
```

```
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the next line which contains the indicated text. Once you are sure that you have correctly specified the line that you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line (""), that is, the line found by our search. After the deletion, your buffer should contain:

This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.

To delete both lines 2 and 3:

And this is some more text.
Text editing is nice.

you type

```
:2,3d
```

which specifies the range of lines from 2 to 3, and the operation on those lines – “d” for delete.

Again, this presumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command as so:

```
:/And this is some/,/Text editing is nice./d
```

This tells the editor to start deleting with the next line that contains the characters “And this is some” and continue until it has deleted the line containing “Text editing is nice.”

A word or two of caution:

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited – that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing carriage return to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command. To undo the previous command, type “u” or “undo”. Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. It is possible to undo only commands which have the power to change the buffer, for example delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e) which interact with disk files cannot be undone, nor can commands such as print which do not change the buffer. Most importantly, the **only** command which can be reversed by undo is the last “undo-able” command which you gave.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines which were formerly numbered 2 and 3. Executing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u  
2 more lines in file after undo  
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now “dot” (the current line).

More about the dot (.) and buffer end (\$)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode we type dot (and only a dot) on a line and press carriage return;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

Thus if we type “.=” we are asking for the number of the line and if we type “.” we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign (S=) edit will print the line number corresponding to the last line in the buffer.

“.” and “\$” therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,$d
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

Moving around in the buffer (+ and -)

It is frequently convenient during an editing session to go back and re-read a previous line. We could specify a context search for a line we want to read if we remember some of its text, but if we simply want to see what was written a few, say 3, lines ago, we can type

```
-3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. We can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line which is 2 ahead of our current position.

You may use “+” and “-” in any command where edit accepts line numbers. Line numbers specified with “+” or “-” can be combined to print a range of lines. The command

```
:-1,+2copy$
```

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines will be placed after the last line in the buffer (\$).

Try typing only “-”; you will move back one line just as if you had typed “-1p”. Typing the command “+” works similarly. You might also try typing a few plus or minus signs in a row (such as “+++”) to see edit’s response. Typing a carriage return alone on a line is the equivalent of typing “+1p”; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a “+” or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

```
At end-of-file
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

Nonzero address required on this command
Negative address — first buffer line is 1

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

There may be occasions when you want to delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs edit to delete specified lines and then switch to text input mode in order to accept the text which will replace them. Let's say we want to change the first two lines in the buffer:

This is some sample text.
And this is some more text.

to read

This text was created with the UNIX text editor.

To do so, you can type:

```
:1,2c  
2 lines changed  
This text was created with the UNIX text editor.  
.  
:
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After a carriage return enters the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. You will remain in text input mode until you exit in the usual way, by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands which apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer — the **global (g)** command.

To print all lines containing a certain sequence of characters (say, “text”) the command is:

```
:g/text/p
```

The “g” instructs edit to make a global search for all lines in the buffer containing the characters “text”. The “p” prints the lines found.

To issue a global command, start by typing a “g” and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word “text” to the word “material” the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the “g” at the end of the global command which instructs edit to change each and every instance of “text” to “material”. If you do not type the “g” at the end of the command only the *first* instance of “text” in each line will be changed (the normal result of the substitute command). The “g” at the end of the command is independent of the “g” at the beginning. You may give a command such as:

```
:14s/text/material/g
```

to change every instance of “text” in line 14 alone. Further, neither command will change “Text” to “material” because “Text” begins with a capital rather than a lower-case *t*.

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a “p” at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other — in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d  
72 less lines in file after global
```

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small buffer of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change “noun” to “nouns” we may type either

```
:/noun/s/noun/nouns/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/noun/s//nouns/
```

In this example, the characters to be changed are not specified — there are no characters, not even a space, between the two slash marks which indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean “use the characters we last searched for as the characters to be changed.”

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/  
It doesn't mean much here, but  
://  
it does illustrate the editor.
```

Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters “does”.

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

It's also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/noun/nouns/
```

we could use the command

```
:/nouns/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters “nouns”.

Special characters

Two characters have special meanings when used in specifying searches: “\$” and “^”. “\$” is taken by the editor to mean “end of the line” and is used to identify strings which occur at the end of a line.

```
:g/ing$/s//ed/p
```

tells the editor to search for all lines ending in “ing” (and nothing else, not even a blank space), to change each final “ing” to “ed” and print the changed lines.

The symbol “^” indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert “1.” and a space at the beginning of the current line.

The characters “\$” and “^” have special meanings only in the context of searching. At

other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s\$/dollar/
```

looks for the character "\$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "\$" would have represented "the end of the line" in your search, rather than the character "\$". The backslash retains its special significance unless it is preceded by another backslash.

Issuing UNIX commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command *rm* to remove the file named "junk" type:

```
!:rm junk
```

```
!
```

```
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a UNIX command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed. The editor prints a "!" when the command is completed. The tutorial "Communicating with UNIX" describes useful features of the system, of which the editor is only one part.

Filename and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename*. Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. You can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
```

```
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command unless it is the first filename given in the editing session*. Thus, in the next write command which does not specify a name, edit will write onto the current file and not onto the file "chapter3".

The file (f) command

To ask for the current filename, type **file** (or **f**). In response, the editor provides current information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
```

```
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading additional files (r)

The **read (r)** command allows you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text will be placed, the command *r*, and then the name of the file.

```
:Sr bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the read command unless it is the first filename given in the editing session.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending*. The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer.

Recovering files

Under most circumstances, edit's crash recovery mechanism is able to save work to within a few lines of changes after a crash or if the phone is hung up accidentally. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login which gives the name of the recovered file. To recover the file, enter the editor and type the command **recover (rec)**, followed by the name of the lost file.

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve (pre)**, which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

and then seek help. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. After a preserve, you can use the recover command once the problem has been corrected.

If you make an undesirable change to the buffer and issue a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual*, which is applicable to both *ex* and *edit*. The manual is available from the Computer Center Library, 218 Evans Hall. One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that *edit* continues to meet your needs. However, should you become interested in using *ex*, it is easy to switch. To begin an editing session with *ex*, use the name *ex* in your command instead of *edit*.

Edit commands work the same way in *ex*, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In *edit*, only the characters “^”, “\$”, and “\” have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in *ex*, as described in the *Ex Reference Manual*. Another feature of the *edit* environment prevents users from accidentally entering two alternative modes of editing, *open* and *visual*, in which the editor behaves quite differently than in normal command mode. If you are using *ex* and the editor behaves strangely, you may have accidentally entered open mode by typing “o”. Type the ESC key and then a “Q” to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

This tutorial was produced at the Computer Center of the University of California, Berkeley. We welcome comments and suggestions concerning this item and the UNIX documentation in general.

Table of Contents

Ex Reference Manual

Starting Ex.....	1
File Manipulation	2
Current File	2
Alternate File	2
Filename Expansion	2
Multiple Files and Named Buffers	2
Read Only	2
Exceptional Conditions	3
Errors and Interrupts	3
Recovering from Hangups and Crashes	3
Editing Modes	3
Command Structure	3
Command Parameters	4
Command Variants	4
Flags After Commands	4
Comments	4
Multiple Commands per Line	4
Reporting Large Changes	4
Command Addressing	4
Addressing Primitives	4
Combining Addressing Primitives	5
Command Descriptions	5
Regular Expressions and Substitute Replacement Patterns	14
Regular Expressions	14
Magic and Nomagic	14
Basic Regular Expression Summary	14
Combining Regular Expression Primitives	15
Substitute Replacement Patterns	15
Option Descriptions	15
Limitations	19
Update to Ex Reference Manual	21
Command Line Options	21
Commands	21
Options	21
Environment Enquiries	22
Vi Tutorial Update	22
Deleted Features	22
Change in Default Option Settings	22
Vi Commands	22
Macros	23



Ex Reference Manual

Version 3.5/2.13 — September, 1980

William Joy

*Revised for versions 3.5/2.13 by
Mark Horton*

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

1. Starting *ex*

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. If there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor will seek the description of the terminal in that file (rather than the default /etc/termcap.) If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable, otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* will be executed before each editor session.

A command to enter *ex* has the following prototype:†

```
ex [ - ] [ -v ] [ -t tag ] [ -r ] [ -l ] [ -wn ] [ -x ] [ -R ] [ +command ] name ...
```

The most common case edits a single file with no options, i.e.:

```
ex name
```

The *-* command line option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The *-v* option is equivalent to using *vi* rather than *ex*. The *-t* option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The *-r* option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The *-l* option sets up for editing LISP, setting the *showmatch* and *lisp* options. The *-w* option sets the default window size to *n*, and is useful on dialups to start in small windows. The *-x* option causes *ex* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt(1)*. The *-R* option sets the *readonly* option at the start. ‡ *Name* arguments indicate files to be edited. An argument of the form *+command* indicates that the editor should begin by

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets '[' ']' surround optional parameters here.

‡ Not available in all v2 editors due to memory constraints.

executing the specified command. If *command* is omitted, then it defaults to “\$”, positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form “/pat” or line numbers, e.g. “+100” starting at line 100.

2. File manipulation

2.1. Current file

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited*. This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not *edited* then *ex* will not normally write on it if it already exists.*

2.2. Alternate file

Each time a new value is given to the current file name, the previous current file name is saved as the *alternate* file name. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file name.

2.3. Filename expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character ‘%’ in filenames is replaced by the *current* file name and the character ‘#’ by the *alternate* file name.†

2.4. Multiple files and named buffers

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list*. The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z*.‡

2.5. Read only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidentally overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the *-R* command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. It is possible to write, even while in read only mode, by indicating that you really

* The *file* command will say “[Not edited]” if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

‡ It is also possible to refer to *A* through *Z*; the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

3. Exceptional Conditions

3.1. Errors and interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

3.2. Recovering from hangups and crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the `-r` option. If you were editing the file *resume*, then you should change to the directory where you were when the crash occurred, giving the command

```
ex -r resume
```

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

```
ex -r
```

will print a list of the files which have been saved for you. (In the case of a hangup, the file will not appear in the list, although it can be recovered.)

4. Editing modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append*, *insert*, and *change* commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi*.

5. Command structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

* As an example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

5.1. Command parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command “10p” will print the tenth line in the buffer while “delete 5” will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name.‡

5.2. Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an ‘!’ immediately after the command name. Some of the default variants may be controlled by options; in this case, the ‘!’ serves to toggle the default.

5.3. Flags after commands

The characters ‘#’, ‘p’ and ‘l’ may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, ‘p’ is rarely necessary. Any number of ‘+’ or ‘-’ characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: “. Any command line beginning with “ is ignored. Comments beginning with “ may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

5.5. Multiple commands per line

More than one command may be placed on a line by separating each pair of commands by a ‘|’ character. However the *global* commands, comments, and the shell escape ‘!’ must be the last command on a line, as they are not terminated by a ‘|’.

5.6. Reporting large changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an *undo*. After commands with more global effect such as *global* or *visual*, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

6. Command addressing

6.1. Addressing primitives

The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus ‘.’ is rarely used alone as an address.

† Counts are rounded down if necessary.

‡ Examples would be option names in a *set* command i.e. “set number”, a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. “1,5 copy 25”.

** A ‘p’ or ‘l’ must be preceded by a blank or tab except in the single special case ‘dp’.

<i>n</i>	The <i>n</i> th line in the editor's buffer, lines being numbered sequentially from 1.
\$	The last line in the buffer.
%	An abbreviation for "1,\$", the entire buffer.
+n -n	An offset relative to the current buffer line.†
/pat/ ?pat?	Scan forward and backward respectively for a line containing <i>pat</i> , a regular expression (as defined below). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing <i>pat</i> , then the trailing / or ? may be omitted. If <i>pat</i> is omitted or explicitly empty, then the last regular expression specified is located.‡
'' 'x	Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as '''. This makes it easy to refer or return to this previous context. Marks may also be established by the <i>mark</i> command, using single lower case letters <i>x</i> and the marked lines referred to as 'x'.

6.2. Combining addressing primitives

Addresses to commands consist of a series of addressing primitives, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';' the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer.†

7. Command descriptions

The following form is a prototype for all *ex* commands:

address command ! parameters count flags

All parts are optional; the degenerate case is the empty command which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses, which are *not*, however, part of the command.

abbreviate *word rhs*

abbr: **ab**

Add the named abbreviation to the current list. When in input mode in visual, if *word* is typed as a complete word, it will be changed to *rhs*.

(.) append

abbr: **a**

text

Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

† The forms '.+3' '+3' and '+++ ' are all equivalent; if the current line is line 100 they all address line 103.

‡ The forms \ / and \ ? scan using the last regular expression used in a scan; after a substitute // and ?? would scan using the substitute's regular expression.

† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus '.100' is equivalent to '..100'. It is an error to give a prefix address to a command which expects none.

a!

text

.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text*.

args

The members of the argument list are printed, with the current argument delimited by '[' and ']'.

(. , .) **change count**

abbr: c

text

.

Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a *delete*.

c!

text

.

The variant toggles *autoindent* during the *change*.

(. , .) **copy addr flags**

abbr: co

A *copy* of the specified lines is placed after *addr*, which may be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy*.

(. , .) **delete buffer count flags**

abbr: d

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

edit file

abbr: e

ex file

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible† the editor reads the file into its buffer.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited*. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read.‡

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty*, a terminal, or a binary or executable file (as indicated by the first word).

‡ If executed from within *open* or *visual*, the current line is initially the first line of the file.

e! file

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e + n file

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: "+/pat".

file

abbr: f

Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line.*

file file

The current file name is changed to *file* which is considered '[Not edited]'.

(1 , \$) global /pat/ cmds

abbr: g

First marks each line among those specified which matches the given regular expression. Then the given command list is executed with '.' initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. *Append*, *insert*, and *change* commands and associated input are permitted; the '.' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds*. The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global*, (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire *global*. Finally, the context mark "" is set to the value of '.' before the *global* command begins and is not changed during a *global* command, except perhaps by an *open* or *visual* within the *global*.

g! /pat/ cmds

abbr: v

The variant form of *global* runs *cmds* at each line not matching *pat*.

(.) insert

abbr: i

text

.

Places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from *append* only in the placement of text.

* In the rare case that the current file is '[Not edited]' this is noted also; in this case you have to use the form *w!* to write to the file, since the editor is not sure that a *write* will not destroy a file unrelated to the current contents of the buffer.

i!
text

The variant toggles *autoindent* during the *insert*.

(. . .+1) **join** *count flags* abbr: j

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a ')'. If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.

j!

The variant causes a simpler *join* with no white space processing; the characters in the lines are simply concatenated.

(.) **k** *x*

The *k* command is a synonym for *mark*. It does not require a blank or tab before the following letter.

(. . .) **list** *count flags*

Prints the specified lines in a more unambiguous way: tabs are printed as '^I' and the end of each line is marked with a trailing '\$'. The current line is left at the last line printed.

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key n. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See section 6.9 of the "Introduction to Display Editing with Vi" for more details.

(.) **mark** *x*

Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form "x" then addresses this line. The current line is not affected by this command.

(. . .) **move** *addr* abbr: m

The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: n

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes which may have been made.

n *filelist*

n + *command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(. . .) **number count flags** abbr: # or nu
Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open flags** abbr: o

(.) **open /pat/ flags**
Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details.

‡

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

(. . .) **print count** abbr: p or P
Prints the specified lines with non-printing characters printed as control characters '^x'; delete (octal 177) is represented as '^?'. The current line is left at the last line printed.

(.) **put buffer** abbr: pu
Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: q

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit*.† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the q! command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

(.) **read file** abbr: r

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

‡ Not available in all v2 editors due to memory constraints.

* But no modifying commands may intervene between the *delete* or *yank* and the *put*, nor may lines be moved between files without using a named buffer.

† *Ex* will also issue a diagnostic if there are more files in the argument list.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

(.) **read** *!command*

Reads the output of the command *command* into the buffer after the specified line. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename*; a blank or tab before the ! is mandatory.

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command. Except when you use *preserve* you will be notified by mail when a file is saved.

rewind

abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set *nooption*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set*; they are interpreted left-to-right.

shell

abbr: **sh**

A new shell is created. When it terminates, editing resumes.

source *file*

abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(.,.) **substitute** */pat/repl/ options count flags*

abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl*. If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

‡ Within *open* and *visual* the current line is set to the first line read rather than the last.

** The system saves a copy of the file you were editing only if you have made changes to the file.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop!** is used. This command is only available where supported by the teletype driver and operating system.

(. , .) **substitute** *options count flags* abbr: s

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the **&** command.

(. , .) **t** *addr flags*

The **t** command is a synonym for *copy*.

ta tag

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file. ‡

The tags file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag; this field is usually a contextual scan using *'/pat/*' to be immune to minor changes in the file. Such scans are always performed as if *nomagic* was set.

The tag names in the tags file must be sorted alphabetically. ‡

unabbreviate *word* abbr: una

Delete *word* from the list of abbreviations.

undo abbr: u

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual*.) Also, the commands *write* and *edit* which interact with the file system cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line '.' as '''. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual* the current line regains its pre-command value after an *undo*.

unmap lhs

The macro expansion associated by *map* for *lhs* is removed.

(1 , \$) **v** */pat/ cmds*

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line which does not match *pat*.

version abbr: ve

Prints the current version number of the editor as well as the date the editor was last changed.

‡ If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* will reuse the previous tag.

‡ Not available in all v2 editors due to memory constraints.

(.) **visual** *type count flags*

abbr: **vi**

Enters visual mode at the specified line. *Type* is optional and may be ‘-’, ‘↑’ or ‘.’ as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

visual *file*

visual + *n* *file*

From visual mode, this command is the same as edit.

(1 , \$) **write** *file*

abbr: **w**

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.* If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never changed.

If an error occurs while writing the current and *edited* file, the editor considers that there has been “No write since last change” even if the buffer had not previously been modified.

(1 , \$) **write**>> *file*

abbr: **w>>**

Writes the buffer contents at the end of an existing file.

w! *name*

Overrides the checking of the normal *write* command, and will write to any file which the system permits.

(1 , \$) **w** !*command*

Writes the specified lines into *command*. Note the difference between **w!** which overrides checks and **w !** which writes to a command.

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

The variant overrides checking on the sensibility of the *write* command, as **w!** does.

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(. , .) **yank** *buffer count*

abbr: **ya**

Places the specified lines in the named *buffer*, for later retrieval via *put*. If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

* The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty*, */dev/null*. Otherwise, you must give the variant form **w!** to force the write.

(. +1) *z count*

Print the next *count* lines, default *window*.

(.) *z type count*

Prints a window of text with the specified line at the top. If *type* is '-' the line is placed at the bottom; a '.' causes the line to be placed in the center.* A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single '!' is printed when the command completes.

(*addr* , *addr*) ! *command*

Takes the specified address range and supplies it as standard input to *command*; the resulting output then replaces the input lines.

(\$) =

Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*

(. , .) < *count flags*

Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(. +1 , . +1)

(. +1 , . +1) |

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

* Forms 'z=' and 'z|' also exist; 'z=' places the current line in the center, surrounds it with lines of '-' characters and leaves the current line at this line. The form 'z|' prints the window before 'z=' would. The characters '+', '|', and '-' may be repeated for cumulative effect. On some v2 editors, no *type* may be given.

(. , .) & options count flags

Repeats the previous *substitute* command.

(. , .) ~ options count flags

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '/' or '??'.

8.2. Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic*, the default for *edit*, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic*.†

8.3. Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

<i>char</i>	An ordinary character matches itself. The characters '^' at the beginning of a line, '\$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.
^	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
\$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	Matches any single character except the new-line character.
\<	Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

† To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '^' at the beginning of a regular expression, '\$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

[*string*] Matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by ‘-’ in *string* defines the set of characters collating between the specified lower and upper bounds, thus ‘[a-z]’ as a regular expression matches any (single) lower-case letter. If the first character of *string* is an ‘|’ then the construct matches those characters which it otherwise would not; thus ‘[|a-z]’ matches anything but a lower-case letter (and of course a newline). To place any of the characters ‘|’, ‘|’, or ‘-’ in *string* you must escape them with a preceding ‘\’.

8.4. Combining regular expression primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character ‘*’ to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character ‘~’ may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences ‘\(\’ and ‘\)\’ with side effects in the *substitute* replacement patterns.

8.5. Substitute replacement patterns

The basic metacharacters for the replacement pattern are ‘&’ and ‘~’; these are given as ‘\&’ and ‘\~’ when *nomagic* is set. Each instance of ‘&’ is replaced by the characters which the regular expression matched. The metacharacter ‘~’ stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character ‘\’. The sequence ‘\n’ is replaced by the text matched by the *n*-th regular subexpression enclosed between ‘\(\’ and ‘\)\’.[†] The sequences ‘\u’ and ‘\l’ cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences ‘\U’ and ‘\L’ turn such conversion on, either until ‘\E’ or ‘\e’ is encountered, or until the end of the replacement pattern.

9. Option descriptions

autoindent, ai

default: noai

Can be used to ease the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit ^D. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end-of-file with a ^D.

[†] When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of ‘\(\’ starting from the left.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '^j' and immediately followed by a '^D'. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '^O' followed by a '^D' repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap

default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *t*, *undo* or *shift* command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in *globals*, and only applies to the last of many commands on a line.

autowrite, aw

default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next*, *rewind*, *stop*, *tag*, or *!* command, or a '^j' (switch files) or '^]' (tag goto) command in *visual*. Note, that the *edit* and *ex* commands do not autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for *.I* *rewind* , *stop!* for *stop*, *tag!* for *tag*, *shell* for *!*, and *:e #* and a *:ta!* command from within *visual*).

beautify, bf

default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

directory, dir

default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory is not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible

default: noedcompatible

Causes the presence or absence of *g* and *c* suffixes on *substitute* commands to be remembered, and to be toggled by repeating the suffices. The suffix *r* makes the substitution be as in the *~* command, instead of like *&*. *##*

errorbells, eb

default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht

default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic

default: noic

Version 3 only.

* Bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp
Autoindent indents appropriately for *lisp* code, and the () { } [and] commands in *open* and *visual* are modified to have meaning for *lisp*.

list default: nolist
All printed lines will be displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.

magic default: magic for *ex* and *vt*
If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '^' and '\$' having special effects. In addition the metacharacters '*' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'

mesg default: mesg
Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set. **

number, nu default: nonumber
Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

open default: open
If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt default: optimize
Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP Libp
Specifies the paragraphs for the { and } operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt default: prompt
Command mode input is prompted for with a '^'.

redraw default: noredraw
The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

† *Nomagic* for *edit*.
** Version 3 only.

- remap** default: remap
 If on, macros are repeatedly tried until they are unchanged. **††** For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.
- report** default: report=5†
 Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.
- scroll** default: scroll=1/2 window
 Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode **z** command (double the value of *scroll*).
- sections** default: sections=SHNHH HU
 Specifies the section macros for the **[[** and **]]** operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.
- shell, sh** default: sh=/bin/sh
 Gives the path name of the shell forked for the shell escape command **!**, and by the *shell* command. The default is taken from SHELL in the environment, if present.
- shiftwidth, sw** default: sw=8
 Gives the width a software tab stop, used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.
- showmatch, sm** default: nosm
 In *open* and *visual* mode, when a **)** or **}** is typed, move the cursor to the matching **(** or **{** for one second if this matching character is on the screen. Extremely useful with *lisp*.
- slowopen, slow** terminal dependent
 Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.
- tabstop, ts** default: ts=8
 The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.
- taglength, tl** default: tl=0
 Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

†† Version 3 only.

† 2 for *edit*.

tags default: tags=tags /usr/lib/tags
 A path of files to be used as tag files for the *tag* command. **††** A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system.)

term from environment TERM
 The terminal type of the output device.

terse default: noterse
 Shorter error diagnostics are produced for the experienced user.

warn default: warn
 Warn if there has been '[No write since last change]' before a '!' command escape.

window default: window= speed dependent
 The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600
 These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws
 Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm default: wm=0
 Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.

writeln, wa default: nowa
 Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with *map* to 32, and the total number of characters in macros to be less than 512.

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex*. Bruce Englar encouraged the redesign which led to *ex* version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

†† Version 3 only.

Ex changes — Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 3.1 to 3.5 of *ex*. Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command line options

- 3.4 A new command called *view* has been created. *View* is just like *vi* but it sets *readonly*.
- 3.4 The encryption code from the *v7* editor is now part of *ex*. You can invoke *ex* with the *-x* option and it will ask for a key, as *ed*. The *ed x* command (to enter encryption mode from within the editor) is not available. This feature may not be available in all instances of *ex* due to memory limitations.

Commands

- 3.4 Provisions to handle the new process stopping features of the Berkeley TTY driver have been added. A new command, *stop*, takes you out of the editor cleanly and efficiently, returning you to the shell. Resuming the editor puts you back in command or visual mode, as appropriate. If *autowrite* is set and there are outstanding changes, a write is done first unless you say "stop!".
- 3.4 A
 - `:vi <file>`
 - command from visual mode is now treated the same as a
 - `:edit <file>` or `:ex <file>`
 - command. The meaning of the *vi* command from *ex* command mode is not affected.
- 3.3 A new command mode command *xit* (abbreviated *x*) has been added. This is the same as *wq* but will not bother to write if there have been no changes to the file.

Options

- 3.4 A read only mode now lets you guarantee you won't clobber your file by accident. You can set the on/off option *readonly* (*ro*), and writes will fail unless you use an *!* after the write. Commands such as *x*, *ZZ*, the *autowrite* option, and in general anything that writes is affected. This option is turned on if you invoke *ex* with the *-R* flag.
- 3.4 The *wrapmargin* option is now usable. The way it works has been completely revamped. Now if you go past the margin (even in the middle of a word) the entire word is erased and rewritten on the next line. This changes the semantics of the number given to *wrapmargin*. 0 still means off. Any other number is still a distance from the right edge of the screen, but this location is now the right edge of the area where wraps can take place, instead of the left edge. *Wrapmargin* now behaves much like *fill/nowrap* mode in *nroff*.
- 3.3 The options *w300*, *w1200*, and *w9600* can be set. They are synonyms for *window*, but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with
 - `:set w300=12 w1200=23`
- 3.3 The new option *timeout* (default on) causes macros to time out after one second. Turn it off and they will wait forever. This is useful if you want multi character macros, but if your terminal sends escape sequences for arrow keys, it will be necessary to hit escape twice to get a beep.

- 3.3 The new option *remap* (default on) causes the editor to attempt to map the result of a macro mapping again until the mapping fails. This makes it possible, say, to map `q` to `#` and `#1` to something else and get `q1` mapped to something else. Turning it off makes it possible to map `^L` to `l` and map `^R` to `^L` without having `^R` map to `l`.
- 3.3 The new (string) valued option *tags* allows you to specify a list of tag files, similar to the "path" variable of `csh`. The files are separated by spaces (which are entered preceded by a backslash) and are searched left to right. The default value is `"tags /usr/lib/tags"`, which has the same effect as before. It is recommended that "tags" always be the first entry. On Ernie CoVax, `/usr/lib/tags` contains entries for the system defined library procedures from section 3 of the manual.

Environment enquiries

- 3.4 The editor now adopts the convention that a null string in the environment is the same as not being set. This applies to `TERM`, `TERMCAP`, and `EXINIT`.

Vi Tutorial Update

Deleted features

- 3.3 The "`q`" command from visual no longer works at all. You must use "`Q`" to get to `ex` command mode. The "`q`" command was deleted because of user complaints about hitting it by accident too often.
- 3.5 The provisions for changing the window size with a numeric prefix argument to certain visual commands have been deleted. The correct way to change the window size is to use the `z` command, for example `z5<cr>` to change the window to 5 lines.
- 3.3 The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: "`:map!`".

Change in default option settings

- 3.3 The default window sizes have been changed. At 300 baud the window is now 8 lines (it was 1/2 the screen size). At 1200 baud the window is now 16 lines (it was 2/3 the screen size, which was usually also 16 for a typical 24 line CRT). At 9600 baud the window is still the full screen size. Any baud rate less than 1200 behaves like 300, any over 1200 like 9600. This change makes *vi* more usable on a large screen at slow speeds.

Vi commands

- 3.3 The command "`ZZ`" from *vi* is the same as "`:x<cr>`". This is the recommended way to leave the editor. `Z` must be typed twice to avoid hitting it accidentally.
- 3.4 The command `^Z` is the same as "`:stop<cr>`". Note that if you have an arrow key that sends `^Z` the stop function will take priority over the arrow function. If you have your "susp" character set to something besides `^Z`, that key will be honored as well.
- 3.3 It is now possible from visual to string several search expressions together separated by semicolons the same as command mode. For example, you can say

```
/foo/;/bar
```

from visual and it will move to the first "bar" after the next "foo". This also works within one line.

- 3.3 `^R` is now the same as `^L` on terminals where the right arrow key sends `^L` (This includes the Televideo 912/920 and the ADM 31 terminals.)

- 3.4 The visual page motion commands `^F` and `^B` now treat any preceding counts as number of pages to move, instead of changes to the window size. That is, `2^F` moves forward 2 pages.

Macros

- 3.3 The “mapinput” mechanism of version 3.1 has been replaced by a more powerful mechanism. An “!” can follow the word “map” in the `map` command. `Map!`ed macros only apply during input mode, while `map`’ed macros only apply during command mode. Using “`map`” or “`map!`” by itself produces a listing of macros in the corresponding mode.
- 3.4 A word abbreviation mode is now available. You can define abbreviations with the `abbreviate` command

```
:abbr foo find outer otter
```

which maps “foo” to “find outer otter”. Abbreviations can be turned off with the `unabbreviate` command. The syntax of these commands is identical to the `map` and `unmap` commands, except that the ! forms do not exist. Abbreviations are considered when in visual input mode only, and only affect whole words typed in, using the conservative definition. (Thus “foobar” will not be mapped as it would using “map!”) `Abbreviate` and `unabbreviate` can be abbreviated to “`ab`” and “`una`”, respectively.

Table of Contents

The Vi Editor

Preliminary Notes	1
Creating an Ordinary File	1
Invoking Vi	2
Moving Around in the File	3
Cursor-Positioning Keys	5
Scrolling and Paging	5
Moving From Line to Line	5
Skipping Over Sentences, Paragraphs, and Sections	6
Searching for a Pattern	7
Moving Within a Line	10
Returning to Your Previous Position	11
Adding, Deleting, and Correcting Text	11
Inserting and Appending Text	12
Character Corrections	13
Line Corrections	14
Copying and Moving Text	15
Shifting Lines	17
Continuous Text Input	17
Undoing a Command	17
Special Vi Commands	18
Setting Vi Options	18
Defining Macros	22
Defining Abbreviations	23
Reading Data Into Your Current File	24
Writing Edited Text Onto a File	24
Editing Other Files	25
Editing the Next File in the Argument List	26
Filtering Buffer Text Through HP-UX Commands	27
Vi and Ex	28
The Shell Interface	28
Getting Into Vi	28
Getting Back to the Shell	29
Miscellaneous Topics	30
Vi Initialization	30
Recovering Lost Lines	31
Entering Control Characters in Your Text	31
Adjusting the Screen	31
Printing Your File Status	32
Appendix A: Character Functions	33
Appendix B: Example .exrc File	39

The Vi Editor

Vi is a display-oriented, interactive text editor. The contents of your file are displayed on your screen, so you can see the result of each *vi* command as soon as the command is executed. There is rarely any doubt about the current state of your file.

Preliminary Notes

Vi has two peculiar traits that might prove somewhat confusing to the beginning user. The first is that many of your commands do not print on your terminal when you type them in. Be assured that *vi* is still listening to you, however. If you watch the screen when you type in a command, *vi* usually gives some indication that your command has been received and interpreted. More specifically, the only commands that will print on your terminal are those that begin with /, :, ?, and !. If these characters are embedded in a long string of commands, only those characters after and including one of those above will be printed.

The second trait is that *vi* always uses the bottom line of the screen for command output, error messages, and echoed command lines. This is where you should look for information and command verification.

Creating an Ordinary File

The remainder of this article discusses the various commands and features of the *vi* editor. Because many *vi* commands do not print on the screen when they are executed, it is difficult to represent the results that appear on your screen before and after a command has executed. Thus, this article is designed to be read while you have access to a computer so you can try each command as it is discussed.

To be able to try each command, you need a file with some text in it. To create a file, type

```
$ vi filename
```

where *filename* is the name of the file you are creating. This file name is completely up to you. *Vi* responds by printing

```
"filename" [new file]
```

at the bottom of your screen, and prints a tilde (~) at the beginning of each line on the screen. The tilde is a special character that *vi* uses to mark the end of the text in a file that already exists, or, in the case of a new file, to show that there is currently no text in the file. The tildes are simply markers that are used for your convenience; they do not become part of the text in your file.

You are now ready to put text in your file. To do this, type **a** (for append). Even though the command does not print on your screen, *vi* is now waiting for your text. As you type in your text, note that everything you type appears on your screen, and that the tilde on each line disappears as you begin typing on that line.

It does not really matter what you type in for your text, but you need at least two paragraphs of material (paragraphs must be separated by at least one blank line). That amount of text ensures that most of the commands can be illustrated on your file. When you are done entering text, press [ESC], and exit the editor by typing **ZZ**. You should now have a shell prompt on your screen.

Invoking Vi

Material Covered:

vi file ...	command; invokes <i>vi</i> with one or more file arguments;
[ESC], [ALT], ctrl-[commands; end text insertion or modification;
[DEL], [RUB], ctrl-?	commands; generate an interrupt.

You invoke *vi* the same way you invoke any shell command. *Vi* accepts several options and a list of file names, which are the names of the files you want to create or edit. For a list of the available options, refer to the HP-UX Reference manual. For example,

\$ vi file1 file2 file3

invokes *vi* with file1, file2, and file3 as arguments. File1 is created or edited first. *Vi* remembers file2 and file3 so that you can create or edit them after you are finished with file1. Begin editing the file you created previously by typing

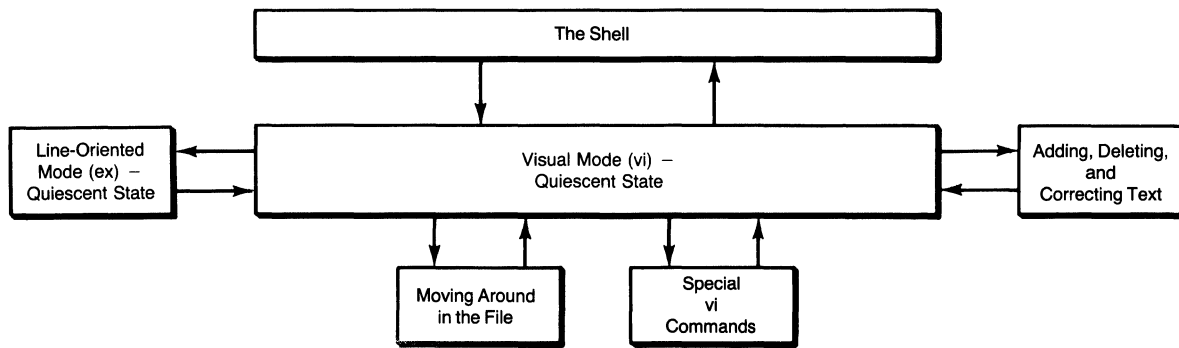
\$ vi filename

where *filename* is the name of the file you created. Note that *vi* prints out either a screenful of text from *filename*, or the entire contents of *filename* followed by a tilde on each remaining empty line. *Vi* does the latter if *filename* does not contain enough text to fill the screen. Your cursor is positioned at the beginning of the first line of the file. *Vi* is now waiting for your commands.

Vi always copies the contents of the file you are editing into a special buffer. All additions, deletions, and corrections are performed on the copy in the buffer. This way, the original file remains unchanged until you are sure you want to change it. Then, when you are finished editing the file, you can tell *vi* to overwrite the previous contents of the file with the revised text in the buffer. Even if you are creating a file, the text you put in your file is actually put in the buffer. The text remains there until you tell *vi* to transfer it to the file you are creating.

Once you have invoked *vi*, it enters a do-nothing state in which it waits for a command. This is called a *quiescent state*. You can determine what state *vi* is in by pressing [ESC] or [DEL]. [ESC] is used to end text insertion and to cancel partially formed commands. If you press [ESC] and *vi* responds by ringing the bell, then *vi* is in a quiescent state. If *vi* does not ring the bell, then it is busy executing a command. Ctrl-[generates the same sequence as the [ESC] or [ALT] key on your keyboard. [DEL] generates an interrupt, which forces *vi* to stop whatever it is doing and return to a quiescent state. The DEL signal can also be generated with ctrl-?.

Once *vi* is in a quiescent state, there are several things you can do. They are shown in the following diagram.



Moving Around in the File

Material Covered:

[↑], k , ctrl-P	commands; move the cursor up one line in the same column;
[→], l , [SPACE]	commands; move the cursor one character to the right;
[↓], j , ctrl-J , ctrl-N	commands; move the cursor down one line in the same column;
[←], h , [BACKSPACE], ctrl-H	commands; move the cursor one character to the left;
ctrl-D	command; scroll down;
ctrl-U	command; scroll up;
ctrl-E	command; scroll up one line;
ctrl-Y	command; scroll down one line;
ctrl-F	command; move forward one page in the file;
ctrl-B	command; move backward one page in the file;
+ , [RETURN], ctrl-M	commands; move the cursor to the first printable character on the next line;
-	command; move cursor to the first printable character on the previous line;
nG	command; move cursor to first printable character on line number <i>n</i> ; default <i>n</i> = last line of the file;
H	command; move cursor to the first printable character of the first line on the screen;
M	command; move cursor to the first printable character of the middle line on the screen;
L	command; move cursor to the first printable character of the last line on the screen;
m	command; mark a particular line with a label;
%	command; show matching left or right parenthesis or brace;
(command; move cursor to the beginning of the most previous sentence;
)	command; move cursor to the beginning of the next sentence;

{	command; move cursor to the beginning of the most previous paragraph;
}	command; move cursor to the beginning of the next paragraph;
[[command; move cursor to the beginning of the most previous section;
]]	command; move cursor to the beginning of the next section;
/	command; initiates a forward pattern search;
?	command; initiates a backward pattern search;
n	command; repeats the most previous pattern search;
N	command; repeats the most previous pattern search in the opposite direction;
^	metacharacter; used in pattern searches to match a pattern at the beginning of a line;
\$	metacharacter; used in pattern searches to match a pattern at the end of a line;
\	metacharacter; used in pattern searches to strip away the special meaning of a metacharacter;
.	metacharacter; used in pattern searches to match any single character;
\<	metacharacter; used in pattern searches to match a pattern at the beginning of a word;
\>	metacharacter; used in pattern searches to match a pattern at the end of a word;
[...]	metacharacters; used in pattern searches to match any one of the enclosed characters;
*	metacharacter; used in pattern searches to match zero or more instances of the preceding character;
w	command; move cursor forward to the beginning of the next word, or to the next punctuation mark, whichever comes first;
W	command; move cursor forward to the beginning of the next word, ignoring punctuation;
b	command; move cursor backwards to the beginning of the previous word, or to the most previous punctuation mark, whichever comes first;
B	command; move cursor backwards to the beginning of the previous word, ignoring punctuation;
e	command; move cursor forward to the end of the next word, or to the next punctuation mark, whichever comes first;
E	command; move cursor forward to the end of the next word, ignoring punctuation;
fc	command; move cursor forward to the next instance of the specified character, <i>c</i> ;
Fc	command; move cursor backwards to the next instance of the specified character, <i>c</i> ;
tc	command; move cursor forward to the first character to the left of the next instance of the specified character, <i>c</i> ;
Tc	command; move cursor backwards to the first character to the right of the next instance of the specified character, <i>c</i> ;
;	command; repeats the most previous f , F , t , or T command;
,	command; repeats the most previous f , F , t , or T command, in the opposite direction;
^, 0 (zero)	commands; move cursor to the first printable character on the

\$	current line;
	command; move cursor to the end of the current line;
	command; move cursor to specified column number in current line;
-- or --	commands; returns cursor to its most previous position.

This section describes several commands that enable you to move around in your file. You should try each of these commands as they are discussed to familiarize yourself with them.

Cursor-Positioning Keys

If your terminal has cursor-positioning keys, these keys can be used in *vi* to position the cursor in the file you are editing. The **h**, **j**, **k**, and **l** commands perform the same functions as the cursor-positioning keys. The **h** command moves the cursor one space to the left ([BACKSPACE] and ctrl-H also moves the cursor one space to the left). The **j** command moves the cursor down one line in the same column (as do ctrl-J and ctrl-N), the **k** command moves the cursor up one line in the same column (as does ctrl-P), and the **l** command moves the cursor one space to the right ([SPACE] also moves the cursor one space to the right). These commands are summarized below:

[↑] = **k** = ctrl-P
 [→] = **l** = [SPACE]
 [↓] = **j** = ctrl-J = ctrl-N
 [←] = **h** = [BACKSPACE] = ctrl-H

Scrolling and Paging

The **ctrl-D** command scrolls down in the file, leaving several lines of continuity between the previous screenful of text and the new screenful of text (note that [CTRL] must be held down while the next key is pressed). The **ctrl-U** command scrolls up in the file, also leaving several lines of continuity on the screen. If either **ctrl-D** or **ctrl-U** is preceded by a number argument, then the number of lines scrolled is equal to that specified number, and remains so until changed again.

If you want more control over the scrolling process, the **ctrl-E** command exposes one more line at the bottom of the screen, and the **ctrl-Y** command exposes one more line at the top. Preceding **ctrl-E** or **ctrl-Y** with a number causes the command to be executed that many times.

There are two paging commands, **ctrl-F** and **ctrl-B**, which move forward and backward one page in the file, respectively. Both commands leave a few lines of continuity between screenfuls of text. Giving a number argument to either of these paging commands executes the command that many times.

Note that paging moves you more abruptly than scrolling does, and leaves you fewer lines of continuity between screenfuls of text.

Moving From Line to Line

The **+** and **-** commands move the cursor to the first printable character on the next line or the previous line, respectively. [RETURN] or ctrl-M have the same effect as **+**. A preceding number argument executes these commands that many times.

The **G** command, when preceded by a line number, positions the cursor at the beginning of that line in the file. For example, **3G** positions the cursor at the beginning of the third line. If you do not specify a number, the cursor is positioned at the beginning of the last line of the file.

The **H** command positions the cursor at the beginning of the first line on the screen. If you precede **H** with a number, as in **4H**, the cursor is positioned at the beginning of the fourth line on the screen.

The **M** command positions the cursor at the beginning of the middle line on the screen. The **M** command ignores any line number argument.

The **L** command positions the cursor at the beginning of the last line on the screen. You can precede the **L** command by a number, as in **4L**, which positions the cursor at the beginning of the fourth line above the bottom of the screen.

Note that the **H**, **M**, and **L** commands reference the first, middle, and last lines of the current screenful of text. They do not reference the first, middle, and last lines of the entire file.

The **m** command enables you to mark specific lines with a label so that you can return to them. The label must be a single, lower-case letter in the range "a" through "z". To mark a line, first move the cursor to the particular line (using any of the commands described in *Moving Around in the File*), and type **m?**, where ? is the label you have selected. For example, **+++me** moves the cursor ahead three lines and marks that line with the label "e".

To reference a line you have marked, precede your label with a grave accent (`). For example, **`e** moves the cursor to the line you marked with the label "e". Note also that the cursor is placed in exactly the same spot within the line that it was when you marked the line. If you are not particularly interested in a specific position within a marked line, use an apostrophe (') instead of a grave accent. Thus, **'e** moves the cursor to the beginning of the line marked by the label "e", regardless of where the cursor was in the line when you marked it. Try marking a few lines, using both the apostrophe and grave accent, until you are familiar with their differences.

Marks are defined until you begin editing another file, or until you leave the editor. Marks cannot be erased.

The **%** command shows you the matching left or right parenthesis or brace for the parenthesis or brace currently marked by the cursor.

Skipping Over Sentences, Paragraphs, and Sections

The **(** and **)** (left and right parentheses) commands move the cursor to the beginning of the previous and next sentences, respectively. A sentence is defined to end at a period, an exclamation point, or a question mark, followed either by two spaces or the end of a line. Any number of closing parentheses, brackets, double quotes, or single quotes may follow the period, exclamation point, or question mark, as long as they occur before the two spaces or the end of the line. The **(** and **)** commands can be preceded by a number to move the cursor over several sentences at once.

The { and } (left and right braces) commands move the cursor to the beginning of the previous and next paragraphs, respectively. A paragraph is defined as a block of text beginning and ending with a blank line, or a block of text delimited by macro invocations. The default list of macros (from the *-ms* and *-mm* macros packages) includes **.IP**, **.LP**, **.PP**, **.QP**, **.P**, **.LI**, and **.bp**. These macros are used so that files containing *nroff/troff* text can be easily edited with *vi*. You may add your own macro names to those already recognized by appropriately setting the **paragraphs** option (see *Setting Vi Options* later in this article). The { and } commands can be preceded by a number to move the cursor over several paragraphs at once.

The [[and]] (double left and right brackets) commands move the cursor to the beginning of the previous and next sections, respectively. A section is defined as beginning and ending with a line containing a ctrl-L (formfeed character) in the first column, or as a block of text delimited by macro invocations. The default list of macros defining a section is **.NH**, **.SH**, **.H**, and **.HU**. You may add your own macro names to those already understood by appropriately setting the **sections** option (see *Setting Vi Options* later in this article). If [[or]] is preceded by a number argument, it is interpreted to be the new window size (number of lines per screenful of text).

Searching for a Pattern

You can tell *vi* to search for a particular pattern (string of characters) in your file. To do this, type a slash (/), followed by the pattern you want to search for, followed by [RETURN]. Note that the entire command is printed at the bottom of your screen. If *vi* finds the pattern, *vi* positions the cursor at the beginning of the pattern. If the pattern cannot be found, *vi* prints an error message and returns the cursor to its location prior to the search.

The slash initiates a forward search, with wraparound, starting from the current position of the cursor. Replacing the slash with a question mark (?) initiates a backward search, with wraparound, starting from the current position of the cursor. If a number argument is specified before / or ?, it is interpreted to be the new window size (number of lines per screenful of text).

If you want your pattern to match only at the beginning of a line, begin your pattern with a caret (^). If you want your pattern to match only at the end of a line, end your pattern with a dollar sign (\$).

Here are some examples:

```
/test[RETURN]
```

This is a forward search for the string "test". Note that this pattern matches "re-test", "testing", "detestable", or "test". To find only the word "test" standing alone (but not at the end of a sentence, or just before a comma), type

```
/ test [RETURN]
```

The spaces require that "test" not be part of another word.

```
?^Today[RETURN]
```

This is a backward search for the string "Today" appearing only at the beginning of a line.

```
/regret${RETURN}
```

This is a forward search for the string "regret" appearing only at the end of a line.

The **n** command enables you to repeat the most recently executed search. Each time **n** is typed, the previous search is re-executed. The **N** command also repeats the most recently executed search, but in the opposite direction. These commands are handy for finding a particular occurrence of a pattern without having to re-type the search each time.

There are times when you want to position the cursor at the beginning of the line containing the pattern. This can be done by typing your search command in a slightly different way. For example,

```
/key/+0[RETURN]
```

searches forward and positions the cursor at the beginning of the line containing the string "key". You can also position the cursor at the beginning of a line relative to the line containing the pattern. For example,

```
/FIFO/-3[RETURN]
```

searches forward and positions the cursor at the beginning of the third line before the line containing the string "FIFO". Also,

```
?CRT?+2[RETURN]
```

searches backward and positions the cursor at the beginning of the second line after the line containing the string "CRT".

There are two options, **magic** and **nomagic**, which affect the way you can specify patterns (see the section entitled *Setting Vi Options*). If the **nomagic** option is set, then only the characters **^** and **\$** have special meaning in patterns. If you want to include either of these characters in the actual pattern you search for, they must be preceded by a backslash (****). The backslash *quotes* the character immediately following it, and strips away any special meaning that character might have. For example,

```
/\^L[RETURN]
```

searches for the string "**^L**". The backslash was necessary to keep the caret from being interpreted to mean "match this pattern at the beginning of a line".

If the **magic** option is set, then you have several other special characters that you can use in patterns, including **^** and **\$**. The **.** (dot) matches any character, as in

```
/chap.[RETURN]
```

which matches any five-character string that begins with "chap". The character combinations \< and \> match the beginning of a word and the end of a word, respectively. For example,

```
?\
```

matches any word beginning with "how", including "how" itself. Also,

```
/ed\>[RETURN]
```

matches any word ending with "ed", including "ed" itself.

Brackets are also special, and match any one of the characters enclosed in them. For example,

```
/file[123][RETURN]
```

matches "file1", "file2", and "file3". If the characters inside the brackets are preceded by a ^, then the brackets match any single character *not* enclosed in them, as in

```
/chap[^1234][RETURN]
```

which matches any five-character string beginning with "chap", except "chap1", "chap2", "chap3", and "chap4". If you want to specify large spans of letters or numbers, as in a through z, or 0 through 9, they can be abbreviated inside the brackets, as in [a-z] or [0-9].

The asterisk (*) matches zero or more instances of the character immediately preceding it. For example,

```
/b*[RETURN]
```

matches zero or more b's. Note that this is a useless search, since zero b's can be found much quicker than one or more b's. To find one or more b's, you must type

```
/bb*[RETURN]
```

Also,

```
/[123][123]*[a-z][RETURN]
```

matches a one, two, or three, followed by any number of one's, two's, and three's, followed by a single lower-case letter. Experiment with the asterisk until you understand the implications of matching zero or more occurrences of a pattern.

If the **magic** option is set, then the characters ^, \$, ., \<, \>, [,], and * have special meaning and must be quoted with a backslash if you want them to be literally matched in a pattern (note that the characters \< and \> must each be preceded by a backslash, as in \\\< and \\>). If the **nomagic** option is set, then only ^ and \$ require a backslash to be literally matched. Note that, to match a backslash literally, it also must be preceded with a backslash.

The characters `^`, `$`, `.`, `\<`, `\>`, `[`, `]`, `*`, and `\` are commonly called *metacharacters* whenever their special meanings are utilized. This helps to distinguish between their normal, literal use, and their use as special characters.

Moving Within a Line

The `w` and `W` commands advance the cursor to the beginning of the next word in the sentence, wrapping around to the next line if necessary. The difference between the two commands is that the `w` command also stops at each punctuation mark it encounters; the `W` command does not stop at punctuation.

The `b` and `B` commands move the cursor backwards to the beginning of the previous word, wrapping around to the previous line if necessary. The `b` command stops at punctuation, while the `B` command does not.

The `e` and `E` commands advance the cursor to the end of the next word in the sentence, wrapping around to the next line if necessary. The `e` command stops at punctuation, while the `E` command does not.

Note that the `w`, `W`, `b`, `B`, `e`, and `E` commands all wrap around to lines other than the current line. These commands can be preceded by a number to move the cursor over several words at once.

The `f` and `F` commands move the cursor forward or backward, respectively, to the next occurrence of the specified character. The cursor is placed on the specified character. For example, `fc` moves the cursor forward to the first occurrence of the character "c", and `F:` moves the cursor backwards to the first occurrence of a colon. The `f` and `F` commands can be preceded by a number, as in `3fr`, which moves the cursor forward to the third occurrence of the character "r". Both `f` and `F` work only on the current line, and do not wrap around to other lines.

The `t` and `T` commands are identical to the `f` and `F` commands, except that the cursor is placed one character to the left or right of the specified character, respectively. For example, `2Tm` moves the cursor backwards to the second occurrence of the character "m", and places the cursor one character to the right. `3t.` moves the cursor forward to the third occurrence of a period, and places the cursor one character to the left.

The `;` command repeats the most previously executed `f`, `F`, `t`, or `T` command. Thus, `fi;;` is identical to `4fi`, and `Tj;` is identical to `2Tj`. The `,` command also repeats the most previously executed `f`, `F`, `t`, or `T` command, but in the opposite direction. Thus, if you execute `Tk`, a subsequent `,` searches forward in the current line for the letter k.

The `^` (caret) command moves the cursor to the first printable character on the current line. The `0` (zero) command is a synonym for `^`. Any number argument is ignored.

The `$` command moves the cursor to the end of the current line. If a number argument *n* is specified, `$` moves the cursor to the *n*th end of line it finds. Thus, `$` can wrap around to other lines, but only if preceded by a number argument (note that several explicitly typed `$`'s will not do this).

The | (vertical bar) command moves the cursor to the character in the column specified by a preceding number argument. If no number is given, | is a synonym for ^ and 0, in that it moves the cursor to the first printable character in the line.

Note that the f, F, t, T, ^, 0, and | commands work only on the current line. If you want to use these commands on a line other than the current line, you must first move the cursor to the line of interest.

Returning to Your Previous Position

The `` (two grave accents) command and the '' (two apostrophes) command both return you to your previous position. These commands can be used after you have executed a search command or one of the commands listed under *Moving Around in the File*, and you want to get back to where you were. Vi remembers only your last previous position.

Adding, Deleting, and Correcting Text

Material Covered:

i	command; insert text before cursor;
I	command; insert text at the beginning of a line (same as ^i);
a	command; append text after cursor;
A	command; append text at the end of a line (same as \$a);
o	command; create new line below line containing cursor;
O	command; create new line above line containing cursor;
x	command; delete character marked by cursor;
X	command; delete character immediately before character marked by cursor;
r	command replace character marked by cursor with another character;
s	command; replace one or more characters with one or more characters;
d	command; delete; can be combined with several other commands specifying what is to be deleted;
D	command; delete from current location through end of line (same as d\$);
c	command; change; can be combined with several other commands specifying what is to be changed;
C	command; change from current location through end of line (same as c\$);
.	command; re-execute last operation which changed text in buffer;
y	command; copy specified amount of text into a specified buffer;
Y, yy	commands; copy the specified number of complete lines into a specified buffer;
"	operator; introduces buffer name in which text is saved by previous y or Y commands;
a-z	buffers; the buffer names in which text can be saved with y or Y commands; there is, in addition, an unnamed buffer;
p	command; puts saved text back into the file, after or below the cursor;
P	command; puts saved text back into the file, before or above the cursor;
<<	command; shifts the specified number of lines one shift-width to the left;
>>	command; shifts the specified number of lines one shift-width to the right;
<	command; shifts the specified lines one shift-width to the left; can be combined with other commands;
>	command; shifts the specified lines one shift-width to the right; can be combined with other commands;
J	command; joins the specified number of lines together;

u command; reverses the last change made to the file;
U command; restores the current line back to its state before editing began;

Inserting and Appending Text

The **i** and **a** commands are used for inserting and appending text, respectively. The **i** command places text to the left of the cursor, and the **a** command places text to the right of the cursor. Both commands are cancelled by [ESC].

You may insert or append many lines of text, or just a few characters, with the **i** and **a** commands. To type in more than one line of text, press [RETURN] at the place in your text where you want the new line to appear. When you are inserting or appending text, [RETURN] causes *vi* to create a new line, and to copy the remainder of the current line onto the new line.

If a number *n* is specified before the **i** or **a** command, then the text you add is duplicated *n*-1 times when [ESC] is pressed. This works only if there is room on the current line for the duplications. For example, if you type **5a** at some particular point in a line, and your appended text is "hi", then, when [ESC] is pressed, the text actually appended will be expanded to "hihihihihi".

If you want to start adding text on a new line that does not currently exist, you can create a new line in your text with the **o** and **O** commands. The **o** command creates a new line after the line containing the cursor, and the **O** command creates a new line before the line containing the cursor. The **o** and **O** commands can create only one new line, but pressing [RETURN] while using the **o** and **O** commands causes *vi* to create an additional new line for you. The **o** and **O** commands are cancelled by [ESC], and ignore any preceding number argument. Thus, the only difference between the **i**, **a**, **o**, and **O** commands is that the **o** and **O** commands automatically create a new line on which text can be added, while the **i** and **a** commands do not. New lines can be created with all four commands simply by pressing [RETURN].

During an insert or append operation, if a **ctrl-@** is typed as the first character of the text to be inserted/appended, the **ctrl-@** is replaced by the most previous text that was inserted or appended. A maximum of 128 characters are saved from the previous text addition. If more than 128 characters were inserted or appended in the last text addition, the **ctrl-@** function is not available during the current text addition.

If you are in insert or append mode, the **autoindent** option is set, and you are at the beginning of a line, **ctrl-T** causes **shiftwidth** white space to be inserted at that point. White space inserted in this manner can be back-tabbed over with **ctrl-D** in insert or append mode. **Ctrl-D** is necessary because **shiftwidth** white space cannot be backspaced over.

The **ctrl-W** sequence enables you to back up over words (similar to **b** in command mode) while in insert or append mode. All words backed over are deleted from the text addition, even though the characters still appear on your screen.

The keys you use at the shell level to erase characters or entire lines can also be used in *vi*. When you are inserting or appending text, single characters can be erased with [BACKSPACE], and entire lines can be erased with **ctrl-U**. (Note that [BACKSPACE] and **ctrl-U** are the default keys assigned to erase single characters and entire lines. Your keys may have been re-defined. Check with your system administrator.) Note that you cannot erase characters which you did not insert or append, and that you cannot backspace into a previous line.

Experiment with the **i**, **a**, **o**, and **O** commands until you are familiar with what each command does. Be sure to note the effects of pressing [RETURN] with each of these commands.

Character Corrections

The **x** command deletes the character marked by the cursor. You can delete more than one character by preceding **x** with a number. **3x**, for example, deletes the next three characters, including the one marked by the cursor.

The **X** command deletes the character immediately before the one marked by the cursor. Preceding **X** with a number deletes that many characters before the current location of the cursor.

Both **x** and **X** work only on the current line; they cannot delete characters on any line other than the current line.

The **r** command replaces one character with another. For example, **rT** replaces the character marked by the cursor with the character "T". If a number *n* precedes the **r** command, then *n* characters are replaced by the single character you type next. For example, **4rt** replaces the next four characters with the letter t.

The **s** command replaces one or more characters with the specified string of characters. When not preceded by a number, the **s** command replaces a single character with the specified string. For example,

```
sTTY[ESC]
```

replaces the character marked by the cursor with the string "TTY". When preceded by a number, the **s** command replaces the specified number of characters, beginning with the character marked by the cursor, with the specified string of characters. For example,

```
4sinteresting[ESC]
```

replaces the next four characters with the string "interesting". Note that the **s** command prints a dollar sign at the end of the text to be replaced so you can see the extent of the change. The dollar sign is removed when you press [ESC].

The **d** command can be combined with several of the commands previously discussed to delete characters and words. For example, **dw** deletes the next word, and **db** deletes the previous word. **d[SPACE]** deletes the character marked by the cursor (this is equivalent to the **x** command). The **d** command can be preceded by a number to delete several words or characters, as in **3db**, which deletes the last three words. The **d** command can also be used with the **f**, **F**, **t**, and **T** commands. For example, **dtr** deletes everything from the current position of the cursor up to (but not including) the next "r" that appears in the current line. Experiment with these combinations until you are familiar with their effects.

The **c** command can also be combined with several other commands to change characters and words. The **c** command can be preceded by a number. Here are some examples:

c5yesterday[ESC]

This changes the next five words to the string "yesterday". Note that the "c" and the "5" could be interchanged with the same result.

4cbvariable name[ESC]

This changes the previous four words to the string "variable name".

c[SPACE]in a buffer[ESC]

This changes the character marked by the cursor to the string "in a buffer".

cfqHP-UX operating system[ESC]

This changes everything from the current position of the cursor up to (and including) the first occurrence of a "q" to the string "HP-UX operating system". The **c** command can be used similarly with the **F**, **t**, and **T** commands.

Note that the **c** command marks the end of the text to be changed with a dollar sign so you can see the extent of the change. The dollar sign is removed after you press [ESC].

Line Corrections

The **d** and **c** commands can also delete or change lines or groups of lines. The **d** command can be appended to itself to delete one complete line. For example, **dd** deletes the current line, and **5dd** deletes the current line and the next four lines.

The **d** command can be combined with several other commands. For example, **dL** deletes everything from the current position of the cursor through the last line on the screen. **d3L** deletes everything from the current position of the cursor through the third line from the bottom of the screen. The **d** command can also be used with a search, so that

d/market\${RETURN}

deletes everything from the current position of the cursor up to the beginning of the string "market", which must occur at the end of a line. Try the **d** command with the (,), {, }, [[, and]] commands to delete one or more sentences, paragraphs, or sections.

Note that any of the commands discussed under *Moving Around in the File* can be combined with the **d** command to delete specific portions of text. Also note that, if you delete five or more lines, *vi* informs you of the number of lines deleted with a message at the bottom of your screen.

The **D** command is shorthand for **d\$**, causing all characters from the cursor to the end of the line to be deleted. Any preceding number argument is ignored.

The **c** command can also be appended to itself (thus creating the **cc** command) to change one complete line. **S** is a synonym for **cc**. For example,

ccEnter the value for variable A.[ESC]

changes the current line to the sentence "Enter the value for variable A. " .

4SPlace illustration here.[ESC]

This changes the current line and the three lines following it to the sentence "Place illustration here. " . Note that the **S** command was used, and that the results are the same as if **cc** had been used.

cMPlace output on TTY4.[RETURN]Call exit routine.[ESC]

This changes everything from the current position of the cursor to the middle line on the screen to the two sentences "Place output on TTY4. " and "Call exit routine. " . Note that each sentence is on a separate line.

c([RETURN])Insert new paragraph here.[RETURN][ESC]

Here, the initial "(" moves the cursor to the beginning of the next paragraph, and then the entire previous paragraph is changed to a blank line, followed by the sentence "Insert new paragraph here. " , followed by another blank line.

+c/while/-1[RETURN]continue:[ESC]

The initial "+" advances the cursor to the first printable character on the next line. Then, everything from the beginning of that line up to and including the line before the next "while" statement is changed to the single statement "continue; " .

Like the **d** command, the **c** command can be combined with any of the commands discussed under *Moving Around in the File*, and *vi* informs you when five or more lines are being changed. Also, as in previous **c** examples, the end of the text to be changed is marked with a \$. Try some of the other combinations not covered above until you are familiar with how **c** works.

The **C** is equivalent to **c\$**, causing all the characters from the cursor to the end of the line to be changed to the text that follows. Any preceding number argument is ignored.

The . (dot) command repeats the last command which made a change in the text. Thus, **dw.....** is the same as **6dw**, in that both commands delete the next six words. The dot command can be used to re-execute *any* command which modified the buffer text, but is limited to that command which was executed most recently.

Copying and Moving Text

The **y** command copies a specified portion of text into a buffer. There are 26 named buffers, named "a" through "z", and one unnamed buffer. If you do not specify a buffer name, the copied text is automatically placed in the unnamed buffer. For example, **yw** copies the next word into the unnamed buffer, and **y2B** copies the previous two words into the unnamed buffer.

When specifying a buffer name, the name must be preceded by a double quote ("). This tells *vi* that the character to follow is a buffer name. For example, **)"ay2**(copies the previous two sentences into buffer "a" (the initial **)** ensures that complete sentences are copied). Also,

```
"ty/^two[RETURN]
```

copies everything from the current position of the cursor up to the line beginning with the string "two", and puts the text in buffer "t".

Note that the **y** command starts copying at the current position of the cursor. Thus, partial words or sentences may be copied if the cursor is in the middle of a word or sentence when you give the **y** command. Note also that, when copying forward in the file, the character marked by the cursor is included in the copied text. When copying backwards, however, the copied text begins with the character preceding the character marked by the cursor.

The **Y** command is used to copy complete lines of text, regardless of the position of the cursor within the line. For example, **3Y** copies three lines, including the current line, into the unnamed buffer. **"f6Y** copies six lines, including the current line, into buffer "f". Also,

```
"gY/inventory[RETURN]
```

copies every line from the current line up to and including the line containing the string "inventory", and saves them in buffer "g". A synonym for **Y** is **yy**.

The **p** and **P** commands put the copied text back into the file relative to the location marked by the cursor. The **p** command puts the text after or below the cursor, and the **P** command puts the text before or above the cursor. Exactly where the text is placed in relation to the cursor is determined by the amount of text being placed. If there is room on the current line for the text, then the text is placed after (**p**) or before (**P**) the cursor. If there is too much text to fit on one line, then *vi* creates one or more new lines below (**p**) or above (**P**) the cursor, and puts the text there. For example, **"rp** puts the text contained in buffer "r" into the file after or below the cursor. If no buffer name is specified, the text in the unnamed buffer is put back into the file.

Up to now, the copied text has been left in its original location and duplicated elsewhere in the file. If you do not want the text left in its original location, you can use the **d** command. For example, **5dd** deletes the next five lines, and saves them in the unnamed buffer (that's right - every deletion you perform is saved in the unnamed buffer until it is overwritten by the next deletion). **"wd2}** deletes the next two complete paragraphs (if the cursor is at the beginning of a paragraph) and saves them in buffer "w". The **p** or **P** command can then be used to put the deleted text elsewhere in the file.

You can copy text from one file into another. First, save the text in any of the named buffers. Once the text is saved, stop editing the current file and begin editing the file in which the text is to be inserted (the commands used to edit other files are described in the section entitled *Editing Other Files*). Now use the **p** or **P** command to put the saved text into the file. Do not use the unnamed buffer to transfer text from one file to another, because the contents of the unnamed buffer are lost when you change files.

Shifting Lines

The << and >> commands move the specified number of lines one shift-width to the left or right, respectively. One shift-width is equal to the number of columns specified by the **shiftwidth** option (see the section entitled *Setting Vi Options*). For example, **4>>** moves four lines one shift-width to the right. The << and >> commands are limited to numerical arguments only.

The < and > commands can be used with numbers and other commands to shift large groups of lines. For example, **>3L** moves every line from the current line to the third line from the bottom of the screen one shift-width to the right. Also,

```
</RAM[RETURN]
```

moves every line from the current line to the first line containing the string "RAM" one shift-width to the left. The < and > commands may be combined with any of the commands discussed under *Moving Around in the File*.

Continuous Text Input

When you are typing in large amounts of text, it is convenient to have your lines automatically broken and continued on the next line so that you do not have to press [RETURN]. The **wrapmargin** option enables you to do this (see the section entitled *Setting Vi Options*). For example, if the **wrapmargin** option is set equal to 10, *vi* breaks each line at least 10 columns from the right-hand edge of the screen.

If you want to join broken lines together, use the **J** command. For example, **3J** joins three lines together, beginning with the current line. *Vi* supplies white space at the place or places where the lines were joined, and moves the cursor to the first occurrence of the supplied white space.

Undoing a Command

The **u** command reverses the last change you made to your text. The **u** command is able to undo only the last change you have made. Note that a **u** command also undoes itself. If you have made several changes to a line, and you want to reverse *all* of the changes, use the **U** command. The **U** command restores the current line back to the state it was in when you began editing it.

Special Vi Commands

Material Covered:

:set	command; enables, disables, sets, or lists options;
autoindent	option; enables/disables automatic indentation;
autowrite	option; enables/disables automatic writing to the <i>vi</i> buffer after an editing session;
ignorecase	option; disables/enables upper- and lower-case distinction;
list	option; enables/disables tab and end-of-line markers;
magic	option; enables/disables extended set of metacharacters;
number	option; enables/disables line numbering;
shiftwidth	option; defines number of columns per shift-width;
showmatch	option; enables/disables parenthesis-, brace-, and bracket-matching;
slowopen	option; enables/disables screen refresh only when [ESC] is pressed;
wrapmargin	option; defines number of columns in right margin;
timeout	option; enables/disables one second time limit for macro entry;
readonly	option; enables/disables write protection for file;
paragraphs	option; defines the macro names recognized by the { and } commands;
sections	option; defines the macro names recognized by the [[and]] commands;
:map	command; defines macros;
:unmap	command; deletes macros;
ctrl-V	command; used to alter the meaning of special keys or characters;
:ab	command; defines abbreviations;
:una	command; deletes abbreviations;
:r	command; read contents of file or output of shell command into current file;
:w	command; write part or all of <i>vi</i> buffer to current file or to another file;
:e	command; edit same file over again, or begin editing another file;
:n	command; edit next file in <i>vi</i> argument list;
!	command; enables portions of the <i>vi</i> buffer to be filtered through an HP-UX command.

Setting Vi Options

Vi has several options that you can set for the duration of your editing session.

The **autoindent** option, when set, automatically indents each line of text so that it begins in the same column as the previous line. While inserting text, you cannot backspace over this indentation, but you can backtab over it with ctrl-D. This option is helpful when typing in program text. To enable this option, type

```
:set ai[RETURN]
```

Disable this option by typing

```
:set noai[RETURN]
```

The default is **noai**.

The **autowrite** option, when set, automatically writes the contents of the *vi* buffer to the current file you are editing when you quit editing the current file. This is helpful when you change files or leave the editor using commands that do not normally save the contents of the *vi* buffer. To enable this option, type

```
:set aw[RETURN]
```

Disable this option by typing

```
:set noaw[RETURN]
```

The default is **noaw**.

The **ignorecase** option, when set, causes *vi* to ignore case in searches. To enable this option, type

```
:set ic[RETURN]
```

Disable this option by typing

```
:set noic[RETURN]
```

The default is **noic**.

The **list** option, when set, causes a tab to be printed as "**^I**", and marks the end of each line with a dollar sign. To enable this option, type

```
:set list[RETURN]
```

Disable this option by typing

```
:set nolist[RETURN]
```

The default is **nolist**.

The **magic** option, when set, causes the period, left and right brackets, the asterisk, and the character combinations **\<** and **\>** to be treated in a special way when used in search patterns (see the section entitled *Searching for a Pattern*). To enable this option, type

```
:set magic[RETURN]
```

Disable this option by typing

```
:set nomagic[RETURN]
```

The default is **nomagic**.

The **number** option, when set, causes line numbers to be prefixed to each text line on your screen. To enable this option, type

```
:set nu[RETURN]
```

Disable this option by typing

```
:set nonu[RETURN]
```

The default is **nonu**.

The **shiftwidth** option enables you to specify the number of columns to skip when using <, <<, >, >>, ctrl-D; and ctrl-T (see the section entitled *Shifting Lines*). Ctrl-D backtabs over inserted shift-widths (using <, <<, >, or >>) or any indentation provided by the **autoindent** option. Ctrl-T inserts one shift-width at the beginning of the current line during text insertion. To set this option, type

```
:set sw = val[RETURN]
```

where *val* is the number of columns to skip. The default is **sw = 8**.

The **showmatch** option, when set, causes *vi* to show you the opening parenthesis, brace, or bracket when you type the corresponding closing parenthesis, brace, or bracket. This is helpful in complex mathematical expressions. To enable this option, type

```
:set sm[RETURN]
```

Disable this option by typing

```
:set nosm[RETURN]
```

The default is **nosm**.

The **slowopen** option, when set, causes *vi* to wait until you press [ESC] to update the screen after inserting or appending text. This is used on slow terminals to decrease the amount of time spent waiting for the screen to be updated. To enable this option, type

```
:set slow[RETURN]
```

Disable this option by typing

```
:set noslow[RETURN]
```

The default is **slow**.

The **wrapmargin** option enables you to specify the number of columns you want in your right margin. This is used when you are using continuous text input (see section entitled *Continuous Text Input*). To set this option, type

```
:set wm = val[RETURN]
```

where *val* is the number of columns in your right margin. The default is **wm = 0**.

The **timeout** option, when set, places a one second time limit on the amount of time it takes you to type in a macro name (see the section entitled *Defining Macros*). To enable this option, type

```
:set to[RETURN]
```

Disable this option by typing

```
:set noto[RETURN]
```

The default is **to**.

The **readonly** option, when set, places write protection on the file you are editing. This is used when you want simply to look at a file, and you want to ensure that you do not inadvertently change or destroy the contents of the file. To enable this option, type

```
:set readonly[RETURN]
```

Disable this option by typing

```
:set noreadonly[RETURN]
```

The default is **noreadonly**.

The **paragraphs** option contains the list of macro names recognized by the { and } commands as marking the beginning and end of a paragraph. Suppose you have three macros, .PG, .P, and .EP, that you want *vi* to recognize as paragraph delimiters. All you have to do is type

```
:set para = PGP EP[RETURN]
```

Note that, if a macro name is only one character long, you must type the single character macro name, followed by a space. The default **paragraph** string is

```
para = IPLPPPQPbpP LI
```

You may add your macros to this string, or completely redefine it using different macro names.

The **sections** option contains the list of macro names recognized by the [[and]] commands as marking the beginning and end of a section. **Sections** is defined in exactly the same way as **paragraphs** above. The default list of macro names is

```
sect = NHSHH HU
```

There are several other options available, but they are less commonly used than these. You can get a list of all possible options and their settings by typing


```
:set all[RETURN]
```

A list of all the options which you have changed is generated by typing

```
:set[RETURN]
```

If you want to know the value of a particular option, type

```
:set opt?[RETURN]
```

where *opt* is the name of the option. Note that multiple options can be set on one line, as in

```
:set ai aw nu[RETURN]
```

If a number argument is specified before the **:set** command, it is interpreted to be the new window size (number of lines per screenful of text).

Defining Macros

Vi has a macro facility which enables you to substitute a single keystroke for a longer sequence of keystrokes. If you are repeatedly typing the same sequence of commands, then you can probably save time and typing by defining a macro to perform the sequence of commands for you.

You use the **:map** command to define a macro. After the **:map** command, you type the key or keys that invoke the macro, and then the sequence of keystrokes that you want to put in the macro. For example,

```
:map d d4w[RETURN]
```

causes **d** to delete the next four words every time it is pressed. Also,

```
:map c /I ctrl-V[RETURN]dwiYou ctrl-V[ESC][RETURN]
```

causes **c** to find an occurrence of "I ", delete it, and replace it with "You ". The **ctrl-V** command tells *vi* to simply enter the next keystroke into the text of the macro and to ignore any special meaning that keystroke might have. The **ctrl-V** command is used above to flag [RETURN] and [ESC], both of which would have terminated the **:map** command before it was completed. Instead, [RETURN] and [ESC] are entered as keystrokes in the macro string. The final [RETURN] terminates the **:map** command.

If the macro name specified consists of a pound sign (#) followed by a number in the range 0 – 9, then a special function key on your terminal is mapped. For example,

```
:map #3 cILLUSTRATION GOES HEREctrl-V[ESC][RETURN]
```

maps special function key number 3 such that, when pressed, it changes the current line to the line "ILLUSTRATION GOES HERE ". Of course, this feature is valid only on terminals which have special function keys.

Vi normally allows only one second to enter a macro name, so you should use only one keystroke to invoke the macro. However, if the **notimeout** option is set, *vi* imposes no time limit. If this is the case, you can use up to 10 keystrokes to invoke a macro. The sequence of keystrokes that define the macro can contain up to 100 keystrokes.

The **u** (undo) command, when invoked after a macro has been executed, reverses the effects of the entire macro.

Previously defined macros can be deleted with the **:unmap** command. For example, to delete the **c** macro defined above, type

```
:unmap c
```

If a number argument is specified before the **:map** or **:unmap** command, it is interpreted to be the new window size (number of lines per screenful of text).

Defining Abbreviations

You can define an abbreviation with the **:ab** command. For example,

```
:ab CRT cathode ray tube[RETURN]
```

defines "CRT" as an abbreviation that is expanded to "cathode ray tube" everywhere you type "CRT" in the text. Also,

```
:ab cs Department of Computer Sciences[RETURN]
```

defines "cs" as an abbreviation that is expanded to "Department of Computer Sciences" everywhere you type "cs" in the text.

The abbreviation name must contain only letters, digits, or underscores. *Vi* only expands abbreviations when they are delimited by white space on both sides, or by white space on the left and punctuation on the right. Abbreviations are not expanded if they appear as part of another word.

Abbreviations can be deleted with the **:una** command. For example,

```
:una cs
```

deletes the abbreviation associated with "cs".

If a number argument is specified before the **:ab** or **:una** command, it is interpreted to be the new window size (number of lines per screenful of text).

Reading Data Into Your Current File

The `:r` command enables you to read the contents of a file or the standard output from a shell command into the file you are currently editing. For example,

```
:r test_data[RETURN]
```

reads the contents of the file `test_data` into the current file after the cursor. Also,

```
:7r std_dev[RETURN]
```

reads the contents of the file `std_dev` into the current file after line seven.

You can also read the output from a shell command into your file by typing

```
:r !cmd[RETURN]
```

where `cmd` is the name of the shell command. For example,

```
:r !ls[RETURN]
```

reads a list of the files in your working directory into the file you are editing, beginning at the current cursor position.

If a number argument is specified before the `:r` command, it is interpreted to be the new window size (number of lines per screenful of text).

Writing Edited Text Onto a File

The `:w` command is used to write the current contents of the `vi` buffer onto a file. The contents of the `vi` buffer remain unchanged. It is a good idea to write the contents of the `vi` buffer onto a file periodically, especially if you have been editing the file for a long time, and have made significant changes. That way, should a system crash or a power failure occur, some or all of your changes are saved.

If you specified a file name when you invoked `vi`, then you need not specify a file name if you want to write to the current file. `Vi` remembers the name of the file you are editing or creating, and writes to that file by default. For example, if you invoked `vi` as

```
$ vi test_data
```

then you need only type

```
:w[RETURN]
```

to write the contents of the `vi` buffer onto `test_data`. However, if you did not specify a file name when you invoked `vi`, then you must supply a file name with the `:w` command. For example, if you invoked `vi` as

\$ vi

then you must type

```
:w filename[RETURN]
```

where *filename* is the name of the file on which you want the contents of the *vi* buffer to be written.

You can write your changes to an existing file other than the one you are editing. For example,

```
:w! format[RETURN]
```

writes your changes to the file **format**. Note that the exclamation point tells *vi* to overwrite the previous contents of **format** with the contents of the *vi* buffer.

You can also write your changes to a file that does not yet exist. For example,

```
:w thesis[RETURN]
```

causes *vi* to create a file called **thesis**, and writes all your changes on **thesis**.

You can specify that a portion of your text be written to another file that does not yet exist. For example,

```
:2,35w prog[RETURN]
```

creates a file called **prog** and writes line 2 through line 35 of the current file on **prog**. The same thing can be done with a file that already exists, as in

```
:3,10w! list[RETURN]
```

writes line 3 through line 10 of the current file on the file called **list**. The exclamation point causes the previous contents of **list** to be destroyed and replaced by the specified portion of the *vi* buffer.

If a number argument is specified before the **:w** command, it is interpreted to be the new window size (number of lines per screenful of text).

Note that, while you may append other files to the file you are currently editing, *vi* provides no facilities that enable you to append the current file to another file.

Editing Other Files

The **:e** command enables you to edit other files without leaving *vi*. For example,

```
:e report[RETURN]
```

tells *vi* to stop editing the current file and to start editing **report**. If **report** does not exist, *vi* creates it for you. Note that *vi* requires a **:w** command to precede a **:e** command, so that the previous contents of the *vi* buffer are saved (unless the **autowrite** option is set, in which case *vi* is silent).

You can also tell *vi* to start editing a file beginning with a particular line. For example,

```
:e + test[RETURN]
```

tells *vi* to start editing **test**, beginning with the last line of the file. Also,

```
:e + M letter[RETURN]
```

tells *vi* to start editing *letter* at the middle line of the screen. Any *vi* command discussed in the section entitled *Moving Around in the File* and not containing any spaces can be inserted after the " + " in the previous examples. For example,

```
:e +/CAE/+ 0ctrl-V[RETURN] cov_let[RETURN]
```

tells *vi* to start editing **cov_let**, with the cursor positioned at the beginning of the first line containing the string "CAE". Note that ctrl-V had to be used to flag [RETURN] so that the **:e** command is not terminated before it is completed.

If you decide that you do not like the changes you have made to a file, you can discard the changes and begin editing the same file over again by typing

```
:e![RETURN]
```

The exclamation point tells *vi* that you know what you are doing, and that you do not want to save the current contents of the *vi* buffer. To discard the changes and begin editing a different file, type

```
:e! name[RETURN]
```

where *name* is the name of the file you want to edit. Again, the exclamation point tells *vi* that a **:w** command is not necessary.

If a number argument is specified before the **:e** command, it is interpreted to be the new window size (number of lines per screenful of text).

Editing the Next File in the Argument List

The **:n** command tells *vi* to stop editing the current file and begin editing the next file in the argument list. For example,

```
:n[RETURN]
```

tells *vi* to start editing the next file in the argument list. *Vi* insists that you use a **:w** command before you begin editing the next file, unless you type

```
:n![RETURN]
```

which tells *vi* to discard any changes you have made to the current file, and begin editing the next file.

If a number argument is specified before the `:n` command, it is interpreted to be the new window size (number of lines per screenful of text).

Filtering Buffer Text Through HP-UX Commands

Portions of the *vi* buffer text can be given as input to an HP-UX command, the output of which is then re-inserted into the previous location of that text. The `!` command is used to invoke filtering.

For example, suppose you have a list of items, one per line, that you want to sort alphabetically. This is easily done in several ways. If a single `!` is used, then you must supply modifiers which specify the extent of the text to be sorted. Let's assume that your file looks like this:

```
.PP
crackers
peas
roast
apples
oranges
tomatoes
grapes
.PP
```

`.PP` is an *nroff/troff* paragraph macro, which is recognized by `{` and `}` as beginning and ending a paragraph. Thus, if your cursor is positioned at the beginning of the first `.PP` macro, and you type

```
!}sort[RETURN]
```

then the list of grocery items is replaced by the output from the *sort* command. The `}` command is used to select the next paragraph as input for *sort*.

A second way to sort the same text is by typing

```
7!!sort[RETURN]
```

If two `!`'s are typed, then whole lines are assumed, and the number argument specifies how many whole lines to sort. For this example to work, your cursor must be somewhere on the "crackers" line.

Note that, in both of the above examples, a single `!` and the command name is all that is printed at the bottom of your screen. No number arguments or modifiers are echoed.

Any HP-UX command with useful output can be used in place of *sort*, depending on what you want to do. Since *vi* has no right margin justification function, another useful command might be *nroff*, which could be used to justify right margins or perform other formatting.

Note that filtering affects only the buffer contents, not the actual contents of your current file.

Vi and Ex

Material Covered:

Q command; escape from *vi* into *ex*;
vi command; escape from *ex* back to *vi*.

Vi is actually one mode of editing within the editor *ex*. In fact, all of the commands beginning with **:** are also available in *ex*. You can escape to the *ex* line-oriented editor by giving the command **Q**. When the **Q** command is given, *vi* responds with a line of information, and then *ex* takes over and prints the *ex* prompt (**:**). To get from *ex* to *vi*, type **vi** after the *ex* prompt. *Vi* clears the screen and prints a screenful of text, with your current line at the time you typed **vi** at the top of the screen.

There are several things which can be done more easily in *ex*, the most notable of which are global searches and substitutions. Thus, you may find yourself, after a while, switching between the two editing modes to access functions which are better handled by one or the other. For information concerning the *ex* editor, refer to the *Ex Reference Manual* included in *HP-UX Selected Articles*.

The Shell Interface

Material Covered:

vi command; invokes the *vi* editor;
view command; invokes the *vi* editor in read-only mode;
! command; escape to the shell for the duration of one command;
:sh command; escape to the shell indefinitely;
ZZ command; writes the contents of *vi* buffer to current file and leaves editor;
:q! command; discards contents of *vi* buffer and leaves editor.

Getting Into Vi

There are two ways to invoke *vi* from the shell, one of which is to type

```
$ vi
```

optionally followed by the names of the files you want to edit. You can also invoke *vi* by typing

```
$ view
```

optionally followed by the names of the files you want to edit. *View* is the same editor as *vi*, except that the **readonly** option is automatically set. This protects the contents of a file from being accidentally overwritten or destroyed. *View* is used whenever you want to look at an important file, but you do not want to change its contents.

Note that the **readonly** option can be disabled or overridden while you are in *view*. Nothing prevents you from typing

```
:set noreadonly[RETURN]
```

which simply changes *view* into *vi*. Also, you can still overwrite the contents of a file when the **readonly** option is set by using a **:w!** command.

Getting Back to the Shell

You can get back to the shell temporarily in either of two ways. You can execute a shell command while editing a file by typing

```
!:cmd[RETURN]
```

where *cmd* is the name of the shell command you want to execute. For example,

```
!:ls[RETURN]
```

prints a list of all the files in your working directory. Once the command has been executed, you can either enter another command with **!:**, or you can continue editing where you left off by pressing [RETURN]. If you press [RETURN], *vi* responds by clearing the screen and displaying the text you were working on before the shell command was executed.

You can escape to a shell temporarily by typing

```
:sh[RETURN]
```

This puts you in a shell, where you can execute as many commands as you want. When you want to continue editing, press ctrl-D. *Vi* clears the screen and displays the text you were working on.

If a number argument is specified before the **!:** or **:sh** command, it is interpreted to be the new window size (number of lines per screenful of text).

There are two ways to return to the shell permanently. If you want to save all your changes to the current file and return to the shell, use the **ZZ** command. **ZZ** writes the contents of the *vi* buffer onto the current file (if any changes have been made), and leaves the editor.

If you do not want to save the changes you have made to the current file, then use **:q!**. **:q!** simply leaves the editor and discards the contents of the *vi* buffer. The file you were editing is left unchanged. You should be very sure that this is what you want to do, since the contents of the *vi* buffer are permanently lost.

If a number argument is specified before the **:q!** command, it is interpreted to be the new window size (number of lines per screenful of text).

Miscellaneous Topics

Material Covered:

.profile	file; automatically executed by the shell at login; can contain macros, abbreviations, and option settings; must reside in your home directory;
EXINIT	variable; placed in .profile file; contains macro, abbreviation, and option information;
.exrc	file; contains <i>ex</i> and <i>vi</i> initialization constructs; this file is automatically scanned by <i>ex</i> if EXINIT is not defined;
buffers 1–9	buffers; contain the last nine text deletions performed during the current edit session;
ctrl-V	operator; enable control characters to be inserted in file;
z	command; adjust and redefine window size;
ctrl-L	command; refreshes the screen;
ctrl-G, :f	command; provide information about your current edit session.

Vi Initialization

Option settings, macros, and abbreviations last only the length of your editing session, after which they either return to default settings or become undefined. If you do not want to bother with resetting these things each time you invoke *vi*, you can put your option settings, macros, and abbreviations in a file called **.profile**. This file is automatically executed by the shell when you log in. The **.profile** file must reside in your home directory.

If you include *vi* information in your **.profile**, they must be placed in a string and set equal to the variable EXINIT. EXINIT is a variable that is assumed by the system to contain information pertinent to the *vi* editor. For example, to set the **autoindent**, **autowrite**, and **number** options and define two macros, put the following two statements in your **.profile** file in your home directory:

```
EXINIT='set ai aw nulmap @ dd\map # x'  
export EXINIT
```

This EXINIT string sets the **autoindent**, **autowrite**, and **number** options and defines the two macros **@** and **#**, which delete one line and one character, respectively. Note that each **set** and **map** is separated from the next by a vertical bar (**|**), and that the entire string is enclosed in single quotes and set equal to EXINIT. The *export* command makes the information in EXINIT available to all processes you create.

If EXINIT is not defined when *vi* is invoked, then *vi* looks for the file **.exrc** in your home directory. If it is found, *vi* scans its contents, assuming that the information contained therein consists of various commands for setting up mapping, abbreviations, options, etc.

If the amount of initialization for *vi* is extensive, it is usually more convenient to forget about EXINIT, and use the **.exrc** file instead, since, to use EXINIT, the information must be specified in a string enclosed in single quotes. This could prove to be a very long string if there is a lot of initialization to do. Strings this size are normally hard to read and hard to input.

Appendix B at the end of this article contains a listing of the default `.exrc` file shipped with your system. You are free to use this file as your own `.exrc` file if you wish. To do so, simply copy the file `/etc/d.exrc` into your home directory, and rename it `.exrc`. Your system administrator may have already done this for you. To find out, list the files in your home directory using `ls -a`.

Recovering Lost Lines

Vi has nine buffers, numbered 1 through 9, in which the last nine text deletions are automatically stored. Thus, you can specify one of these buffers with the `p` or `P` command to recover a deletion. For example, "`3p`" puts the deleted text stored in buffer 3 into the *vi* buffer after or below the cursor.

The `.` command, which repeats the last command that made a change in your text, automatically increments the buffer number if the last command referenced a numbered buffer. Thus, "`1p.....`" prints out all the text deleted in the last nine deletions. If you want to put a particular block of deleted text back into your file, but you do not know which buffer to look in, you can perform a sequence of commands like "`1pu.u.u.`" (and so on), which prints the contents of each buffer until you find the text you want. The `u` command gets rid of the unwanted text you encounter as you search.

Note that text stored in buffers 1 through 9 is preserved between files (as long as you do not exit the editor itself), so you may insert deleted text from one file into another by using buffers 1–9 and the `p` or `P` command.

Entering Control Characters in Your Text

If you need to put a control character in your text, you must precede the control character with a `ctrl-V`. The `ctrl-V` causes a caret (^) to be printed on your screen, showing that the next character is to be interpreted as a control character. For example, to enter a `ctrl-L` in your text, type

```
ctrl-V ctrl-L
```

This causes two characters, "`^L`", to be printed on your screen. If you try to backspace over them, however, you can see that they are actually one character.

You may enter any control character into your file except one: the null character (`ctrl-@`). There is also a restriction that applies to the line-feed character (`ctrl-J`). A linefeed is not allowed to occur anywhere except the beginning of a line, because *vi* uses the linefeed to separate lines in your file.

Adjusting the Screen

If a transmission error or the output from a program causes your screen to become cluttered, you can refresh the screen by pressing `ctrl-L`. *Vi* clears the screen and reprints the text you were working on.

The `z` command is used to position specific lines on the screen. `z[RETURN]` places the current line at the top of the window, `z.` places the current line in the middle of the window, and `z-` places the current line at the bottom of the window. If a number argument *n* is specified after `z` but before the modifier, then the window size is changed to be *n* lines long after `z` has executed. If *n* is specified before `z`, then `z` places line number *n* (instead of the current line) at the top, middle, or bottom of the new screen. For example, `z10-` places the current line at the bottom of a 10-line

window. Also, **6z.** places line number 6 at the middle of the screen, leaving the window size unchanged.

Printing Your File Status

If you are editing a file and lose track of where you are in the file, or if you forget the name of the file you are editing, the **ctrl-G** command can help you. In response to the **ctrl-G** command, *vi* prints the name of the file you are editing, the number of the current line, the number of lines in the buffer, and how much of the buffer you have already edited (expressed as a percentage). The **:f** command is a synonym for **ctrl-G**.

Appendix A: Character Functions

This appendix gives the *vi* meanings associated with each character in the ASCII character set. The characters are presented in the following order: control characters, special characters, digits, upper-case characters, and lower-case characters. For each character, its meaning is given as a command and during an insert, as appropriate. (Note that the control key (CTRL) is represented by \wedge in the following list).

- \wedge @ Not a command character. If it is typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted, the mechanism is not available. A \wedge @ cannot be part of the file text due to the editor's implementation.
- \wedge A Unused.
- \wedge B Moves backward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.
- \wedge C Unused.
- \wedge D As a command, scrolls forward one half of a page. A preceding integer specifies the number of logical lines to scroll for each command. This integer is remembered for all future \wedge D and \wedge U commands. During an insert, \wedge D backtabs over **autoindent** white space inserted at the beginning of a line. This white space cannot be backspaced over.
- \wedge E Exposes one more line at the bottom of the screen, leaving the cursor at its present position, if possible.
- \wedge F Moves forward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.
- \wedge G Prints the name of the current file, whether it has been modified, the current line number, the number of lines in the file, and how much of the buffer you have already edited (expressed as a percentage).
- \wedge H (BS) Same as **left arrow** (see **h**). During an insert, eliminates the last input character, backing over it but not erasing it. The character remains so you can see what you typed if you wish to type something only slightly different.
- \wedge I (TAB) Not a command character. When inserted, it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the **tabstop** option.
- \wedge J (LF) Same as **down arrow** (see **j**).
- \wedge K Unused.
- \wedge L (FF) Causes the screen to be cleared and redrawn.
- \wedge M (CR) Advances to the next line, at the first printable character on the line. If preceded by an integer, *vi* advances that many lines. During an insert, \wedge M causes the insert to continue onto another line.
- \wedge N Same as **down arrow** (see **j**).
- \wedge O Unused.
- \wedge P Same as **up arrow** (see **k**).
- \wedge Q Not a command character. In input mode, \wedge Q quote the next character, the same as \wedge V, except that some teletype drivers do not allow \wedge Q to be seen by *vi*. Use \wedge V instead.
- \wedge R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, \wedge R retypes the current line.
- \wedge S Unused.
- \wedge T Not a command character. During an insert, with **autoindent** set and at the beginning of the line, inserts **shiftwidth** white space.

<code>^U</code>	Scrolls up one page. A preceding integer specifies the number of lines to scroll. This integer is remembered for all future <code>^D</code> and <code>^U</code> commands. On a dumb terminal, <code>^U</code> will clear the screen and redraw it further back in the file.
<code>^V</code>	Not a command character. In input mode, <code>^V</code> quotes the next character so that it is possible to insert non-printing and special characters into the file, and include special characters in macros, abbreviations, etc.
<code>^W</code>	Not a command character. During an insert, <code>^W</code> mimics a <code>b</code> command, thus deleting all inserted characters from the current cursor location to the beginning of the previous word. The deleted characters remain on display. (See <code>^H</code>).
<code>^X</code>	Unused.
<code>^Y</code>	Exposes one more line at the top of the screen, leaving the cursor in its present position, if possible.
<code>^Z</code>	Unused.
<code>^[(ESC)</code>	Cancels a partially formed command, such as a <code>z</code> command when no following character has yet been given. It also terminates inputs on the last line (read by commands such as <code>:</code> , <code>/</code> , and <code>?</code>), and ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus press ESC if you don't know what is happening until the editor rings the bell.
<code>^\ ^]</code>	Unused.
<code>^]</code>	Searches for the word which immediately follows the cursor. It is equivalent to typing the <code>ex</code> command <code>:ta</code> , followed by that word, followed by RETURN.
<code>^^</code>	(Control- <code>^</code>), equivalent to the <code>ex</code> command <code>:e #</code> , which returns you to the previous position in the last edited file, or edits a file you specified if you got a "No write since last change" diagnostic, and you don't want to type the file name again. (In the latter case, you will have to do a <code>:w</code> before <code>^^</code> will work. If you don't want to write the file, then do a <code>:e! #</code> instead).
<code>^_ SPACE</code>	Unused.
<code>!</code>	Same as right arrow (see I).
<code>!</code>	An operator which processes lines from the buffer with reformatting commands. Follow <code>!</code> with the object to be processed, and then the command name terminated by RETURN. Doubling <code>!</code> and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the <code>!</code> . Thus, <code>2!)sort</code> , followed by RETURN, sorts the next two paragraphs by running them through the <code>sort</code> command. To read a file or the output of a command into the buffer use <code>:r</code> . To simply execute a command use <code>!:</code> .
<code>"</code>	Precedes a named buffer specification. There are named buffers <code>1 – 9</code> used for saving deleted text, and named buffers <code>a – z</code> into which text can be placed.
<code>#</code>	The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a <code>\</code> to insert it, since it normally backs over the last input character you gave.
<code>\$</code>	Moves to the end of the current line. If you execute <code>:set list</code> , then the end of each line will be shown by printing a <code>\$</code> after the end of the displayed text in the line. Given a count, <code>\$</code> advances to the end of the line that many lines from the current line (i.e. <code>3\$</code> advances to the end of the line two lines after the current line).
<code>%</code>	Moves to the parenthesis or brace which balances the parenthesis or brace at the current cursor position.
<code>&</code>	A synonym for the <code>ex</code> command, <code>:&</code> .
<code>'</code>	When followed by another <code>'</code> , returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter <code>a – z</code> , returns to the line which was marked with this letter

- with the **m** command, at the first non-space character in the line. When used with an operator, such as **d**, the operation takes place over complete lines.
- (Moves to the beginning of a sentence, or to the beginning of a LISP s-expression if the **lisp** option is set. A sentence ends at a ., !, or ? which is followed by either the end of a line or by two spaces. Any number of closing),], ", and ' characters may appear after the ., !, or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries. A count advances that many sentences.
 -) Advances to the beginning of a sentence. A count repeats the effect. See the description of (above for a description of a sentence.
 - * Unused.
 - + Same as carriage-return when used as a command.
 - , Reverses the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. A count repeats the search.
 - Moves to the previous line at the first non-white-space character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required, the screen is also cleared and redrawn, with the current line at the center.
 - . Repeats the last command which changed the *vi* buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit . to delete more words/lines. Given a count, it passes it on to the command being repeated.
 - / Used to initiate a forward search for a pattern. If you press / accidentally, you can use BACKSPACE to return to your previous position.
 - 0 Moves to the first character on the current line. Also used to form numbers after an initial 1 – 9.
 - 1 – 9 Used to form numeric arguments to commands.
 - : A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with RETURN, and the command is then executed. You can return to your previous position by pressing DEL or RUB if you press : accidentally.
 - ; Repeats the last single character search using **f**, **F**, **t**, or **T**. A count iterates the basic scan.
 - < An operator which shifts lines left one **shiftwidth**, normally 8 spaces. Like all operators, < affects lines when repeated, as in <<. Counts cause < to act on more than one line.
 - = Re-indents a line for LISP, as though the line was typed in with the **lisp** and **autoindent** options set.
 - > An operator which shifts lines right one **shiftwidth**, normally 8 spaces. Affects lines when repeated, as in >>. A count causes > to act on more than one line.
 - ? Initiates a backwards search for a pattern. If you press / accidentally, you can use BACKSPACE to return to your previous position.
 - @ A macro character. If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.
 - A Appends at the end of a line, a synonym for **\$a**.
 - B Backs up one word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the command.
 - C Changes the rest of the text on the current line; a synonym for **c\$**.
 - D Deletes the rest of the text on the current line; a synonym for **d\$**.
 - E Moves forward to the end of a word. A count repeats the command.
 - F Finds a single following character, backwards in the current line. A count repeats the search.
 - G Moves to the line number given as a previous argument, or the end of the file if no

	preceding argument is given. The screen is redrawn with the new current line in the center if necessary.
H	Homes the cursor to the top line of the screen. If a count is given, the cursor is moved to the count-th line on the screen. In all cases, the cursor is moved to the first non-white-space character on the line.
I	Inserts at the beginning of a line; a synonym for <code>^i</code> .
J	Joins lines together, supplying appropriate white space: one space between words, two spaces after a <code>.</code> , and no spaces at all if the first character of the line to be appended is <code>)</code> . A count causes that many lines to be joined rather than the default two.
K	Unused.
L	Moves the cursor to the first non-white-space character of the last line on the screen. If a count is given, the cursor is moved to the first non-white-space character of the count-th line from the bottom.
M	Moves the cursor to the middle line on the screen, at the first non-space character.
N	Scans for the next match of the last pattern given to <code>/</code> or <code>?</code> , but in the opposite direction.
O	Opens a new line above the current line and inputs text there, up to an ESC.
P	Puts the last deleted text back before or above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text is inserted before the current location of the cursor. May be preceded by a buffer name to retrieve the contents of that buffer.
Q	Quits from <i>vi</i> and goes to <i>ex</i> mode. In this mode, whole lines form commands, ending with RETURN. All <code>:</code> commands can be given; the editor supplies the <code>:</code> prompt.
R	Replaces characters on the screen with characters you type (overlay fashion). Terminates with ESC.
S	Changes whole lines; a synonym for <code>cc</code> . A count substitutes for that many lines. The lines are saved in numeric buffers, and erased on the screen before the substitution begins.
T	Takes a single following letter, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the command that many times.
U	Restores the current line to its state before you started changing it.
V	Unused.
W	Moves forward to the beginning of a word in the current line, where words are defined as sequences of non-space characters. A count repeats the command.
X	Deletes the character before the cursor. A count repeats the command, but only characters on the current line are deleted.
Y	Yanks a copy of the current line into the unnamed buffer, to be put back by a later <code>p</code> or <code>P</code> . A count yanks that many lines. Can be preceded by a buffer name to put text into that buffer.
ZZ	Exits the editor (same as <code>:x</code>). If any changes have been made, the buffer is written out to the current file, and the editor terminates.
[[Backs up to the previous section boundary, which is marked by a particular macro invocation (the names of which are specified in the sections option), or by <code>^L</code> (formfeed). Lines beginning with <code>{</code> also stop <code>[[</code> , making it useful for looking backwards through C programs. If the lisp option is set, <code>[[</code> also stops at each <code>(</code> it finds at the beginning of a line.
\	Unused.
]]	Moves forward to a section boundary (see description of <code>[[</code>).
^	Moves to the first non-space character on the current line.
_	(Underscore) Unused.
`	When followed by a <code>`</code> , returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a lower-

- case letter, returns to the position which was marked with this letter with an **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use **'**, the operation takes place over complete lines.
- a Appends arbitrary text after the current cursor position. The insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with ESC.
 - b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the command.
 - c An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is being changed is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a \$. A count causes that many objects to be changed.
 - d An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected.
 - e Advances to the end of the next word. A count repeats the command.
 - f Finds the first instance of the next character following the cursor on the current line. A count repeats the command.
 - g Unused.
 - h Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the left arrow key, or one of the synonyms (^H) has the same effect. A count repeats the command.
 - i Inserts text before the cursor. Otherwise, **i** is like **a**.
 - j Down arrow. Moves the cursor down one line in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include ^J (linefeed) and ^N.
 - k Up arrow. Moves the cursor up one line in the same column. Synonym is ^P.
 - l Right arrow. Moves the cursor one character to the right. SPACE is a synonym.
 - m Marks the current position of the cursor in the mark register which is specified by the next character (a – z). Return to this position or use with an operator by preceding the mark letter with ` or `.
 - n Repeats the last search specified with / or ?.
 - o Opens a new line below the current line. Otherwise, **o** is like **O**.
 - p Puts text after/below the cursor. Otherwise, **p** is like **P**.
 - q Unused.
 - r Replaces the single character marked by the cursor with a single character you type. The new character may be a RETURN (this is the easiest way to split lines). A count *n* replaces the next *n* characters with the character you type.
 - s Changes the single character marked by the cursor to the text which follows, up to an ESC. Given a count, that many characters are replaced by the text. The last character to be changed is marked with a \$.
 - t Advances the cursor up to the character before the next character typed on the current line. A count repeats the command.
 - u Undoes the last change made to the current buffer. If repeated, will alternate between these two states. It is thus its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric buffers.
 - v Unused.
 - w Advances to the beginning of the next word. A count repeats the command.
 - x Deletes the single character marked by the cursor. A count causes that many

- characters to be deleted. Works only on the current line.
- y An operator which yanks the following object into the unnamed temporary buffer. If preceded by a buffer name, the text is placed in that buffer also. Text can be recovered with a later **p** or **P**.
- z Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . specifies the center of the screen, and – specifies the bottom of the screen. A count may be given after **z** and before the following character to specify the new window size for the redraw. A count before **z** gives the number of the line to place in the center of the screen instead of the current line.
- { Moves to the beginning of the preceding paragraph. A paragraph begins at a macro invocation defined in the **paragraphs** option, and at the beginning of a section. A paragraph also starts at a blank line.
- | Places the cursor on the character in the column specified by the count.
- } Advances to the beginning of the next paragraph. See { for the definition of a paragraph.
- ~ Unused.
- ^? (DEL) Interrupts the editor, returning it to command mode.

Appendix B: Example .exrc File

The following is a reproduction of the default *.exrc* file shipped with your system. It is useful as an example of how it can be used to set up certain *vi* and *ex* parameters prior to your editing session. These contents can be changed at any time should the need arise to customize the editors for a particular application. Also, note that the line numbers in the following listing do not appear in the file, but are included to clarify the explanatory material that follows.

```
1 set autoindent autowrite showmatch wrapmargin=0 report=1
2 map ^W :set wrapmargin=8^M
3 map ^Z ^!}sort -b^M
4 map ^X {!}sort -b^M
5 map ^[h 1G
6 map ^[H 1G
7 map ^[F G
8 map ^[V ^B
9 map ^[U ^F
10 map ^[T ^Yk
11 map ^[S ^Ej
12 map ^[Q i
13 map ^[P x
14 map ^[L O
15 map ^[M dd
16 map ^[K D
17 map ^[J DjdG$
18 map! ^[A ^V
19 map! ^[D ^H
20 map! ^[C ^V
21 map! ^[B ^M
22 map! ^[L ^M
23 map! ^[Q ^[
24 map! ^[R ^[
```

In the above, the `^` character indicates that the CTRL (control) key is held down while the next following key is pressed. The `^[` sequence is the escape sequence, and is equivalent to the ESC key (if any) on your terminal. Here is a line-by-line description of the contents of the default *.exrc* file:

LINE ACTION

- 1 enables the **autoindent**, **autowrite**, and **showmatch** options, sets the **wrapmargin** option to 0, and sets the **report** option to one line.
- 2 maps the control-W sequence to the *ex* command:

```
:set wrapmargin=8
```

- The control-M at the end of the sequence is a carriage-return. This is entered into the *.exrc* file by pressing control-V followed by a carriage-return.
- 3 maps the control-Z sequence to a shell escape sequence. This sequence pipes the data from the beginning of the current line to the end of the current paragraph into the *sort(1)* command.

- 4 maps the control-X sequence to a shell escape sequence. This sequence pipes the data from the beginning of the current paragraph to the end of the current paragraph into the `sort(1)` command.
- 5 maps escape-h, a sequence often transmitted by the HOME key, to the editor command **1G** (go to line one of the file). This enables you to use the HOME key while editing in *vi*.
- 6 performs the same function as line 5.
- 7 maps escape-F, the sequence transmitted by the HOME DOWN key, to the editor command **G** (go to the last line of the file). This enables you to use the HOME DOWN key while editing in *vi*.
- 8 maps escape-V, the sequence transmitted by the PREV PAGE key, to the editor command **^B** (go back one page). This enables you to use the PREV PAGE key while editing.
- 9 maps escape-U, the sequence transmitted by the NEXT PAGE key, to the editor command **^F** (go forward one page).
- 10 maps escape-T, the sequence transmitted by the ROLL DOWN key, to the editor commands **^Yk** (scroll up one line; move cursor down one line).
- 11 maps escape-S, the sequence transmitted by the ROLL UP key, to the editor commands **^Ej** (scroll up one line; move cursor down one line).
- 12 maps escape-Q, the sequence transmitted by the INSERT CHAR key, to the editor command **i** (start insert mode).
- 13 maps escape-P, the sequence transmitted by the DELETE CHAR key, to the editor command **x** (delete current character).
- 14 maps escape-L, the sequence transmitted by the INSERT LINE key, to the editor command **O** (create a new line above the current line, and start insert mode).
- 15 maps escape-M, the sequence transmitted by the DELETE LINE key, to the editor command **dd** (delete current line).
- 16 maps escape-K, the sequence transmitted by the CLR LINE key, to the editor command **D** (delete to the end of the current line).
- 17 maps escape-J, the sequence transmitted by the CLR DISPLAY key, to the editor commands **DjdG\$** (delete to end of line, go down one line, delete to end of file).
- 18 maps escape-A, the sequence transmitted by the UP ARROW key, to the sequence **^V** (causes cursor to move one space to the right) when it is used in insert mode (**map!** causes a key to be defined in insert mode only).
- 19 maps escape-D, the sequence transmitted by the LEFT ARROW key, to the sequence **^H** (causes cursor to move one space to the left) when it is used in insert mode.
- 20 maps escape-C, the sequence transmitted by the RIGHT ARROW key, to the sequence **^V** (causes cursor to move one space to the right) when it is used in insert mode.
- 21 maps escape-B, the sequence transmitted by the DOWN ARROW key, to the sequence **^M** (carriage-return) when it is used in insert mode.
- 22 maps escape-L, the sequence transmitted by the INS LINE key, to the sequence **^M** (carriage-return) when it is used in insert mode. This makes the INS LINE key have the same definition in *vi* as it has in REMOTE mode.
- 23 maps escape-Q, the sequence often transmitted by the INS CHAR key, to the escape key during insert mode.
- 24 maps escape-R, the sequence often transmitted by the INS CHAR key, to the escape key during insert mode.

Table of Contents

The Ed Editor

Creating an Ordinary File	1
Getting Acquainted with Ed.....	2
Invoking Ed.....	2
Prompting.....	3
Error Messages.....	3
Moving Around in the File.....	3
Line Pointers.....	4
Pointer to the Current Line.....	4
Pointer to the Last Line.....	6
Setting Pointers to Lines.....	7
Searching for Strings.....	8
Forward Searches.....	8
Backward Searches.....	9
Repeating a Search.....	9
Line Number Arithmetic with Searches.....	9
Using Metacharacters With Searches.....	10
Adding, Deleting, and Correcting Text.....	12
Printing Lines.....	13
Appending Text.....	14
Inserting Text.....	15
Deleting Text.....	15
Undoing Commands.....	16
Changing Lines.....	16
Moving Lines.....	17
Copying Lines.....	18
Modifying Text Within a Line.....	19
Making Commands Effective Globally.....	22
Joining Lines Together.....	25
Splitting Lines Apart.....	25
Special Ed Commands.....	26
Finding the Currently Remembered File Name.....	26
Writing Buffer Text Onto a File.....	27
Reading Files Into the Buffer.....	28
Editing Other Files.....	29
Silencing the Character Counts.....	30
Encrypting and Decrypting Text.....	31
The Shell Interface.....	33
Escaping to the Shell Temporarily.....	33
Exiting the Editor.....	34
Miscellaneous Topics.....	35
Interrupting the Editor.....	35
Editing Scripts.....	35

The Ed Editor

Ed is an interactive, line-oriented text editor. Its purpose is to enable you to create ordinary files, and to add to, delete, or modify the text in those files.

Creating an Ordinary File

The remainder of this chapter contains several examples illustrating *ed* commands. These examples are more instructive if you try each of them on some text of your own. Thus, create an ordinary file by typing in the commands and text shown below in **bold** (portions of the example text shown below are taken from *A User Guide to the UNIX System*, by Rebecca Thomas and Jean Yates).

```
$ ed testfile
?testfile
a
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the editor interprets
your input as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \\*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
.
w
461
q
$
```

Be sure to type in the text exactly as it is shown above. The mistakes are corrected later in the examples.

Getting Acquainted with Ed

Material Covered:

ed	command; invokes <i>ed</i> without a file name argument;
ed file	command; invokes <i>ed</i> with a file name argument;
P	command; enables/disables <i>ed</i> prompt (*);
h	command; explains the last question mark given by <i>ed</i> ;
H	command; enables/disables verbose error messages; explains the last question mark given by <i>ed</i> , and all future question marks.

Invoking Ed

Ed can be invoked in one of two ways. The first is to simply type **ed**, followed by [RETURN]. For example,

```
$ ed
```

invokes *ed* without a file name argument. When invoking *ed* this way, you must specify the file you want to edit with a separate command. It is more common to invoke *ed* by typing

```
$ ed filename
```

where *filename* is the name of the file you want to edit. This combines the two separate commands mentioned above into a single command.

Ed responds differently depending on whether or not the file already exists. Try creating a new file called **newfile**:

```
$ ed newfile
?newfile
```

Ed responds with "?newfile", which means that *ed* cannot find that file in your working directory. This is to be expected, since the file does not yet exist. *Ed* is now waiting for your commands to create and edit **newfile**.

If the file already exists, *ed* reads its contents into a buffer named **/tmp/e#**, where **#** is the number of the process running *ed*. *Ed* then displays a count of the characters contained in that file. You have a file called **testfile** in your working directory. You are probably still in *ed* from the previous example, so type **q**[RETURN] to exit *ed*, then edit **testfile** by typing

```
$ ed testfile
461
```

Ed tells you that **testfile** currently contains 461 characters. Do not exit *ed* this time, but leave it in its current state. The examples that follow pick up where you left off above.

Prompting

One of the most noticeable features of *ed* is its lack of prompts. When you type in a command, *ed* attempts to execute it, and, if successful, *ed* returns silently to you for another command. If an error is encountered, or a command cannot be executed for some reason, *ed* prints a question mark, and then silently waits for you to figure out the problem.

Many people find this silence desirable, but for those who do not, there are commands that make *ed* more friendly. The **P** command causes *ed* to prompt you with an asterisk (*). Executing the **P** command again turns off the prompt. By default, *ed*'s prompt is disabled.

Error Messages

As mentioned above, *ed*'s default error message is a single question mark (?). As you gain experience with *ed*, these question marks become easier to interpret, but for the beginning user, it can be somewhat difficult to discover the problem. Fortunately, *ed* provides commands to eliminate this vagueness. The **h** command explains the last question mark printed by *ed*. The **H** command also explains the last question mark, but also causes a more descriptive explanation of the problem to replace all future question marks. Executing the **H** command again disables the descriptive explanation.

Moving Around in the File

Material Covered:

.	(dot) pointer to the current line;
=	operator; yields line number;
p	command; prints specific lines;
+ <i>n</i>	operator; increments dot by <i>n</i> ; default <i>n</i> = 1;
- <i>n</i>	operator; decrements dot by <i>n</i> ; default <i>n</i> = 1;
\$	pointer to the last line of the file;
,	shorthand notation for the range "1,\$";
;	shorthand notation for the range ".,\$";
k	command; creates a pointer to a specific line;
/ ... /	command; initiates a forward search for the string of characters enclosed between the slashes;
? ... ?	command; initiates a backward search for the string of characters enclosed between the question marks;
.	metacharacter; matches any single character when used in a search string;
\ <i>n</i>	metacharacter; strips away the special meaning (if any) of the character <i>n</i> when used in a search string;
\$	metacharacter; when specified as the last character in a search string, matches the string at the end of a line;
^	metacharacter; when specified as the first character in a search string, matches the string at the beginning of a line;
<i>n</i> *	metacharacter; matches zero or more adjacent occurrences of the character <i>n</i> when used in a search string;
[...]	metacharacters; match any one of the characters enclosed between them when used in a search string;
^	metacharacter; stands for "any character except" when specified as the first character inside [...], causing the braces to match any one character <i>not</i> en-

\{... \} closed between them;
metacharacters; match a specified number of occurrences of the single character enclosed between them when used in a search string.

Your position in a file is always relative to a specific line. *Ed* does not provide commands that move you from character to character. There are five commands that enable you to reference specific lines in a file.

Line Pointers

Of the five commands mentioned above, three are pointers to specific lines in the file.

Pointer to the Current Line

Ed maintains a line pointer called dot (.), which always points to the current line in the file. The current line is defined to be the last line affected by an *ed* command. The following table lists some of the more common *ed* operations, and the value of dot after these operations have been performed:

After this operation...	Dot points to...
Invoking <i>ed</i>	Last line of file.
Search for pattern	Closest line containing pattern, relative to your previous position.
Delete last line of file	New last line of file.
Delete line(s) other than last line	Line following last deleted.
Appending, inserting, or changing text	Last line entered.
Read from a file	Last line read in.
Write to a file	Your previous position; dot is not changed.
Substitute new text for old text	Last line affected by substitution.
Execute a shell command	Your previous position; dot is not changed.
Set a line pointer	Your previous position; dot is not changed.
Any unsuccessful or erroneous command	Your previous position; dot is not changed.

Dot can be used as a line number argument for *ed* commands. Assuming you are still editing **test-file**, type

.p
to the file.

The **p** command prints specific lines from the *ed* buffer, thus **.p** prints the current line. Note that dot is automatically set to the last line of the file when you first begin editing. You can also specify a range of line numbers with dot. For example,

.-3,.p

important to note that *ed* always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the last four lines of the file. Has the value of dot changed? No, because the last line affected by the **p** command was still the last line of the file. Now try

.-5,.-3p

your input to the text located in a special buffer where *ed* keeps a copy of the file you are editing. It is `*.` important to note that *ed* always makes changes to the

which prints the fifth line before dot to the third line before dot. What is dot's value now? Find out by typing

.p

important to note that *ed* always makes changes to the

Dot is now set to the last line affected by the previous **p** command.

Note that dot need not be typed when specifying ranges. Whenever *ed* sees the + and - operators, *ed* assumes that they refer to the current value of dot. For example,

-2, + 2p

your input to the text located in a special buffer where *ed* keeps a copy of the file you are editing. It is `*.` important to note that *ed* always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes

prints the range of lines from two lines before dot to two lines after dot. Dot is set to the last line printed.

The + and - operators can be used independently to increment or decrement dot by one, respectively. For example, the command

--, + p

important to note that *ed* always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the range of lines from dot decremented by two to dot incremented by one. Also, you can step forward through your text, one line at a time, with a series of plus signs, or step backward with a series of minus signs. Note that [RETURN] is equivalent to +. [RETURN] increments dot by one and prints the resulting current line.

The **p** command provides one other shortcut. Whenever a line number, or one or more operators pointing to a line, appear on a line by themselves, the **p** command is assumed. Some examples are:

```
8
original file are not changed until you write the changes
----
ed keeps a copy of the file you are editing. It is \ \ *.
+ +
copy of yourrr file in the buffer. The contents of the
```

If a range appears on a line by itself, only the last line of the range is printed. For example,

```
-, +
original file are not changed until you write the changes
```

You can find out the current value of dot by typing

```
. =
8
```

which tells you that dot is currently pointing to the eighth line of the file.

Note that you cannot manually set the value of dot. A command like

```
. = 6
?
```

produces an error. *Ed* reserves to itself the right to change the value of dot, although you may indirectly change dot's value through *ed* commands.

Pointer to the Last Line

Ed also maintains a pointer, called **\$**, which always points to the last line of the file. For example,

```
$
to the file.
```

prints the last line of the file. **\$** can also be used in ranges, as in

```
1,$-6p
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
```

which prints the first three lines of **testfile**. Also,

```
+4,$p
```

copy of yourrr file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the last three lines of the file. Note that the + and - operators can apply to \$ only when \$ is explicitly typed. By themselves, + and - always apply to dot.

You can find out the value of \$ by typing

```
$=  
9
```

which tells you that the ninth line is the last line in the file. Note that = does not change the value of dot.

The value of \$ changes only when a command creates a new last line. \$ is not user-settable.

Because the "1,\$" and ".,\$" ranges are so commonly used when editing with *ed*, *ed* provides a shorthand notation for each range. The comma can be used in place of "1,\$", so that ,p prints all the lines in the file. Also, the semicolon means the same thing as ".,\$", so ;p prints all the lines from the current line to the end of the file.

Setting Pointers to Lines

The **k** command creates a pointer to a specific line, so you can reference that line without knowing its line number. The pointer name must be a lower-case letter. Creating a pointer does not change the value of dot. For example,

```
.  
to the file.  
-4ka  
-2kb  
.  
to the file.
```

creates two pointers, a and b, which point to the fourth line before dot, and the second line before dot, respectively. Note that dot does not change.

To reference a line with a line pointer you have created, precede its letter name with a single quote ('), as in

```
^a,^bp  
ed keeps a copy of the file you are editing. It is \ \ *.  
important to note that ed always makes changes to the  
copy of yourrr file in the buffer. The contents of the
```

which prints all lines from the line pointed to by a to the line pointed to by b.

A pointer set by the **k** command always points to the same line, even if that line's line number changes. Thus, the **k** command does not create pointers to specific line numbers, but to specific lines.

Once a pointer has been created, the only way to delete it is to delete the line it points to. Otherwise, that pointer continues to exist until your editing session is over. You can, however, re-assign a pointer to another line, as in

```
^ap
ed keeps a copy of the file you are editing. It is \\ \\ *.
2ka
^ap
text entry mode. In command mode, the edytor interprets
```

which re-assigns a to the second line of the file.

You can find out the current line number of a pointer by typing

```
^a=
2
^b=
7
```

which tells you that a is currently pointing to line number 2, and b is currently pointing to line number 7.

Searching for Strings

Ed provides a facility which enables you to search for a particular string of characters in your file. A string of characters searched for in this manner is called a *pattern*.

Forward Searches

To initiate a forward search, enclose the pattern between two slashes, and press [RETURN]. For example,

```
/unput/
your unput as a command. In text entry mode, ed adds
```

searches for the pattern "unput". If the pattern is found, dot is set to the line containing the pattern, and the line is printed on your screen. An unsuccessful search looks like this:

```
/bob/
?
```

The value of dot is unchanged.

Ed searches forward in your file, starting with the line following the current line. If your pattern has not been found by the time *ed* gets to the end of the file, *ed* wraps around to the beginning of your file and continues looking. *Ed* searches until the pattern is found, or until *ed* reaches the line prior to the starting line of the search.

Backward Searches

You can search backwards in your file by enclosing the pattern between two question marks. For example,

```
?file?  
to the file.
```

searches backwards from the current line, looking for a line containing the string "file". *Ed* found the pattern after wrapping around to the end of the file.

Repeating a Search

Ed remembers the last pattern that was matched. Thus, if you want to repeat a search, you simply type two slashes or question marks. The pattern itself need not be re-typed. For example,

```
?file?  
original file are not changed until you write the changes  
??  
copy of yourrr file in the buffer. The contents of the  
??  
ed keeps a copy of the file you are editing. It is \ \ *.
```

initiates a backward search for the pattern "file", then finds the next two instances of "file". Note that a repeated search need not be in the same direction as the initial search. For example,

```
/buffer/  
copy of yourrr file in the buffer. The contents of the  
??  
your input to the text located in a special buffer where
```

initiates a forward search for "buffer", then repeats the search backwards.

Line Number Arithmetic with Searches

The + and - operators can be used with searches to position yourself at specific lines. For example,

```
/note/ +  
copy of yourrr file in the buffer. The contents of the
```

searches forward for a line containing "note", and positions you on the following line. Also,

```
?text?
```

```
your input to the text located in a special buffer where
```

```
??--
```

```
The ed editor operates in two modes: command mode and
```

searches backwards for the second line containing "text", and positions you two lines before it.

Note that, although searches have wrap-around capabilities, the + and - operators do not. Thus, an error results if a + or - operator attempts to increment or decrement dot to values greater than \$, or less than one.

The = operator can be used with forward and backward searches to find the line number referred to by the search, as in

```
/unput/=
3
```

Note that dot is not set to the line containing "unput" in the last example, because = does not change the value of dot.

Using Metacharacters With Searches

There are several characters that have special meaning within the context of a search. These characters, consisting of ., *, [,], ^, \$, \, \{, and \}, are called metacharacters.

The . metacharacter matches any single character except a new-line. Thus, the search

```
/.nput/
your unput as a command. In text entry mode, ed adds
//
your input to the text located in a special buffer where
```

first matches "unput" in line 3, and then, when repeated, matches "input" in line 4.

The * metacharacter matches zero or more occurrences of the character immediately preceding it. For example,

```
/your*/
ed keeps a copy of the file you are editing. It is\ \*.
```

matches "you" in the line displayed. *Ed* stops searching when it finds the first string of characters that matches the given pattern. Thus, "your" or "yourrr" can also be matched with the above search, depending on the current line when the search is initiated.

The last example shows that, even though an "r" is explicitly typed in `/your*/`, there need not be an "r" in the string of characters that are actually matched. This is because zero occurrences of the preceding character is considered a legal match when the asterisk is used. Keeping this in mind, consider the search `/r*/`. Is it useful? No, because zero or more r's can be found on every

line in the file. If you want to search for one or more r's, type `/rr*/`.

The `\{` and `\}` metacharacters enable you to control how many occurrences of a particular character are matched. For example, the search `/g\{4\}/` finds a string of four g's. The integer between the two metacharacters specifies how many instances of the preceding character are to be matched. Note that this construct matches *exactly* four g's, not four or more. Thus, "yourrr" can be matched by

```
/r\{3\}/  
copy of yourrr file in the buffer. The contents of the
```

If you put a comma after the integer, the `\{ ... \}` construct matches *at least* the specified number of occurrences. For example, `/33.3\{4,\}/` matches "33.", followed by at least four 3's. Finally, two integers separated by a comma can be placed in the `\{ ... \}` construct to define an inclusive range which specifies the number of occurrences to match. An example is `/-13\{2,5\}1-/`, which matches -1331-, -13331-, -133331-, or -1333331-.

The `[` and `]` metacharacters match any one of the characters enclosed between them. For example, `/h[iaut]/` matches "hit", "hat", or "hut". A range of characters can be specified by typing the beginning and ending character of the range, separated by a minus sign. An example is `/[a-zA-Z][0-9][0-9]*/`, which searches for a single upper- or lower-case letter, followed by one or more digits (the `*` applies only to the `[...]` construct immediately preceding it). The minus sign loses its special meaning within the `[...]` construct if it occurs at the beginning (after an initial `^`, if any), or at the end of the character list.

If the first character after the left bracket is a circumflex (`^`), then the `[...]` construct matches any single character *not* included between the brackets. For example, `/[^0-9][^0-9]*/` matches one or more occurrences of any character except a digit. The `^` has special meaning in the `[...]` construct only when it is the first character after the left bracket.

Note that the metacharacters `.`, `*`, `[`, `\`, `$`, `\{`, and `\}` have no special meaning when listed within the `[...]` construct. Also, the right bracket does not terminate the construct if it is the first character listed after the left bracket (after an initial `^`, if any). For example, `/[a-r]/` searches for a single right bracket, or a lower-case letter in the range a through r.

The `^` is also special when typed at the beginning of a string within a search, and requires that the string be matched at the beginning of a line. For example,

```
/^ed/  
ed keeps a copy of the file you are editing. It is \ \ *.
```

searches for a line beginning with "ed". The `^` is special only when typed at the beginning of a search string. If `^` is embedded in a pattern, or if it is the only character in the pattern, it is matched literally.

The various ways to use `^` can be illustrated with `/^[^a-z]/`. The first `^` means "match the following pattern at the beginning of a line". The second `^` is literal; it has no special meaning. The third `^`, as the first character inside the brackets, means "match any single character except". Thus, this search looks for a `^`, followed by any single character except a lower-case letter, occurring at the beginning of a line.

The **\$** metacharacter is special when typed at the end of a string within a search, and requires that the string be matched at the end of a line. For example,

```
/and$
```

The ed editor operates in two modes: command mode and

searches for a line ending with "and". Also, **/~TEST\$** searches for a line consisting of the single word "TEST".

The **\$** is special only when typed at the end of a search string. When embedded in the string, the **\$** is matched literally.

The **** (backslash) metacharacter is used to strip away the special meaning associated with a metacharacter. This is useful when you need to match a metacharacter literally in a string. To strip away the special meaning of a metacharacter, simply precede it with ****. For example,

```
/\\\*\\.$
```

ed keeps a copy of the file you are editing. It is *****.

matches the string "*****" at the end of a line. Note that **** itself must also be preceded with **** to be matched literally. If you attempt to match the string without using the **** (as in **/*.\$**), ed interprets the search to mean "search for zero or more occurrences of a backslash followed by any single character at the end of a line", which is obviously not what you want. Also,

```
/file\\.$
```

to the file.

matches "file." at the end of a line. If you are ever in doubt about whether or not a character has special meaning, it is safe to precede it with **** just to be sure. If the character has no special meaning, then the **** is ignored.

Adding, Deleting, and Correcting Text

Material Covered:

l	command; list specific lines;
n	command; print lines with line numbers;
a	command; append lines of text after current line;
i	command; insert lines of text before current line;
d	command; delete lines of text;
c	command; change lines of text;
m	command; move lines of text;
t	command; copy lines of text;
j	command; join lines together;
s	command; substitute new text for old text;
g	command; global; perform command list on selected lines of entire file;
G	command; interactive global; on each line selected in the entire file, perform a user-specified command;
v	command; global; perform command list on all lines <i>not</i> selected in the entire

	file;
V	command; interactive global; on each line <i>not</i> selected in the entire file, perform one user-specified command;
u	command; reverse the most recent modification to the buffer;
\(... \)	metacharacters; used in left-hand side of s command to break up pattern into pieces that can be referenced individually;
%	metacharacter; used in right-hand side of s command to duplicate right-hand side of most recent s command;
&	metacharacter; used in right-hand side of s command to duplicate left-hand side of same s command.

Printing Lines

Besides **p**, there are two other commands that enable you to print specific lines in the *ed* buffer. The **l** (list) command is similar to **p**, but gives you slightly more information. The **l** command enables you to see characters that are normally invisible. Backspace and tab are represented by overstrikes, and other invisible characters, such as bell, vertical tab, and formfeed, are represented by `\nnn`, where *nnn* is the octal equivalent of the character in the ASCII character set.

The **l** command also breaks long lines into smaller lines of 72 characters each. Thus, if you have lines of text in a file that are longer than 72 characters, **l** breaks them down into 72-character lines so they can fit on your screen. A `\` is printed at the end of each line that is broken.

Print out the contents of **testfile** with the **l** command, and look for any invisible characters:

```
,l
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is\ \*.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

If you did not make any typing errors that could produce invisible characters, the output looks as shown above. Note that a carriage return and a line feed are not considered invisible, since the placement of text on your screen indicates their presence.

Since some invisible characters can cause strange terminal behavior, you almost always want to eliminate them from your text. This is where the **l** command can save you time and effort by making these characters visible.

The **n** (number) command also enables you to print specific lines, but differs from **p** and **l** in that each line is preceded by its line number and a tab character. Try printing out the contents of **testfile** with **n**:

```
,n
1      The ed editor operates in two modes: command mode and
2      text entry mode. In command mode, the edytor interprets
3      your unput as a command. In text entry mode, ed adds
4      your input to the text located in a special buffer where
5      ed keeps a copy of the file you are editing. It is \ \ *.
6      important to note that ed always makes changes to the
7      copy of yourrr file in the buffer. The contents of the
8      original file are not changed until you write the changes
9      to the file.
```

Note that the line numbers and tab characters are display enhancements only, and do not become part of the text in the *ed* buffer.

The **p** command is the most common command used to print lines in the *ed* buffer. Keep in mind, however, that wherever it is legal to use the **p** command, the **l** and **n** commands may also be used. The **l** and **n** commands leave dot pointing to the last line printed.

Appending Text

The **a** (append) command appends one or more lines of text after the specified line. By default, the lines of text are added after line dot. Dot is left pointing to the last line appended. After the **a** command is typed, everything you enter is appended to the specified line. To stop appending text, type a period at the beginning of a line, all by itself. This terminates the **a** command, and returns you to command mode. For example,

```
0a
The ed editor is a simple, easy-to-use text editor.
.
1,3p
The ed editor is a simple, easy-to-use text editor.
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
```

The **a** command is one of the few *ed* commands that accepts 0 as a line number, enabling you to add text to the beginning of the file, as above. Note that the period at the beginning of an empty line terminates the appended text. The following example can easily occur by forgetting to type the terminating period (*do not try this example!*):

```
$a
It is always comforting to know that your original
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
.
$-7,$p
original file are not changed until you write the changes
to the file.
It is always comforting to know that your original
```

```
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
```

This poor user typed in the three lines of text that he wanted to append to the end of his file, and then attempted to print out the results. *Ed*, however, was still appending text, and calmly added the user's commands to the file. The user finally realized his mistake, typed the solitary period, and printed out the last eight lines of his file, three of which were the three commands he attempted to execute. The moral of the story is: **REMEMBER THE PERIOD!**

If you type the **a** command and then change your mind, simply type a solitary period on the next line. This terminates the **a** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **a** command.

Inserting Text

The **i** (insert) command is similar to the **a** command, except that the added text is inserted before the specified line. By default, the added text is inserted before line dot. Dot is left pointing to the last line inserted. Like the **a** command, the inserted text is terminated by a solitary period at the beginning of a line. For example,

```
2i
Also, it takes very little time to learn.
.
1,3p
The ed editor is a simple, easy-to-use text editor.
Also, it takes very little time to learn.
The ed editor operates in two modes: command mode and
```

If you type the **i** command and then change your mind, simply type a solitary period on the next line. This terminates the **i** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **i** command.

Deleting Text

The **d** (delete) command deletes one or more lines of text from the file. If no lines are specified, line dot is deleted. After a deletion, dot is left pointing to the line following the last line deleted. If the last line of the file is deleted, dot points to the new last line. For example,

```
$d
a
on top of the original contents of your file.
.
$-1,$p
original file are not changed until you write the changes
on top of the original contents of your file.
```

The current last line is deleted, and a new one is typed in its place using the **a** command. The **a** command is used because dot is left pointing at the new last line after the deletion. Thus, it is convenient to append after dot to create the desired last line.

The **d** command can delete several lines at once by specifying a range of lines, as follows:

3,6d

,p

The ed editor is a simple, easy-to-use text editor.

Also, it takes very little time to learn.

ed keeps a copy of the file you are editing. It is `*`.

important to note that ed always makes changes to the

copy of yourrr file in the buffer. The contents of the

original file are not changed until you write the changes

on top of the original contents of your file.

This shows that **testfile** currently contains 7 lines of text, since lines 3 through 6 have been deleted.

Undoing Commands

The **u** (undo) command reverses the effect of the most recent command that made a change to any of the text in the buffer. Use it now to restore the four lines you just deleted:

u

,p

The ed editor is a simple, easy-to-use text editor.

Also, it takes very little time to learn.

The ed editor operates in two modes: command mode and

text entry mode. In command mode, the edytor interprets

your unput as a command. In text entry mode, ed adds

your input to the text located in a special buffer where

ed keeps a copy of the file you are editing. It is `*`.

important to note that ed always makes changes to the

copy of yourrr file in the buffer. The contents of the

original file are not changed until you write the changes

on top of the original contents of your file.

Note that the **u** command reverses only the most recent command that modified text. Commands that have been succeeded with one or more other commands cannot be reversed with **u**. Besides **d**, **u** also reverses the **a**, **i**, **c**, **g**, **G**, **v**, **V**, **j**, **m**, **r**, **s**, and **t** commands. Dot is left pointing to the last line affected by the reversal.

Changing Lines

The **c** (change) command replaces one or more lines with the text you specify. The **c** command is a combination of the **d** and **i** commands, in that the specified lines are deleted, and the text you type in is inserted in their place. Like the **a** and **i** commands, the replacement text is terminated with a solitary period at the beginning of a line. Dot is left pointing to the last line of replacement text typed in. For example,

1,2c

The ed editor is easy to learn and easy to use.

.

1,3p

The ed editor is easy to learn and easy to use.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets

In this example, the first two lines are deleted and replaced with a single line. Of course, you can also replace a single line with several lines, as in

2c

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and

.

1,/text/p

The ed editor is easy to learn and easy to use.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets

which replaces the second line of the file with five lines.

If you type the **c** command and then change your mind, simply type a solitary period at the beginning of the next line. This terminates the **c** command with no changes made, and leaves dot pointing to the first line you specified when you typed the **c** command.

Moving Lines

The **m** (move) command moves one or more lines to a new position in the file. By default, **m** moves line dot. Dot is left pointing to the last line moved. For example,

2,5m\$

.p

The ed editor is easy to learn and easy to use.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets your unput as a command. In text entry mode, ed adds your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is `*.` important to note that ed always makes changes to the copy of yourrr file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done

with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

which moves lines two through five to the end of the file. Note that **m** appends the moved lines after the specified line. Thus, line number zero is legal as a destination line number, enabling you to move lines to the beginning of the file. The destination line cannot be one of the lines being moved.

Note that the **m** command, as well as any command that accepts line number arguments, accepts pattern searches and line pointers (set by the **k** command) to reference specific lines. For example, **2,/user/+++m\$** has the same effect as **2,5m\$** in the previous example. Using pattern searches and line pointers becomes more valuable when you edit large files.

Copying Lines

The **t** command copies one or more lines and places the copy at a specified location in the file. By default, **t** copies line dot. Dot is left pointing to the last line copied, in its new location. For example,

```
1t$  
.-4,$-1t1
```

```
,p
```

The ed editor is easy to learn and easy to use.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the editor interprets your input as a command. In text entry mode, ed adds your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is \ \ *. important to note that ed always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor is easy to learn and easy to use.

This example copied the first line and moved it to the end of the file. Then, the four lines before the new last line were copied and moved after the first line of the file, producing the text shown above.

The only difference between the **m** and **t** commands is that **t** copies the indicated lines and moves them to a new position, leaving the original lines intact. The **m** command moves the specified lines from their original position to a new position. No new text is created.

Modifying Text Within a Line

The **s** (substitute) command is the only *ed* command that enables you to change one or more characters within a line, without having to type the line over again. By default, **s** modifies text on line dot. Dot is left pointing to the last line in which a modification has occurred.

The **s** command enables you to correct the mistakes in your file. Of course, you could use the **d** and **i** commands and re-type each line containing an error, but that is more work than is necessary. For example,

```
/textct/  
textct entry mode. In command mode, the edytor interprets  
s/textct/text/p  
text entry mode. In command mode, the edytor interprets
```

All **s** command lines are of the form

```
s/replace this/with this/
```

Thus, the above example first searches for the line containing the string "textct", and then replaces "textct" with "text" on that line. Note that the **p** command is appended to the **s** command to verify that the intended substitution took place.

Note that the pattern search in the previous example can be included on the **s** command line. The **s** command accepts one line number, to perform a specific replacement on a single line, or two line numbers separated by a comma, to perform a replacement on a range of lines. For example,

```
/unput/s//input/p  
your input as a command. In text entry mode, ed adds
```

which searches for the pattern "unput" and replaces it with "input". Another feature is illustrated in the above example. Note that the *replace this* portion of the **s** command is empty. This is because the *replace this* portion of the **s** command is a pattern search, just like those discussed under *Searching for Patterns*. You recall from that discussion that *ed* remembers the last pattern you searched for. Thus, since "unput" is the last pattern you searched for, it need not be re-typed in the **s** command. *Ed* remembers the pattern and supplies it for you.

Metacharacters can be used in the **s** command. The *replace this* portion recognizes all the metacharacters discussed under *Searching for Patterns*, plus two additional metacharacters, **\(** and **\)**. These two metacharacters are used to break up the *replace this* portion into pieces that can be referenced individually. For example, in line 1 of the file, suppose you want to interchange the phrases "easy to learn" and "easy to use". The obvious way to do that is to re-type the entire line, but there is an easier way:

```
1s/\(ea.*rn\) and \(ea.*se\)\/\2 and \1/p  
The ed editor is easy to use and easy to learn.
```

Although it is hard to read, it is handy to be able to define pieces of patterns and rearrange them in the *with this* portion. In the above example, the entire *replace this* portion matches "easy to learn and easy to use". The first **\(... \)** matches "easy to learn", and the second **\(... \)** matches "easy to use". These pieces are referred to in the *with this* portion with **\n**, where **n** refers to the n-th occurrence of a **\(... \)** pair in the *replace this* portion, counting from the left.

Thus, the *with this* portion interchanges the two pieces defined in the *replace this* portion.

Here is another example. Suppose you have a file containing information like

```
Alderson, Mike
Anderson, David
Belford, John
Donally, Kyle
```

```
.
```

and you want to rearrange each name so that the first name is first, followed by the last name. Re-typing each line could take forever, but the task is easy using the `\(` and `\)` metacharacters. The command

```
,s/\([^\,]*\), \(.*\)/\2 \1/
```

does the job. The first `\(... \)` pair matches any number of characters except a comma – the last name. The comma-space between each last and first name is explicitly matched. Finally, the second `\(... \)` pair matches any number of any characters – the first name. These pieces are rearranged in the *with this* portion.

Note that the two portions of an `s` command do not have to be delimited by slashes. You can use any character except a space or a new-line, as long as you use the same character throughout the command line. For example, the previous example can be made a bit more clear by using a capital `o` as the delimiter:

```
,sO\([^\,]*\), \(.*\)O\2 \1O
```

You must be careful to choose a delimiter that is not already used in the `s` command line.

The *with this* portion of the `s` command recognizes only the `\` metacharacter, plus two new metacharacters, `&` and `%`. All other metacharacters previously discussed are interpreted literally in this portion.

The `&` metacharacter is recognized only in the *with this* portion, and stands for whatever is matched by the pattern in the *replace this* portion. For example,

```
2s/done/& quickly/p
```

```
It was designed to enable the user to get his work done quickly
```

The `&` stands for whatever pattern is matched in the *replace this* portion, so it stands for "done" in this example. Thus, this example replaces "done" with "done quickly". As another example, first add the line "ed is great" to the end of the file:

```
$a  
ed is great
```

Now use **&** to create two sentences out of one:

```
$s./&? &!/p
ed is great? ed is great!
```

The **&** must be preceded by **** to be interpreted literally.

The **%** is also recognized only in the *with this* portion, and stands for whatever was specified in the *with this* portion of the last **s** command that was executed. For example,

```
1s/ed editor/ed text editor/p
The ed text editor is easy to use and easy to learn.
/ed editor/s//%/p
The ed text editor operates in two modes: command mode and
//s//%/p
The ed text editor is easy to learn and easy to use.
```

In the first **s** command, the *with this* portion has to be explicitly typed out. Thereafter, a **%** is the only character appearing in the *with this* portion, and stands for "ed text editor". Since the replacement text is the same for the remaining **s** commands, it does not need to be re-typed. Note also how **ed**'s pattern memory is utilized, especially in the last **s** command above.

The **%** is special only when it is the only character in the *with this* portion. If **%** is included in a string of one or more characters, it is no longer special. You can also precede the **%** with a **** to cause literal interpretation.

Now that you know all about the **s** command, you can go through and fix the remaining errors in your file. Here are some suggestions:

```
/edy/s//edi/p
text entry mode. In command mode, the editor interprets
+ 3s/\ \ \ \ \ * \.//p
ed keeps a copy of the file you are editing. It is
/yourrr/s//your/p
copy of your file in the buffer. The contents of the
```

Note that, in the second **s** command above, the *with this* portion is empty. This is legal, and is often used when you want to replace erroneous text with nothing at all.

Finally, note that the **s** command operates only on the first occurrence of a pattern on a specified line. Thus, if there are two or more patterns on a line that are identical to the pattern specified in the *replace this* portion, only the first occurrence is actually replaced. The **s** command must be re-executed once for each additional pattern that is to be replaced on the same line.

The **s** command must replace text on at least one of the addressed lines, or **ed** prints a question mark.

Making Commands Effective Globally

The **g** (global) command is used to execute one or more commands on several lines. The lines on which the commands are to be executed are usually specified by pattern searches. The form of a **g** command is

```
x,yg/pattern/command list
```

where *x* and *y* are optional line number arguments, *pattern* is the pattern to be searched for, and *command list* is the list of one or more commands to be executed on each line containing *pattern*. If *x* and *y* are missing, "1,\$" is assumed.

The **g** command first marks every line containing the specified pattern. Then, dot is successively set to each marked line, and the list of commands is executed. If only one command is specified, it is placed on the same line as the **g** command. If several commands are specified, the first command is placed on the same line as the **g** command, and all other commands are placed on the following lines. Every line of a multi-line command list is terminated by `\` except the last. Ending a line with `\` in this way quotes the following new-line, and hides it from the **g** command, thus preventing the new-line from terminating the **g** command prematurely. If no commands are specified, the **p** command is assumed. Any command except **g**, **G**, **v**, and **V** can be used in the command list.

The **g** command can be used as a modifier for the **s** command, enabling the **s** command to replace all the occurrences of a particular pattern on a line, instead of just the first. For example,

```
$s/ed/The & editor/gp  
The ed editor is great? The ed editor is great!
```

which replaces both instances of "ed" on the last line with "The ed editor". The **g** command is often used with the **s** command in this way to avoid having to repeat the **s** command once for every additional pattern you want to change on a line. Note that, if the **p** command is omitted, the line is not printed after the substitution is done.

The **g** command becomes more powerful when you specify more than one command to be executed. For example, suppose that you want to change every instance of the string "ed" to "ED", and then mark every line on which the substitution occurs by preceding the line with a series of asterisks. This can be done by typing

```
g/ed/s//ED/g\  
i\  
***  
.p  
***  
The ED text EDitor is easy to use and easy to learn.  
***  
It was designED to enable the user to get his work done quickly  
with the least possible amount of interference from the  
***  
EDitor. This is evident in the lack of prompts and the  
curt error messages.  
***  
The ED text EDitor operates in two modes: command mode and
```

text entry mode. In command mode, the EDitor interprets

your input as a command. In text entry mode, ED adds

your input to the text locatED in a special buffer where

ED keeps a copy of the file you are EDiting. It is

important to note that ED always makes changes to the
copy of your file in the buffer. The contents of the

original file are not changED until you write the changes
on top of the original contents of your file.

It was designED to enable the user to get his work done
with the least possible amount of interference from the

EDitor. This is evident in the lack of prompts and the
curt error messages.

The ED text EDitor is easy to learn and easy to use.

The ED EDitor is great? The ED EDitor is great!

This example, though not very useful, illustrates how the **g** command can be used to perform a script of *ed* commands on specific lines. Note that the **g** command accepts as input all lines up to and including the first line that does not end in `\`. Thus, the first line that is not part of the **g** command above is the line containing `,p`. Note also that the period that usually must be typed to end the **i** command is not necessary if the line containing the period is also the last line of the **g** command. Thus, the period, along with the line on which it is typed, can be omitted.

A **g** command can be included in a **g** command list only when it is part of another command, as illustrated in the last example. It is illegal to try to nest command lists by specifying **g** command lists within other command lists.

The **v** command is identical to the **g** command, except that the command list is executed on all lines that do *not* contain the specified pattern.

If the results of a **g** command are not exactly what you had in mind, you can use the **u** command to restore your text to its previous state.

u
,p
The ed text editor is easy to use and easy to learn.
It was designed to enable the user to get his work done quickly
with the least possible amount of interference from the
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor operates in two modes: command mode and
text entry mode. In command mode, the editor interprets

your input as a command. In text entry mode, ed adds your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is important to note that ed always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed text editor is easy to learn and easy to use.

The ed editor is great? The ed editor is great!

Note that the **u** command also reverses itself, so you can follow one **u** command with another to get back text that you have already reversed.

The **G** (interactive global) command is used when you have one command to execute on each line containing a specific pattern, but this command varies depending on the line. The **g** or **v** command is not appropriate in this case, since the command list for these commands is constant.

The **G** command is invoked in the form

```
x,yG/pattern/
```

where *x* and *y* are line number arguments (if not specified, "1,\$" is assumed), and *pattern* is the particular pattern you want to match in a line. **G** first marks every line containing a string that matches *pattern*. Then, dot is successively set to each marked line, and the resulting current line is printed on your screen. After the current line is printed, **G** waits for you to enter any single command, and the command you enter is executed. You may specify any command except the **a**, **i**, **c**, **g**, **G**, **v**, or **V** commands. Note that your command can address and affect lines other than the current line. A new-line is interpreted to be a null command. The **G** command can be terminated prematurely by pressing [DEL] or [BREAK]; otherwise it terminates normally when all lines in the file have been scanned for a string matching *pattern*.

Here is an example:

```
G/editor/
```

```
The ed text editor is easy to use and easy to learn.
```

```
s/easy/simple/
```

```
editor. This is evident in the lack of prompts and the
```

```
The ed text editor operates in two modes: command mode and
```

```
s/The ed text editor/ed/
```

```
text entry mode. In command mode, the editor interprets
```

```
s/the editor/ed/
```

```
editor. This is evident in the lack of prompts and the
```

```
The ed text editor is easy to learn and easy to use.
```

```
s/easy to use/simple to use/
```

```
The ed editor is great? The ed editor is great!
```

```
s/[^?]*? //
```

In this example, **G** looks for all the lines containing "editor", and executes the commands you specify. Note that a new-line was typed on each of the two blank lines above, causing no command to be executed.

The **&** character can be typed in place of a command. This causes the most recent command executed within the current invocation of **G** to be re-executed.

The **V** command is identical to the **G** command, except that the lines that are marked and printed are those that do *not* contain a string that matches *pattern*.

The **u** command can be used to reverse all the effects of a **G** command.

Joining Lines Together

The **j** (join) command joins two or more lines together. By default, **j** appends line `dot+1` to line `dot`, but you can specify a range of lines to be joined. Note that **j** does not add any white space between the joined lines. `Dot` is left pointing to the line created after the specified lines have been joined.

As an example, try joining the last two lines of the file together. First, however, you need to shorten line `$-1` so the joined line fits on one line of the screen. Do this by typing

```
$-1s/easy to learn and //p  
The ed text editor is simple to use.
```

Now join the last two lines together with

```
jp  
The ed text editor is simple to use. The ed editor is great!  
s/\.T/. T/p  
The ed text editor is simple to use. The ed editor is great!
```

The last **s** command in this example is used to insert two spaces between the two joined lines. Note that the **p** command can be appended to the **j** command to verify that the two lines have been joined.

Splitting Lines Apart

The **s** command can be used to split a single line into two separate lines. This is done by inserting a new-line between the characters where the split is desired. To do this, the new-line must be preceded by `\` to avoid terminating the **s** command prematurely. Thus, you can split the two lines that were joined in the previous example into two separate lines with the **s** command (you cannot use the **u** command to split the last line into two lines now – why?). Do this by typing the following:

```
s/\. T/.T/p  
The ed text editor is simple to use. The ed editor is great!  
s/\.T/.\  
T/  
$-1,$p
```

The *ed* text editor is simple to use.
The *ed* editor is great!

The first **s** command gets rid of the extra white space in the sentence (note that the **u** command could have been used here). The second **s** command inserts a new-line between the period and the capital **T**, thus creating two separate lines. Note that, although the second **s** command takes up two lines, it is actually one command.

Special Ed Commands

Material Covered:

f	command; set/print currently remembered file name;
;	delimiter; set dot's value;
w	command; writer characters in buffer to file, or read standard output from a shell command;
r	command; read contents of file into buffer, or read standard output from shell command;
e, E	commands; begin editing another file, or read standard output from shell command;
-	option; silences character counts generated by w , r , e , E , or an invocation of <i>ed</i> ;
X	command; initiates text encryption mode;
-x	option; initiates text encryption mode.

Finding the Currently Remembered File Name

If you invoke *ed* with a file name argument, *ed* remembers that file name until your editing session is over, or until the file name is changed as a result of commands that are discussed later in this section. The **f** (file name) command enables you to find out at any time what file name *ed* is remembering. For example,

```
f  
testfile
```

which tells you that *ed* is remembering **testfile** as the current file name.

The **f** command also enables you to change the current file name. For example, to change the current file name to **file2**, type

```
f file2  
file2
```

Ed echoes "file2" so you can verify that the current file is set correctly. Now change the file name back to the current file, or errors could result in later operations:

```
f testfile  
testfile
```

If no file name is specified when *ed* is invoked, then *ed* initially remembers no current file name. Thus, this file name must be supplied when using the **w**, **r**, **e**, or **E** commands (discussed later), or it can be set with the **f** command.

Writing Buffer Text Onto a File

The **w** (write) command writes the text contained in the *ed* buffer onto the specified file, or onto the currently remembered file if no file name is specified. If the write is successful, a count of the number of characters written is printed. Dot is left unchanged.

The **w** command accepts zero, one, or two line number arguments specifying the line or lines to be written. If no line number arguments are given, "1,\$" is assumed.

Try the **w** command by typing

```
w
986
```

The previous contents of **testfile** have now been overwritten by the contents of the *ed* buffer. The number 986 tells you that the write was successful, and that 986 characters were written.

Note that the *ed* buffer is not affected by the **w** command. Its contents are still the same. In fact, all of the line pointers (dot, **\$**, and any that you have set) are still pointing to the same lines as they were prior to the **w** command. Thus, you may write out the contents of the *ed* buffer several times during an edit session without disturbing the current state of the editor. It is a good idea to write often, especially if you have been editing a long time and have made many changes. Depending on how often you write, you can be sure that a current version of your file resides in the relative safety of the file system, should a system crash or a power failure eat up whatever data is in the *ed* buffer.

You can tell *ed* to write to a file other than the currently remembered file by typing

```
/^ed;/^on/w file1
561
```

This command writes the range of lines from the line beginning with "ed" to the line beginning with "on" onto the file **file1**. If **file1** exists, its previous contents are completely overwritten by the specified lines of text. If **file1** does not exist, it is created with a file mode of 666 (modified by the current value of the file creation mask, *umask*) and the specified text is written on it. Again, the number returned indicates that *ed* was successful in writing 561 characters on the file.

The semicolon that appears in the last example is new. If a comma had been used to separate the two searches, *ed* would have started the search for a line beginning with "ed" from the current line. After finding that line, however, *ed* would return to the current line to search for the line beginning with "on". The value of dot would be reset only after finding the line beginning with "on", with the result that a single line address is passed to the **w** command, causing a single line to be written. The semicolon causes the value of dot to be set to the line beginning with "ed", so that the second search is carried out with respect to this line, instead of the previous current line. Thus, two addresses are processed, and the correct lines are written. The semicolon can always be used in place of a comma to force dot to be set at that point in the construct.

You can also run shell commands with the **w** command. The shell command is introduced with **!**. For example,

```
w !ls
file1
testfile
986
```

runs *ls* and also writes the current contents of the buffer to the currently remembered file. Note that the output from *ls* appears on your screen, but is not added to the actual contents of the buffer (the listing that appears on your screen may be longer than that shown above). After the listing is produced, *ed* writes the contents of your buffer to the currently remembered file, and reports the number of characters written. Note that there is no way to run a shell command and write to a file other than the currently remembered file with the **w** command. Note also that **!** is illegal if the editor was invoked from a restricted shell (see *rsh(1)* in the HP-UX Reference manual).

The currently remembered file name is set to the file name you specify with the **w** command, if the specified file name is the first file name mentioned since *ed* was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced with **!** is never remembered as the current file name.

Reading Files Into the Buffer

The **r** (read) command reads the contents of a specified file, or the currently remembered file, if no file is specified, into the *ed* buffer after the specified line. If no line is specified, the contents are read in after line **\$**. Dot is set to the last line read in.

To illustrate the **r** command, first create a new file called **readfile**:

```
w
986
e readfile
?readfile
a
Here is some text that is to be read in.
It is used to illustrate the r command.
.
w
81
```

You now have a file in your working directory called **readfile**, containing the text shown above. Now begin editing **testfile** again, and read in the contents of **readfile**:

```
e testfile
986
0r readfile
81
1,5p
Here is some text that is to be read in.
It is used to illustrate the r command.
The ed text editor is simple to use and easy to learn.
```

It was designed to enable the user to get his work done quickly with the least possible amount of interference from the

This example reads the contents of **readfile** into **testfile** after line 0, or at the beginning of the file. *Ed* responds by printing the number of characters that were read in. The first five lines of the buffer are printed to verify that the text is placed correctly.

You can also run shell commands with the **r** command. The shell command is introduced with **!**. For example,

```
/curt/r !date
29
6,9p
editor. This is evident in the lack of prompts and the
curt error messages.
Thu Jul 22 10:59:13 MDT 1982
ed operates in two modes: command mode and
```

which reads the output from *date* into **testfile** after the line containing the pattern "curt". The lines surrounding the insertion are printed to verify that the read executed correctly. Note that, unlike the **w** command, the output from the command becomes part of the text in the buffer. Also, the number of characters read from the command is printed on your screen, but the actual output appears only in the buffer. Note that the **!** is illegal if the editor was invoked from a restricted shell.

The currently remembered file name is reset to the file name you specify with the **r** command, if the specified file name is the first file name mentioned since *ed* was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced by **!** is never remembered as the current file name.

An **r** command can be reversed with the **u** command. Try this now:

```
u
6,8p
editor. This is evident in the lack of prompts and the
curt error messages.
ed operates in two modes: command mode and
```

Note that the date and time are no longer present in the buffer.

Editing Other Files

The **e** (edit) command discards the entire contents of the *ed* buffer and reads in the specified file. If no file is specified, then the currently remembered file is read. Dot is set to the last line of the buffer.

If you have made any changes to the buffer since the last **w** command, *ed* requires that you precede the **e** command with a **w** command to save the contents of the buffer. If you are sure that you want to discard the contents of the buffer, you can invoke the **e** command a second time. This forces *ed* to discard the buffer contents and read in the new file. For example,

```
e file1
?
e file1
561
```

The question mark after the first invocation of **e** is to warn you that you have made changes to the current contents of the buffer, and that these changes will be lost if you do not write them on **testfile**. The second invocation of **e** tells *ed* "I don't care! Do it anyway!". *Ed* complies by discarding the current buffer and reading in the contents of **file1**. *Ed* reports to you the number of characters read.

If you are sure that you want to discard the current contents of the buffer without saving them, you can use the **E** (Edit) command. **E** is similar to **e**, except that *ed* does not check to see if any changes have been made to the current buffer. Thus, you do not have to type the **e** command twice.

If you have made several changes to the buffer, and then decide that you do not like what you have done, you can start editing the same file all over again by typing **e** or **E** with no specified file name. This causes the contents of the currently remembered file to be read into the buffer, destroying the previous contents. Of course, if you have written some of the changes you have made to the current file already, there is no quick and easy way to reverse them.

If you specify a file name with the **e** or **E** command, that file name becomes the new current file, and is remembered for future use with **w**, **r**, **e**, or **E**.

You can also execute shell commands with the **e** or **E** command. The shell command is introduced with **!**. For example,

```
E !ls
23
,p
file1
readfile
testfile
```

This example runs the shell command *ls*, and places its output in the *ed* buffer, destroying whatever was in the buffer previously. The number of characters placed in the buffer is printed for you. The actual list of files and the number of characters read into the buffer may be different than those shown above. Note that **!** is illegal if the editor was invoked from a restricted shell. A shell command is never remembered as the current file name.

Silencing the Character Counts

If the character counts that *ed* produces (when *ed* is invoked, or with the **w**, **r**, **e**, or **E** commands) are annoying or are not helpful, they can be silenced with the **-** option. It is specified when *ed* is invoked, as in

```
$ ed - filename
```

The `-` option also suppresses the question mark generated by the `e` and `q` commands whenever they are not preceded by a `w` command (the `q` command is discussed in the next section).

Encrypting and Decrypting Text

Ed provides a feature that enables you to encrypt and decrypt the text in a file so that other users are not able to read your files. The text is encrypted and decrypted by means of the DES encryption algorithm (see *crypt(1)* in the HP-UX Reference manual). To encrypt your text, you must supply a *key*, which is simply a string of one or more characters. The key determines the manner in which the DES algorithm encrypts your text. You *must* remember this key.

The **X** (encrypt) command enables you to encrypt the text in the *ed* buffer. The **X** command accepts no arguments, but prompts you to enter a key. The echoing on your screen is disabled while you enter the key, so there is no visible record of it. For example,

```
E file1
561
,p
editor. This is evident in the lack of prompts and the
curt error messages.
The ed text editor operates in two modes: command mode and
text entry mode. In command mode, ed interprets
your input as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is
important to note that ed always makes changes to the
copy of your file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.
X
Enter file encryption key:
w
561
q
$
```

This example edits **file1**, and prints out its contents. After the **X** command is invoked, you are prompted to enter a key. This key can be any string of characters, but whatever it is, *do not forget your key!* When the **w** command is invoked, the text in the buffer is encrypted according to the key you entered and written on **file1**. The **q** command, which is discussed later, exits the editor and leaves you at the shell level. Now execute the *cat* command to try to print out the contents of **file1**:

```
$ cat file1
(garbage)
.
.
.
$
```

You probably got a screenful of garbage. If your bell beeped a couple of times, this is because the text is encrypted into invisible characters as well as visible characters. There is no practical way for another user to tell what is actually contained in your file.

To edit a file containing encrypted text, use the `-x` option when `ed` is invoked:

```
$ ed -x file1
```

```
Enter file encryption key:
```

```
561
```

```
,p
```

```
editor. This is evident in the lack of prompts and the  
curt error messages.
```

```
The ed text editor operates in two modes: command mode and  
text entry mode. In command mode, ed interprets  
your input as a command. In text entry mode, ed adds  
your input to the text located in a special buffer where  
ed keeps a copy of the file you are editing. It is  
important to note that ed always makes changes to the  
copy of your file in the buffer. The contents of the  
original file are not changed until you write the changes  
on top of the original contents of your file.
```

The `-x` option is the same as the `X` command, except that it is used when you invoke `ed`. When prompted for the key, you must enter the same key that you entered when the text was encrypted. Otherwise, the text in that file is inaccessible. This is why it is so important that you remember your key. After the key is entered, the text in `file1` is decrypted and read into the `ed` buffer. You may now edit the text normally.

When you are done editing, if you invoke the `w` command to write your changes to the file, the text is encrypted according to your key. If you want to change your key or disable encryption altogether, you must use the `X` command. When you are prompted for your key, either type in your new key to change the encryption key, or simply type a new-line. If you type a new-line, a null key is entered, and encryption is disabled. Disable encryption now by typing

```
X
```

```
Enter file encryption key: (new-line)
```

```
w
```

```
561
```

The contents of `file1` are now in a readable form.

Note that, when encryption is enabled, all subsequent `e`, `r`, and `w` commands encrypt the text in the `ed` buffer.

As a general rule, text encryption is seldom needed by the typical user except when extreme security is required. The HP-UX file system has its own security system which is sufficient for most security needs. Using text encryption often and/or on several files at once is a dangerous practice, since you must remember your key to successfully edit these files. You should therefore exercise caution when using the text encryption feature.

The Shell Interface

Material Covered:

! command; execute shell command;
q command; exit editor after checking for changes to the buffer;
Q command; exit editor without checking buffer for changes.

Escaping to the Shell Temporarily

The **!** command enables you to execute a shell command from within the *ed* editor. To do this, type a **!**, followed by the shell command. For example,

```
!(date;who) >whofile  
!
```

executes the *date* and *who* commands, and redirects their output into the file **whofile**. Note that *ed* returns a **!** to tell you when the command has completed execution.

If the character **%** appears anywhere in the shell command, it is replaced with the currently remembered file name. Thus,

```
!sort % >sortedfile  
sort file1 >sortedfile  
!
```

sorts (in reverse alphabetical order) the current contents of **file1**. Note that the current contents of **file1**, not the *ed* buffer, are sorted. The sorted version of **file1** is redirected to the file **sortedfile**. The I/O redirection in the last two examples is used so that the output from these shell commands does not clutter up your screen while you are editing. Note that, if the output from a shell command is printed on your screen, the output does not become part of the *ed* buffer unless **!** is used with the **r**, **e**, or **E** commands.

A final feature of the **!** command is the ability to re-execute the last shell command you executed with **!**, without having to re-type the entire command. This is done by typing two exclamation points, as in

```
!!  
!
```

which re-executes the last shell command executed within the *ed* editor. Thus, **sort % >sortedfile** is re-executed.

If a shell command contains any metacharacters, *ed* echoes the command line back to you with all metacharacters expanded (this is what *ed* did in the first *sort* example above). For example,

```
!cat * >bigfile  
cat file1 readfile sortedfile testfile whofile >bigfile  
!
```

which echoes the expanded command line, then executes the command.

Exiting the Editor

The **q** (quit) command exits the editor. The contents of the buffer are not automatically written on the current file. If you have made any changes to the buffer since the last time you invoked the **w** command, *ed* requires that you issue the **w** command before exiting with **q**. Invoking the **q** command a second time forces *ed* to let you exit without writing the contents of the buffer on the current file. To illustrate this command, first add some text to the buffer, then try to exit without writing:

```
$a
Here is some extra text.
.
q
?
q
$
```

A change is made to the buffer by adding a single line of text to the end of the buffer. When the first **q** command is typed, *ed* sees that there have been changes to the buffer since the last write, so *ed* issues a question mark. This warns you that there are changes to the text in the buffer that will not be saved if you exit without writing. The second **q** command forces *ed* to discard the contents of the buffer and exit. Be very sure that this is what you want to do, since you cannot recover the buffer contents once you have exited. The **\$** is the default shell prompt, indicating that you are once more at the shell level (your shell prompt may be different).

If you know that you want to discard the contents of the buffer and exit, but you do not want to type the **q** command twice, use the **Q** command. The **Q** command is similar to **q**, but *ed* does not check to see if changes have been made to the contents of the buffer.

The **-** option previously discussed disables the question mark that *ed* issues when you do not write before executing an **e** or **q** command. You are living dangerously when it is disabled, however. That question mark has kept many users from accidentally throwing away hours of work. Besides, the **E** and **Q** commands are implemented for those special cases when you want to discard the contents of the buffer.

Miscellaneous Topics

Material Covered:

[DEL],[RUB],[BREAK] keys; any of these keys generates an interrupt signal to *ed*;

Editing Scripts

Interrupting the Editor

[DEL], [RUB], or [BREAK] causes *ed* to stop whatever command it is executing and return to you for a command. *Ed* tries to restore the state of your file to whatever it was before the command was issued. This is easily done if *ed* is interrupted while printing, since dot is not set until printing is done. If *ed* is reading or writing files, or performing substitutions or deletions, however, the state of the buffer (and the current file) is unpredictable; dot may or may not be changed. Thus, it is usually safer to let *ed* finish whatever it is doing, rather than risk finding the buffer or the current file in some garbled state.

Editing Scripts

An editing script is simply a file containing a list of *ed* commands. If you have several files on which a specific list of commands must be executed, it is easier to use an editing script than it is to invoke *ed* once for every file, and perform the tasks in each.

Suppose you have several files named **file1**, **file2**, ..., and you want to perform some specific substitutions, additions, and deletions on each. First, create a file (called **script**, for example), and put all the *ed* commands that you want to execute, in the order that they must be executed, in the file:

```
$ ed script
?script
a
Or !date
1s.*$& DATE OF LAST UPDATE/
$-3,$d
g/Karl Harrison/s//Georgia Mitchell/
w
q
.
w
87
q
$
```

The file **script** now contains *ed* commands to put the current date and time at the beginning of each file, append "DATE OF LAST UPDATE" to the date and time, delete the last four lines of each file, and replace every instance of "Karl Harrison" in each file with "Georgia Mitchell". Note that the **w** and **q** commands are included so that the script writes the buffer on each file and exits the editor automatically.

To use **script**, invoke *ed* as follows:

```
$ ed - file1 <script  
$ ed - file2 <script  
etc.
```

The I/O redirection character < causes *ed*, when invoked, to take its input from **script**. Thus, as *ed* is invoked with each file name, that file is edited according to the commands contained in **script**.

Table of Contents

Sed - A Non-interactive Text Editor

Abstract.....	1
Introduction.....	2
Overall Operation.....	2
Command-line Flags.....	3
Order of Application of Editing Commands.....	3
Pattern-space.....	3
Examples.....	3
ADDRESSES: Selecting Lines for Editing.....	3
Line-number Addresses.....	4
Context Addresses.....	4
Number of Addresses.....	4
Examples.....	5
FUNCTIONS.....	5
Whole-line Oriented Functions.....	5
Example.....	6
Substitute Function.....	6
Examples.....	8
Input-output Functions.....	8
Examples.....	9
Multiple Input-line Functions.....	9
Hold and Get Functions.....	10
Example.....	10
Flow-of-Control Functions.....	10
Miscellaneous Functions.....	11
Reference.....	11

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

- 1) To edit files too large for comfortable interactive editing;
- 2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed*.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

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 any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of 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. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to 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 frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. 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*, and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall 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; see Section 1.1 below.

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 Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

- n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- e: tells *sed* to take the next argument as an editing command;
- f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands 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* (see Section 3). 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.

1.3. 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 (Section 3.6.).

1.4. Examples

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.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

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. 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 with curly braces ('{ }')(Sec. 3.6.).

2.1. 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 which 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.

2.2. 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 '^', 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.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

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. Then an attempt is made on subsequent lines to again match the first address,

and the process is repeated.

Two addresses are separated by a comma.

Examples:

/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 1,3, 4 (number = zero!)
^(an\).*\1/	matches line 1

3. 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. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a 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 following the *n* command.

(1)a\

<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

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 lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *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 corpse.

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. (The *r* function is described in Section 3.4.)

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.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n          n
i\         c\
XXXX     XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The `<pattern>` argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). 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 new-line.

By default, only the first string matched by `<pattern>` is replaced, but 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 a file 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 file names may be mentioned after `w` flags and `w` functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla 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 (assuming nocopy mode):

```
In XANadu did Kubhla Khán
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

3.3. 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 the file named by <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 the *w* and <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 a file

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 *r* 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 the *r* and <filename>. If a file mentioned by a *r* function 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.)

Examples

Assume that the file 'notel' 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 notel
```

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.

3.4. 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* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. 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 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 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.

Example

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
```

3.6. 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 *Don't* 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){ -- Grouping

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> -- place a 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:

- 1) reading a new input line, or
- 2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* 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.

Reference

- [1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories, 1978.

Table of Contents

Awk - A Pattern Scanning And Processing Language

Abstract	1
Introduction	2
Usage	2
Program Structure	2
Records and Fields	2
Printing	3
Patterns	3
BEGIN and END	3
Regular Expressions	4
Relational Expressions	4
Pattern Ranges	4
Actions	4
Built-in Functions	5
Variables, Expressions, and Assignments	5
Field Variables	5
String Concatenation	6
Arrays	6
Flow-of-Control Statements	6
Design	6
Implementation	7
References	8
Examples	9

Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *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**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

September 1, 1978

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each 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 the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table 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.

1.1. Usage

The command

```
awk program [files]
```

executes the *awk* commands in the string *program* on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file *pfile*, and executed by the command

```
awk -f pfile [files]
```

1.2. 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 the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied 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.

1.3. Records and Fields

Awk input is divided into “records” terminated by a record separator. The default record separator is a 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 — but the input field separator may be changed, as described below. Fields are referred to as *\$1*, *\$2*, and so forth, where *\$1* is the first field, and *\$0* is the whole input record itself. Fields may

†UNIX is a Trademark of Bell Laboratories.

be assigned to. 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; they may be changed at any time to any single character. The optional command-line argument **-Fc** may also be used to set **FS** to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, **\$1**, on the file **foo1**, and the second field on file **foo2**. The **>>** notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to **bwk**.

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. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C.²

2. Patterns

A pattern in front 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 these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus 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 = ":" }  
... rest of program ...
```

Or the input lines may be counted by

```
END { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name “smith”. If a line contains “smith” as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*¹ and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for “one or more”, and ? for “zero or one”, all as in *lex*. Character classes may be abbreviated: [a-zA-Z0-9] is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names “Aho,” “Weinberger” or “Kernighan,” whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
\/.*\//
```

which 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.” Notice 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.

2.3. 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 it is numeric. 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.

2.4. 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 will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. 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 between an occurrence of **pat1** and the next 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.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a “length” function to compute the length of a string of characters. This program 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, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **\$1** and **\$2**.

3.2. 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 7 to **x**. Strings which cannot be inter-

preted 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 **+**, **-**, *****, **/**, and **%** (mod). The C increment **++** and decrement **--** operators are also available, and so are the assignment operators **+=**, **-=**, ***=**, **/=**, and **%=**. These operators may all be used in expressions.

3.3. 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. Thus one can 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 “too big” when it is, and in any case 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 the string **s** into **array[1]**, ..., **array[n]**. The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. 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.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. 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.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements *if-else*, *while*, *for*, and statement grouping with braces, as in C. We showed the *if* statement in section 3.3 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
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*¹ provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides 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 is an attempt to fill in another part of the matrix of possibilities. It 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 or a 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 the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* 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. We have tried to make the syntax powerful but 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.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;⁴ 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.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number :".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls -l*; each line has the form

```
-rw-rw-rw- 1 ava 123 Oct 15 17:05 xxx
```

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories (May 1975). Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

Program	1	2	3	Task 4	5	6	7	8
<i>wc</i>	8.6							
<i>grep</i>	11.7	13.1						
<i>egrep</i>	6.2	11.5	11.6					
<i>fgrep</i>	7.7	13.8	16.1					
<i>sed</i>	10.2	11.6	15.8	29.0	30.5	16.1		
<i>lex</i>	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
<i>awk</i>	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table 1. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. END {print NR}
2. /doug/
3. /ken\doug\dmr/
4. {print \$3}
5. {print \$3, \$2}
6. /ken/ {print >"jken"}
/doug/ {print >"jdoug"}
/dmr/ {print >"jdmr"}
7. {print NR ": " \$0}
8. {sum = sum + \$4}
END {print sum}

SED:

1. \$=
2. /doug/p
3. /doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
4. /[]* []*[]* []*\([]*\) .*/s/\1/p
5. /[]* []*\([]*\) []*\([]*\) .*/s/\2 \1/p
6. /ken/w jken
/doug/w jdoug
/dmr/w jdmr

LEX:

1. %{
int i;
%}
%%
\n i++;
. ;
%%
yywrap() {
printf("%d\n", i);
}
2. %%
^.*doug.*\$ printf("%s\n", yytext);
. ;
\n ;

Table of Contents

Shell Programming

Terminology	1
Creating a Simple Shell Script	1
Example	2
Creating the Script	2
Running the Script	3
Parameters	4
Types of Parameters	4
Common Parameters	4
Positional Parameters	4
Special Parameters	5
How to Use Parameters	5
Parameter Substitution	7
Quoting	9
The Backslash	9
The Single Quote	9
The Double Quote	11
The Grave Accent	12
Using Parameters in Shell Scripts	12
Examples	12
Example 1 - The Compile Shell Script	12
Example 2 - The Modfile Shell Script	13
Example 3 - Comments and Here Documents	14
Command Separators	15
The Semicolon	15
The Ampersand	15
Mixing ; and & Separators	16
The Double Ampersand	16
The Double Vertical Bar	16
Mixing && and Separators	17
Mixing ;, &, &&, and Separators	17
Command Grouping	18
Grouping With Parentheses	18
Grouping With Braces	20
Control-Flow Constructs	20
The FOR Construct	20
Examples	21
The CASE Construct	22
Examples	22
The IF Construct	23
Examples	23
The WHILE Construct	24
Example	24
The UNTIL Construct	25
Example	25
An Example Shell Program	26
For More Information	28

Shell Programming

The shell is perhaps one of the most versatile programs in your HP-UX system. The shell has two main roles. Its most common role is that of a *command interpreter* – reading input from your terminal, and interpreting it as a request for a particular program to be executed. The shell as a command interpreter is discussed in the supplied tutorial text by Jean Yates.

The second role of the shell is that of a *programming language*. The shell programming language is a structured programming language that is directly recognized and executed by the shell; it requires no compilation. It includes structured programming constructs like **if**, **case**, **for**, **while**, and **until**. In addition, all HP-UX commands, as well as any commands you may have written yourself, can be executed within a shell program. Other capabilities, such as parameters, parameter substitution, command substitution, and comments, are also supported.

This article describes the shell programming language in detail, including tutorial examples as appropriate.

Terminology

The terms *shell script* and *shell program* are often used interchangeably to refer to a program written in the shell programming language. In this article, however, these two terms are used differently. For the remainder of this article, *shell script* refers to a list of commands that is executed once, in the order the commands are listed. A shell script contains no conditional testing, no looping, no branching, and no structured programming constructs; it may or may not contain one or more parameters. A *shell program* refers to those procedures that do not qualify as shell scripts. The term *procedure* refers to both shell scripts and shell programs. It is used when describing concepts that are common to both.

Creating a Simple Shell Script

A *simple shell script* is one in which no parameters occur. A simple shell script is useful when the following two conditions are true:

- you anticipate having to perform one or more tasks several times over a period of time, and

- the tasks you must perform, the commands needed to perform those tasks, and the arguments to the commands *never change*.

A simple shell script is not well adapted to change, because you are required to edit the script if changes are necessary. However, a simple shell script can save you a lot of time and typing, as well as relieve you of certain mundane chores, if you have a well-defined set of objectives for your script.

Example

Suppose you want to monitor the user activity on a multi-user system. After considering the problem, you decide that your script should give you the following information:

1. the current date and time;
2. the names of the users currently logged in;
3. a list of all the processes on the system;
4. the number of disc blocks being used by each user.

You can see that each numbered item is satisfied by a single command:

1. **date**
2. **who**
3. **ps -e**
4. **du /users**

To make the output more readable, you decide to include the *echo* command, also. You now have a complete list of all the commands you need to perform the tasks at hand. All that remains is to put them all together in your script.

Creating the Script

Any procedure is simply an ASCII text file. Thus, you may use the editor of your choice to create your script. The editor *ed* is used in this example.

You decide that **status** is a good name for the script, since it gives information about the status of the system at a particular date and time. In the following editing session, your input is shown in **bold**:

```
$ ed status
?status
a
echo "Current date and time: `date`\n"
echo "\nUsers logged in:\n"
who
echo "\nCurrent processes:\n"
ps -e
echo "\nUser disc usage:\n"
du /users
echo "\n*****\n"
.
w
204
q
$
```

You now have a file called **status** containing the commands you want to run.

Running the Script

You can execute **status** by typing

```
$ sh status
```

which creates a new shell to run the script. A more common way, however, is to mark **status** as executable, so that it may be executed just like other commands. To do this, use the *chmod* command, and type

```
$ chmod 755 status
```

which sets the mode of **status** such that everybody may read and execute **status**, but only you can modify it. (Note that you are free to assign whatever mode you want, as long as you give yourself execute permission. You should also give yourself read and write permission, because you cannot read or modify your script without them!) You may now execute **status** by typing

```
$ status
```

which causes the shell to execute the commands contained in **status**. The output appears on your terminal.

You have already saved yourself a lot of typing with **status**, but there is still more you can do. Suppose you want to collect this information in a file called **logfile** for later inspection. You can do this by typing

```
$ status >>logfile
```

The append redirection `>>` is used instead of `>` so that successive invocations of **status** do not overwrite the old contents of **logfile**. In this way, you can keep a history of user activity that is updated as often as **status** is executed.

The only problem with this is that you are required to manually execute **status** every time you want to update **logfile**. However, there is a command, called *cron*, that executes commands on a scheduled basis, according to entries in the file */usr/lib/crontab*. Thus, you can schedule **status** to be run as often as once a minute (note that **crontab** may be protected such that no one except the super-user may edit it – ask your super-user for details). Your only task is to check **logfile** periodically to get the information you need, and remove information that is no longer useful.

Parameters

Procedures almost always make use of at least one parameter. A *parameter* is a string of one or more characters that is made to stand for another string of characters. Parameters are similar in many respects to variables in other programming languages.

Types of Parameters

Parameters can be divided into three types: *common parameters*, *positional parameters*, and *special parameters*.

Common Parameters

A common parameter is a name consisting of a series of letters, digits, and/or underscores. The first character must be a letter or an underscore. Common parameters are completely user-defineable. Values are assigned to common parameters by writing

```
parameter = value
```

where *value* is the value to be assigned to *parameter*. The values of common parameters are always character strings. Some examples of valid assignment statements are:

```
dir = /usr/lib/gates
user = fred
null =
cwdir = `pwd`
```

In these examples, the string `"/usr/lib/gates"` is assigned to **dir**, `"fred"` is assigned to **user**, and the null string is assigned to **null**. In the last example, the name of your current working directory is assigned to **cwdir** using *command substitution*. Whenever a command is enclosed in grave accents (```), the command is replaced by its output when the command is executed. Command substitution is explained in the supplied tutorial text by Jean Yates.

Positional Parameters

Whenever you give the shell a command to execute, the shell automatically sets the values of ten positional parameters named **\$0**, **\$1**, **\$2**, ... **\$9**. Each positional parameter contains the value of one argument specified on a command line. For example, if you type

```
$ cc prog1.c prog2.c prog3.c
```

then the shell sets **\$0** equal to `"cc"`, **\$1** equal to `"prog1.c"`, **\$2** equal to `"prog2.c"`, and **\$3** equal to `"prog3.c"`. **\$4** through **\$9** are set equal to the null string. Thus, **\$0** always contains the name of the invoked command, and **\$1** through **\$9** contain the values of the command's arguments, if any, in the order they are specified. The null string is assigned to all positional parameters not given a value on the command line.

If more than ten arguments (including the command name) are specified on the command line, only the first ten are assigned to positional parameters. The remaining arguments are saved, but are not accessible until the *shift* command is used. The *shift* command shifts the value of **\$2** to **\$1**, **\$3** to **\$2**, **\$4** to **\$3**, etc. Thus, the remaining arguments are shifted into **\$9** one at a time, until eventually all the remaining arguments are addressable through positional parameters.

These parameters are available for use in procedures, and enable you to write procedures that accept arguments. Their values cannot be changed except with the *set* and *shift* commands (documented in *Special Commands* under *sh(1)*, in the HP-UX Reference manual).

Special Parameters

Several parameters are "special", either because they are used elsewhere by the HP-UX system, or because they are automatically set by the shell. Parameters that are automatically set by the shell are:

- # the number of non-null positional parameters in decimal; can be greater than 10;
- the flags supplied to the shell on invocation, or by the *set* command; flags are stored in - as a character string, with each character in the string specifying a shell flag that has been set;
- ? the decimal value returned by the last synchronously executed (i.e. not executed in the background) command;
- \$ the process ID of the shell running the current procedure;
- ! the process number of the last background (asynchronous) command invoked.

The values of these parameters cannot be changed.

Parameters that are used by the system are **HOME**, **PATH**, **TERM**, **SHELL**, **EXINIT**, **MAIL**, **TZ**, **PS1**, **PS2**, and **IFS**. Some of these parameters (**TERM**, **PATH**, and possibly **MAIL**, **SHELL**, and **EXINIT**) are environment parameters, which are used to set up the environment in which your processes run. These parameters should not be changed indiscriminately. For example, **PATH** tells the shell which directories it should search to find the programs you want to execute. If you redefine **PATH** with some unrelated value, the shell no longer knows where to look. Thus, as a safety precaution, all of the above parameters should be avoided for general use in procedures, whether they are part of the environment or not.

How to Use Parameters

Once a parameter has been assigned a value, you may obtain its value by preceding the parameter with a dollar sign (\$). For example,

```
dirname = /users/bill/dir1
cd $dirname
```

This example assigns the string `"/users/bill/dir1 "` to **dirname**, and then uses **dirname** to change the current working directory. The `cd` command, after substituting the value of **dirname**, is equivalent to

```
cd /users/bill/dir1
```

The dollar sign signals the shell that the following characters specify a parameter, and that the parameter's value is to be substituted in its place. If you had omitted the dollar sign in the previous example, the shell would assume that you want to change your current working directory to a directory called "dirname".

Whenever a parameter is preceded by a dollar sign, the parameter is said to be *dereferenced*. All parameters must be dereferenced to obtain their values (positional parameters always appear with a dollar sign, and thus are always dereferenced).

One or more characters can be added to the end of a parameter value by enclosing the parameter name in braces, and explicitly typing the added character(s). For example,

```
dirname = /users/bill/dir
cd ${dirname}1
.
.
.
cd ${dirname}2
.
.
.
```

This example assigns `"/users/bill/dir"` to **dirname**, and then appends a single character to **dirname** in subsequent `cd` commands. The two `cd` commands are equivalent to

```
cd /users/bill/dir1
```

and

```
cd /users/bill/dir2
```

respectively. Note that the appended character does not affect the value of **dirname**. The following example shows how characters can be added to the beginning and the end of a parameter value:

```
fn = prog1
dirname = /users/bill
mv $fn $dirname/bill.${fn}.R2
```

Although somewhat difficult to read, this example shows a commonly used method of building file names from parameters. Using the given values of **fn** and **dirname**, the `mv` command is equivalent to

```
mv prog1 /users/bill/bill.prog1.R2
```

which moves **prog1** from the current working directory to the directory */users/bill*, and renames it **bill.prog1.R2**. Note that the braces are necessary only when a parameter name is *followed* by one or more characters that are not to be interpreted as part of the parameter name. (Note also that the braces are *not* necessary if the parameter name is immediately followed by a slash (/), as shown by **dirname** in the previous example.)

Parameter Substitution

There are constructs that enable you to substitute other values in place of parameter values, depending on whether or not the parameter is set or null. The four constructs are:

`${parameter:-word}`

If *parameter* is set and non-null, then dereference its value. Otherwise, substitute *word*, where *word* can be any valid parameter value.

`${parameter:=word}`

If *parameter* is not set or is null, then set it to *word*. The value of *parameter* is then dereferenced. Positional parameters may not be set in this way.

`${parameter:?word}`

If *parameter* is set and is non-null, then dereference its value. Otherwise, print *word* and exit from the current process. If *word* is omitted, then the message "parameter null or not set" is printed.

`${parameter:+word}`

If *parameter* is set and is non-null, then substitute *word*. Otherwise, substitute a null value.

Here are some examples:

```
echo "The directory being processed is ${1:-`pwd`}."
```

This statement could exist in a procedure that performs a specific task on each file in a particular directory. The first argument to the procedure (**\$1**) is the name of the directory to process. If no directory is specified, the current working directory is used. This echo statement reports which directory is being processed. If **\$1** is set, its value is printed; otherwise, command substitution is used to print the name of the current working directory.

```
pr ${whichfiles:=*} >/dev/lp
```

This statement could be used to print one or more files on the line printer */dev/lp*. The parameter **whichfiles** could be either a single file name, or a pattern of special characters, specifying the file(s) to print. If **whichfiles** is set, its value is dereferenced, and the specified files are printed. If **whichfiles** is not set or is null, it is set equal to an asterisk (*). Its value is then dereferenced, causing all the files in the current working directory to be printed.

```
cd ${arg3:? "Arg3 not set" }
```

This statement could appear in a procedure to inform the user when a necessary argument (**arg3**, in this case) has not been set. In this example, **arg3** is the name of a directory which must be specified. If **arg3** is set, the *cd* command is executed; otherwise, the message "arg3: Arg3 not set" is printed, and the process is terminated. The initial "arg3:" in the message is added by the shell as an additional identifier. (Note that, if your terminal is set to echo all eight bits of an ASCII character, you might get garbage output on your terminal.)

```
{dirname: + `echo cd $dirname`}
```

In this example, if **dirname** is set, then the command "cd \$dirname" is substituted. Otherwise, no action is taken. The *echo* command used inside command substitution marks is necessary for the following reasons:

The shell expects a *single word* of information following the + in this example. Thus, the *cd* command and its argument must be enclosed in double quotes to force the shell to treat it as a single word.

Enclosing "cd \$dirname" in double quotes is not enough, however, because a subtle error is generated. For example, suppose you typed the following lines in a shell program:

```
dirname = /users/bill  
{dirname: + "cd $dirname"}
```

The shell sees that "cd \$dirname" is to be treated as a single word, and looks for a command named "cd /users/bill" (instead of a command named "cd" with an argument of "/users/bill")! Expressed in this way, the shell cannot distinguish two arguments; it only sees one argument and, obviously, an error is generated.

The solution to this is to allow the *echo* command to pass two distinct arguments to the shell, but still make the entire construct look like a single word. This is easily done, because a command substitution construct is always treated as a single word by the shell. Thus, the double quotes surrounding "cd \$dirname" are not necessary, and the construct `echo cd \$dirname` causes this example to execute correctly.

The braces are necessary in all four of the previously described constructs. If the colon is omitted in any of the constructs, the shell simply checks to see if *parameter* is set or not, and no other action is taken.

Quoting

There are four characters used to quote other characters in procedures. These characters are the backslash (\), the single quote ('), the double quote ("), and the grave accent (`).

The Backslash

The special meaning of a character can be stripped away by preceding that character with a backslash. Whenever a character is preceded by a backslash, the character is said to be *quoted*, and it is interpreted literally. For example,

```
echo prog*.c \*list\* lib\?.3?
```

The first argument tells *echo* to print all files in the current directory whose names begin with "prog", followed by any number of characters, followed by ".c". The second argument tells *echo* to print "*list*", since both asterisks are quoted, and are thus interpreted literally. The third argument tells *echo* to print all files in the current directory whose names are "lib?.3" followed by any single character. The first question mark is literal; the second stands for any single character.

The backslash is the most powerful quoting character, in that it can quote all special characters, including itself. It is also the most limited in scope, since it can quote only one character at a time. The following list shows all the characters that are special to the shell, all of which are quotable with a backslash:

```
? * [ ] \ $ ^ " ` | & ; ( ) < > { } new-line
```

Note that, if a new-line is quoted by a backslash, the new-line is ignored completely.

If you are ever in doubt about whether or not a character needs quoting, it is safe to precede the character with a backslash; if the character has no special meaning, the backslash is ignored.

The Single Quote

The single quote quotes all the special characters except the single quote itself. It has the added advantage of enabling you to quote several characters at once. For example,

```
echo `prog*.c *list* lib\?.3?`
```

prints the exact characters listed between the single quotes. Even the backslash is treated literally. This means that a string like

```
echo `Can\`t find file`
```

does not work as expected, because the backslash loses its quoting ability when enclosed between single quotes. Thus, there is no way to put a single quote between single quotes without inadvertently confusing the shell.

The previous example produces a subtle error that deserves more explanation. If you type

```
$ echo `Can\t find file`
```

to the shell, the shell first examines the command line looking for syntax errors. It sees first the string "echo", followed by the quoted string "can\t", followed by the characters "t find file", followed by the beginning of another quoted string. But wait a minute! Where's the rest of the second quoted string? The shell needs more input, so it types

```
$ echo `Can\t find file`  
>
```

back at you. The > is the shell's default *secondary prompt*, which the shell uses to signal that more information is needed. Suppose you then type

```
$ echo `Can\t find file`  
> text`
```

just to complete the command line so *echo* will run. Well, this satisfies the shell's syntax rules, so the shell prepares to execute the following command line:

```
echo Can\t find file(new-line)text
```

(Note that, although the single quotes have disappeared, their effect can be seen in that the backslash in the first quoted string has been interpreted literally.) Where did the new-line come from? It was the first character of the second quoted string! The command now runs "successfully", and you get

```
$ echo `Can\t find file`  
> text`  
Can      find file  
text  
$
```

on your screen. Hold on! Where's the "\t"? And where did all the spaces come from? This time it's not the shell's fault. The *echo* command has a few tricks of its own in the form of *escape sequences*. Escape sequences are character pairs consisting of a backslash and another character (for a complete list of *echo*'s escape sequences, refer to *echo(1)* in the HP-UX Reference manual). Each escape sequence is interpreted to mean something else, and \t causes *echo* to output a tab.

There are two lessons to learn from this. First, do not try to embed a single quote inside a string quoted by single quotes. You will invariably confuse the shell (and yourself, when you try to figure out what went wrong)! Second, beware of escape sequences in arguments to *echo*. It can be very difficult to figure out why literal characters disappear when using *echo* to print them on your screen.

Since \$ and ` are quoted within single quotes, parameter and command substitution cannot be performed. For example,

```
echo `${cmdname}: current working directory is `pwd``
```

still echoes the exact characters shown between the single quotes.

The single quote also forces the shell to interpret several words as a single word (a *word* is a string of one or more characters, delimited by one or more spaces, tabs, and/or new-lines). Thus, many words can be enclosed in single quotes and assigned to a parameter. For example,

```
errmesg = `Cannot find specified file.`
```

assigns the string "Cannot find specified file." to **errmesg**. The space characters embedded in the string no longer delimit words. Instead, the shell treats the entire string as a single word. If **errmesg** is later dereferenced, the string will look exactly as it did when it was assigned to **errmesg**.

The Double Quote

The double quote quotes all special characters except `\`, `$`, `"`, and ```. Since the backslash is not quoted within double quotes, it may be used to quote these four characters. In the following example,

```
echo "The computer responds \"Not found\" and exits."
```

the backslash is used to quote the double quote character. Thus, a double quote may be included in a string enclosed in double quotes. Note that the backslash itself must also be quoted to be interpreted literally within double quotes.

The infamous example given in the last section can be executed with no surprises using double quotes:

```
echo "Can't find file."
```

This time, all characters show up as expected on your screen.

Since `$` and ``` are not quoted, parameter and command substitution are permitted. For example,

```
echo "$dirname processed at `date`."
```

prints out the name of the directory currently being processed, and the date and time at which it was processed. Note that braces are not required around **dirname**, since it is separated from the next word by a space. The backslash can be used to quote `$` and ```, to prevent parameter and command substitution from occurring.

The double quote also enables you to assign several words to a parameter. For example,

```
descr = "print date and time"  
cmd = date  
cmddescr = "$cmd - $descr"
```


This example assigns the short description of *date* to **descr**, using double quotes to force the shell to interpret the four words as a single word. The string "date" is assigned to **cmd**. Finally, the two parameters are dereferenced as shown, and assigned to **cmddescr**. Thus, **cmddescr** now contains a string similar to that found under the NAME heading in the HP-UX Reference manual.

The Grave Accent

The grave accent is used to signal the shell that a command substitution is to be performed. No special characters are quoted within grave accents. Whatever characters you type between grave accents are interpreted exactly as if they were typed after the shell prompt. For example,

```
echo "File contents:\n`cat *UX*[1-5]`"
```

This example outputs the heading "File contents:", followed by a new-line (`\n`). Then, the *cat* command is executed to print out the contents of all files in the current working directory whose names contain the characters "UX", and end with a single digit in the range 1 through 5. Note that, even though the command substitution is enclosed in double quotes, the characters `*`, `[`, and `]` are treated as unquoted in the command substitution.

The backslash may be used to quote characters within a command substitution.

Using Parameters in Shell Scripts

Adding parameters to shell scripts makes them more flexible and adaptable to your changing needs. Positional parameters are especially useful, in that they enable you to pass arguments to your shell scripts.

Examples

Example 1 - The Compile Shell Script

Compile is a short shell script that accepts one argument. It is useful when you have several C programs that you want to compile, one at a time. Each *a.out* file that is produced is renamed such that it has the same name as its corresponding source file, with the ".c" suffix removed. **Compile** contains the following lines:

```
cc $1
fn=`basename $1 .c`
mv a.out $fn
```

The *basename* command strips away all but the last component of a path name, and optionally removes a specified suffix (see *basename(1)* in the HP-UX Reference manual). Thus, if you invoke **compile** by typing

```
$ compile /users/fred/programs/prog1.c
```

the shell sets **\$1** equal to `"/users/fred/programs/prog1.c"`, and the commands in **compile** become

```
cc /users/fred/programs/prog1.c
fn = `basename /users/fred/programs/prog1.c .c`
mv a.out prog1
```

The *basename* command removed `"/users/fred/programs/"` and `".c"` from the string contained in **\$1**, causing **fn** to be set equal to `"prog1"`. Thus, no matter what directory the source file is located in, you always end up with an executable file in your current working directory with a name similar to that of its source file.

Note that you can also invoke **compile** as

```
$ compile prog1.c
```

with the same results. The *basename* command removes parts of a path name only if those parts exist. Thus, the only part that is removed from `"prog1.c"` is `".c"`. **Compile** can therefore be used to compile C programs no matter where the source files reside.

Example 2 - The Modfile Shell Script

The **modfile** shell script copies a file from one directory into another, edits it according to a script of *ed* commands, and records the fact that the file has been copied in a bookkeeping file. It accepts three arguments: the source directory, the destination directory, and the name of the file to be copied, respectively. **Modfile** contains the following lines:

```
log = /users/kb/logfile
edsc = /users/kb/tools/edscript
cp $1/$3 $2
ed - $2/$3 <$edsc
echo "$3 copied and edited." >>$log
```

Absolute path names are used for **log** and **edsc** so that **modfile** does not depend on your current working directory. Thus, you can type

```
$ modfile /users/fred . file1
```

which copies **file1** from `/users/fred` into your current working directory, or

```
$ modfile ../Cprogs /users/bill prog4.c
```

which copies **prog4.c** from the directory **Cprogs** in your parent directory into `/users/bill`. Thus, **modfile** enables you to copy a file from any directory into any other directory, no matter what your current working directory is, provided the modes of the specified directories permit you to copy files.

The `-` option silences *ed* so that no character counts appear on your screen. The file **edscript** contains a list of one or more *ed* commands that *ed* is to apply to each file that is copied. Finally, the output from the *echo* command is redirected to the file `/users/kb/logfile`, so that a record of the action taken is saved as a convenient reminder.

Note that **modfile** is written so that changes can be made without having to actually change the code. The directory names and the file to be copied are all parameters, and the file **edscript** can be edited to change the way *ed* modifies the copied file.

Example 3 - Comments and Here Documents

A *comment* is introduced by a pound sign (#), and continues until the next new-line. All characters between the pound sign and the new-line are ignored by the shell. Comments can be added to the **modfile** shell script as follows:

```
# Initialize parameters
#
log = /users/kb/logfile
edsc = /users/kb/tools/edscript
#
# Copy file
#
cp $1/$3 $2
#
# Modify copied file
#
ed - $2/$1 <$edsc
#
# Write record
#
echo "$1 copied and edited." >>$log
```

It is good programming practice to provide comments in your shell scripts where necessary. They not only help others understand your scripts, but they can also help refresh your memory if you revisit a script that you have set aside for awhile.

A *here document* is a type of I/O redirection that enables you to include input to a certain program inside the shell script itself. A here document has the following form:

```
command [ args ... ] <<[ - ] word
.
.
.
word
```

The << redirection tells the shell that the input for *command* is to be taken from the following here document. *Word* consists of one or more characters, and signals the beginning and the end of the here document. All lines between the beginning and ending *word* are given to *command* as its standard input. If any character of *word* is quoted, then all characters within the here document are quoted. Otherwise, parameter and command substitution take place, the characters \, \$, and ` are special, and the first character of *word* must be quoted if it is used within the here document. If - is appended to <<, then all leading tabs are removed from the here document and from *word*.

The *ed* command in **modfile**

```
ed - $2/$3 <$edsc
```

can be rewritten to use a here document, as shown:

```
ed - $2/$3 <<-\%  
    /for/c  
    while(i != limit) {  
    .  
    g/exit/d  
%  
%
```

In this example, % defines the beginning and end of a here document containing four lines of input for *ed*. % is preceded by a -, to strip away all leading tabs in the document, and by \, to ensure that all characters in the document are quoted. The here document is indented for clarity.

Using a here document in **modfile** requires that you edit **modfile** if you want to change the way *ed* modifies the copied file. However, by including the here document, you can eliminate the file **edscript**. Also, it is more convenient to debug a procedure if a command's input is readily accessible. Here documents become more valuable as the size and complexity of the procedure grows.

Command Separators

The characters ;, &, &&, and || can be used to separate one or more commands or pipelines, causing sequential, asynchronous (background), or conditional execution of the commands or pipelines.

The Semicolon

The semicolon (;) causes sequential execution of each command or pipeline specified. It is equivalent to a new-line. For example,

```
cd /users/kb; mv ../fred/file1 .; ls
```

causes the shell to execute the *cd* command, then the *mv* command, and finally the *ls* command. *Sequential execution* means that the shell waits for each command to finish before executing the next one. Thus, only one additional process exists at any given time. Note that sequential execution is the normal mode of execution for the shell, so a semicolon is not needed after the *ls* command to ensure that it executes sequentially.

The Ampersand

The ampersand (&) causes asynchronous execution of each command or pipeline specified. For example,

```
cc prog1.c prog2.c & sort -d -o file1 file1 & wc textfile &
```

causes the shell to create a new process for each command listed. *Asynchronous execution* (sometimes referred to as executing a command in the background) means that the shell does not wait for termination of the first command before executing the next. Thus, depending on how long each command executes, three processes can exist at the same time in the previous example. The first process is compiling **prog1.c** and **prog2.c**, the second is sorting **file1**, and the third is counting the number of lines, words, and characters in **textfile**. Note that the ampersand following the *wc* command must be specified, or the *wc* command is executed sequentially.

The shell reports the process numbers of each process created by a *&* separator. Thus, if you execute the above example, three numbers are printed on your screen which identify the three processes created. These are provided for your convenience, should you decide to terminate these processes prematurely with the *kill* command.

Mixing ; and & Separators

Sequential execution works well with commands that have short execution times, and asynchronous execution works well with commands that have long execution times. It is helpful to be able to choose the type of execution based on the execution time of a command. The semicolon and ampersand separators can be intermixed on a line, so you can avoid having to wait for lengthy commands. For example,

```
cc prog1.c prog2.c prog3.c & cd /users/bill; ls -l
```

Here, the shell creates a new process and executes *cc* in that process. Then, without waiting for *cc* to finish, the shell sequentially executes *cd* and *ls*, waiting for the *cd* command to finish before executing the *ls* command.

Note that a syntax error is generated if a semicolon and an ampersand appear adjacent to each other.

The Double Ampersand

The double ampersand (&&) causes the next command or pipeline in the sequence to be executed only if the previous command or pipeline executes successfully. For example,

```
test -d /users/kb/tools && cd /users/kb/tools
```

first checks to make sure that */users/kb/tools* exists. If so, the current working directory is changed to */users/kb/tools*. If not, no further action is taken.

The Double Vertical Bar

The double vertical bar (||) causes the next command or pipeline in the sequence to be executed only if the previous command or pipeline was unsuccessful. For example,

```
test -d /users/kb/tools || mkdir /users/kb/tools
```

first checks to see if the directory */usr/kb/tools* exists. If so, no further action is taken. If not, the directory is created using *mkdir*.

Mixing && and || Separators

The && and || separators can also be intermixed on a line. For example,

```
test -d /usr/tmp && rm /usr/tmp/* || echo "Permission denied"
```

which first checks to see if the directory */usr/tmp* exists. If so, all files in */usr/tmp* are removed. If *rm* fails, the message "Permission denied" is printed. If */usr/tmp* does not exist, no further action is taken.

Mixing ;, &, &&, and || Separators

All four command separators can be intermixed on a line, but the interpretation of the actual execution sequence becomes more complex. For example,

```
test -d /tools && cd /tools; test -z "$fn" || sort -o $fn $fn &
```

The shell uses ; and & to terminate a command sequence. Thus, this example contains two command sequences. The first command sequence is

```
test -d /tools && cd /tools;
```

which first checks to make sure that the directory */tools* exists. If it does, it becomes the current working directory; if not, no further action is taken. Since sequential execution is required by the semicolon, the shell executes this command sequence first, waiting until the *test* and *cd* commands have finished before executing the second command sequence. The second command sequence is

```
test -z "$fn" || sort -o $fn $fn &
```

which first checks to see if the value of *fn* has a zero length. If not, the contents of the file specified by *fn* is sorted; otherwise, no further action is taken. Note that the terminating & places this entire command sequence in the background, not just the *sort* command.

All four command separators are rarely combined in a single command line as shown above. Other constructs in the shell programming language provide the same functions in a much more readable format. Also, the time required to design and debug lengthy sequences of commands is prohibitive. If space is a consideration, however, there is no more compact way of expressing a particular command sequence.

Command Grouping

The left and right parentheses () and the left and right braces { } can be used to force several commands to be grouped together in a single unit.

Grouping With Parentheses

All commands enclosed in parentheses are passed to a new shell process to be executed. For purposes of discussion, *new process* refers to the process that is created to execute the parenthesized commands, and *calling process* refers to the process that reads the parenthesized commands, creates the new process, and passes the commands to the new process. Both processes have their own shell. For instance, in this example,

```
(who;ls)
```

the calling process creates a new process to which the *who* and *ls* commands are passed. The new process executes *who* and *ls* sequentially. The calling process waits for the new process to signal that the commands have been executed. When the signal is received, the new process dies, and the calling process reads the next command.

The & separator can be used to cause asynchronous execution in one or both of the processes. For example,

```
(cd $HOME; ed - newfile <script; rm script) &
```

The & in this example is seen only by the calling process. The new process sees

```
cd $HOME; ed - newfile <script; rm script
```

and executes each command sequentially. The calling process treats the entire parenthesized sequence as a command that is to be run asynchronously. Thus, the process number of the new process is reported, and the calling process proceeds to the next command, without waiting for the new process to signal that the job is completed. (Note that the *cd* command affects only the current working directory in the new process; the calling process's current working directory is unchanged.)

If the sequence is typed as follows,

```
(cd $HOME; ed - newfile <script &)
```

then only the new process is aware of the & separator. The calling process simply sees a command that is to be run sequentially, and waits for a signal from the new process before continuing. The new process sees

```
cd $HOME; ed - newfile <script &
```

Thus, *cd* is executed sequentially, and a separate process is created to execute *ed*. The new process reports the process number of the process that is executing *ed*, signals the calling process that the job is completed, and dies, even though *ed* is still executing asynchronously. The calling process then reads the next command.

Parentheses can be nested, with the result that more than one new process is created. For example,

```
test -f $fn && (ed - $fn <ed1 && (rm ed1; sort -o $fn $fn &)) &
```

The calling process sees

```
test -f $fn && ( ... ) &
```

which tells it to execute the sequence asynchronously. Thus, the calling process creates another process (process A) to execute the sequence, reports the process number of that process, and reads the next command. As far as the calling process is concerned, the specified command sequence has been executed.

Meanwhile, process A sees

```
test -f $fn && ( ... )
```

which is everything that the calling process saw, except that the final `&` is missing. Thus, process A begins sequential execution of its command sequence. It first executes the `test` command, and, if unsuccessful, signals the calling process and dies. Nothing more is done. If successful, however, process A creates a new process (process B) to execute everything within the first level of parentheses. Because there is no `&` separator, process A waits for a signal from process B that the job is done. When process B's signal is received, process A in turn signals the calling process that the job is done. Note that process A's signal is ignored by the calling process, regardless of whether or not process B is created, since the calling process has already proceeded on to the next command.

Process B comes to life and sees

```
ed - $fn <ed1 && ( ... )
```

Thus, process B begins sequential execution of its command sequence. The `ed` command is executed and, if unsuccessful, process B signals process A that the job is done, and dies. If successful, process B creates a new process (process C) to execute everything between the second level of parentheses. Process B then waits for a signal from process C that the job is completed. When process C's signal is received, process B sends a signal to process A, which in turn signals the calling process.

Process C sees the following command sequence:

```
rm ed1; sort -o $fn $fn &
```

Process C first executes the `rm` command. When `rm` has terminated, process C creates another process (process D) to execute the `sort` command, reports process D's process number, signals process B that the job is done, and dies.

Finally, process D sees the following command:


```
sort -o $fn $fn
```

Process D sequentially executes the *sort* command. When *sort* has terminated, process D signals process C that the job is done, and dies. Process C, however, has already died, so process D's signal is ignored. In fact, process D probably begins its task after processes C, B, and A have already died!

Command sequences like the previous example are seldom, if ever, used. Designing a command sequence that performs exactly like you want it to perform takes a great deal of time. There are other constructs available that perform the same functions and are much easier to read and debug. However, command sequences like the previous example are ideally suited to those programmers who want their procedures to be as concise as possible.

Grouping With Braces

Braces are useful for grouping two or more commands together for the purpose of redirecting their combined input or output. Braces do not in themselves cause the creation of new processes. For example, in the following procedure,

```
{
date
ls
pr *
} >dircontents
```

date, *ls*, and *pr* are executed sequentially, and their output is collected in the file **dircontents**. Note that the braces do not create a new process to execute the commands. The braces are simply used to cause the I/O redirection to apply to all three commands.

The `;`, `&`, `&&`, and `||` separators can be used within braces, and braces can be nested. They are most commonly used as shown in the previous example, with each brace appearing on a line by itself, and any number of valid commands and/or control-flow constructs appearing between them.

Control-Flow Constructs

With the introduction of control-flow constructs, procedures cease to be shell scripts, and become shell programs. Control-flow constructs are perhaps the most useful elements of the shell programming language, for they enable you to create powerful shell programs that incorporate conditional testing, branching, and looping.

The FOR Construct

The **for** construct enables you to execute a set of commands once for every new value assigned to a parameter. The **for** construct has the following syntax:

```
for name [ in wordlist ]
do command-list
done
```

Name is any valid parameter name. *Wordlist* contains a list of one or more values that are to be assigned to *name*. One at a time, a value from *wordlist* is assigned to *name*, and the list of commands in *command-list* is executed. Execution terminates when there are no more values left in *wordlist*. If the **in** clause is omitted, then *name* is assigned the value of each positional parameter that is set, and execution terminates when all positional parameters have been used.

Examples

```
for i in *.c
do
    cc $i
    mv a.out `basename $i .c`
done
```

This example is a variation of the **compile** shell script discussed earlier. The parameter **i** is assigned the name of each file in your current working directory that ends in ".c". That file is then compiled, and the resulting *a.out* file is renamed such that its name is the same as its corresponding source file with the ".c" removed. Execution ends when **i** has been assigned the names of all C source files in your current working directory.

```
for dir in /dev /usr /users /lib /etc /bin /tmp
do
    num=`ls $dir | wc -w`
    echo "$num files in $dir"
done
```

This example assigns each directory name to **dir**. The contents of each directory are listed, and the number of files is counted with *wc* and assigned to **num**. The number of files in each directory is then printed.

```
for i
do
    sort -d -o ${i}.srt $i
done
```

Since the **in** clause is missing, **i** is assigned the value of each positional parameter that is set on the command line. The result is that each file that is specified on the command line is sorted. The sorted version is placed in a file having the same name as the unsorted file, with a ".srt" appended to it.

Note that the *wordlist* part of the **for** construct can be almost anything. Some examples are

```
for i in $1
```

which enables you to specify *wordlist* from the command line, or

```
for file in $dir/[a-f]?[1-4].c
```

which causes *wordlist* to include all files in the directory specified by **dir** that begin with a lower-case letter in the range a through f, followed by any single character, followed by a digit in the range 1 through 4, followed by ".c".

Note that **do** or **done** is recognized only when following a new-line or semicolon.

The CASE Construct

The **case** construct enables you to execute a specific set of commands, depending on the value of a parameter. The **case** construct has the following syntax:

```
case $name in
    pattern1 [ | pattern2 ... ] ) command-list1 ;;
    .
    .
    .
esac
```

Name is a dereferenced parameter name. The *patterns* are strings of one or more literal characters and/or the special characters *, ?, [,], and \. The *command-list* is a list of one or more commands to be executed if one of the associated patterns matches the value of *name*. The last command in *command-list* must be terminated with a double semicolon (;).

Examples

```
case $fn in
    *.c)  cc $fn
          mv a.out `basename $fn .c` ;;
    *.f)  fc $fn
          mv a.out `basename $fn .f` ;;
    *.p)  pc $fn
          mv a.out `basename $fn .p` ;;
    *)    echo "$fn: not a source file."
          exit 1 ;;
esac
```

This example compares the value of **fn** with each pattern listed, in the order in which the patterns are listed. If **fn** is a file name ending in ".c", then the commands associated with the "*.c" pattern are executed, and so on. The final pattern consisting of a single asterisk acts as a default condition. If none of the other patterns are matched, the commands associated with the asterisk are executed, since the asterisk matches anything. It is important that the asterisk be listed last, because any patterns following the asterisk are *never* matched. Note that the **case** construct terminates after a match is made.

```
for i
do case $i in
    -[dD]) echo "Please specify directory."
           read dir ;;
    -b|-r) rflag=y ;;
    *)    echo "$i: unknown option."
```

```

                exit 1 ;;
    esac
done

```

This example illustrates how a **case** construct may be included within a **for** construct. This combination is very common, and is most often used to process options from the command line. This particular example accepts **-d**, **-D**, **-b**, and **-r** as valid options, and flags any others as invalid. The **case** construct is executed once for every positional parameter set on the command line. The first pattern matches **-d** or **-D**, both of which require the user to enter a directory name from his terminal (the *read* command is described under *Special Commands* in *sh(1)*, in the HP-UX Reference manual). The second pattern matches either **-b** or **-r**, both of which set **rflag** equal to "y" (note that the second pattern could be written as "[br]"). Finally, any other option prompts an error message, and the process is terminated.

The IF Construct

The **if** construct enables you to execute certain commands, depending on the result of one or more conditional tests. The **if** construct has the following syntax:

```

if command-list1
then command-list2
elif command-list3
then command-list4
    .
    .
else command-listn
fi

```

Only the **if**, **then**, and terminating **fi** are necessary; all **elif** sections and the **else** section are optional. Each *command-list* is a list of one or more commands. The list associated with **if** is executed first. If the last command of the list is successful, the list associated with the first **then** is executed, and the construct is terminated. If the list associated with **if** is unsuccessful, the list associated with the next **elif** is executed. If that **elif**'s list is successful, the list associated with the next **then** is executed, and so on. If all **elif** lists are unsuccessful, the list following **else** is executed, and the **if** construct terminates.

Examples

```

if [ -f -w "$fn" ]
then
    ed - $fn <script
fi

```



This example shows the **if** construct in its simplest form. The square brackets are the alternate syntax for the *test* command, and are equivalent to

```
test -f -w "$fn"
```

Thus, if the file specified by **fn** is both an ordinary file and writable, then it is edited according to the *ed* commands in **script**. Otherwise, no action is taken. Note that this **if** construct is equivalent to

```
test -f -w "$fn" && ed - $fn <script

if [ -f -r /users/kb/$fn ]
then
    diff /users/kb/$fn $fn >difffile
elif [ -f -r /users/bill/$fn ]
then
    diff /users/bill/$fn $fn >difffile
elif [ -f -r /users/fred/$fn ]
then
    diff /users/fred/$fn $fn >difffile
else
    echo "Can't find $fn for comparison. "
fi
```

This example shows all the parts of an **if** construct. Here, the directories */users/kb*, */users/bill*, and */users/fred* are searched for the file specified by **fn**. If it is found, it is compared to a file with the same name in your current working directory. The output from *diff* is redirected into a file called **difffile**. The **else** clause functions as a default; if all other tests fail, the *echo* command is executed.

The WHILE Construct

The **while** construct repeatedly executes a list of commands and, if the last command in the list is successful, executes a second list of commands. The **while** construct has the following syntax:

```
while command-list1
do command-list2
done
```

Command-list1 is a list of one or more commands that is repeatedly executed. If the last command in this list executes successfully, the commands in *command-list2* are executed. The loop terminates when the last command in *command-list1* executes unsuccessfully.

Example

```
while [ -n "$1" ]
do
    sort -d -o $1 $1
    ed - $1 <edscript
    pr -f $1 >/dev/lp
    shift
done
```

This example operates on the positional parameter **\$1**. The **while** loop continues as long as the value of **\$1** has a non-zero length. First, the file name specified by **\$1** is sorted. Then, it is edited according to the *ed* commands in **edscript**, and printed on the system's line printer. The *shift* command moves the value of **\$2** to **\$1**, **\$3** to **\$2**, **\$4** to **\$3**, and so on. Thus, different files are sorted, edited, and printed each time through the loop, even though the parameter name stays the same. The loop terminates when a null value is shifted in for **\$1**.

The UNTIL Construct

The **until** construct repeatedly executes a list of commands and, if the last command in the list is unsuccessful, executes a second list of commands. The **until** construct terminates when the last command in the first command list executes successfully. Thus, the **while** and **until** constructs differ only in the condition required to terminate the loop. The **until** construct has the following syntax:

```
until command-list1
do command-list2
done
```

Command-list1 is a list of one or more commands that is repeatedly executed. If the last command in *command-list1* executes unsuccessfully, the commands in *command-list2* are executed. The **until** construct terminates when the last command in *command-list1* executes successfully.

Example

```
until who | grep fred >/dev/null
do
    sleep 300
done
write fred <fredletter
```

This example checks to see if Fred is logged in. If not, the process "sleeps" for five minutes, and checks again. This continues until Fred finally logs in, at which time the **until** construct terminates, and the message in **fredletter** is sent to Fred via the *write* command. Note that the output from *grep* is redirected to */dev/null*, which essentially discards the data into the system's "bit bucket". A shell program like this can be executed in a background process to ensure that a particular user gets an important message as soon as he logs in.

An Example Shell Program

The following is a shell program that is used to print files on the system's line printer, */dev/lp*.

```
rd = n
range = *
dir = `pwd`
days =
#
# Parse options.
#
while [ -n "$1" ]
do case $1 in
    -c)    rd = y          # raw dump
           shift ;;
    -d)    shift          # directory name
           dir = $1
           shift ;;
    -f)    shift          # last-modified time
           days = $1
           shift ;;
    -r)    shift          # files to print
           range = $1
           shift ;;
    *)    echo "$1: unrecognized option."
           exit 1 ;;
esac
done
#
# Move to specified directory.
#
cd $dir
#
# If -f specified, move all affected files.
#
if [ -n "$days" ]
then
    mkdir ../temp
    find . -mtime $days -exec mv {} ../temp \;
fi
#
# Begin printing.
#
for i in $range
do case $rd in
    n)    pr -f -r $i >/dev/lp ;;
    y)    cat $i >/dev/lp
           echo "\n\n" >/dev/lp ;;
esac
done
```

```

#
# Printing is done. Clean-up time.
#
test -n "$days" && (mv ../temp/* .; rmdir ../temp)
exit 0

```

This shell program, called **print**, accepts four options:

the **-c** option, which specifies that the contents of the files are to be printed with no formatting (i.e. a "raw dump"). The *cat* command is used for this. The **-c** option requires no argument. If the **-c** option is not specified, then the contents of the files are printed out with a heading and page numbers. The *pr* command is used for this.

the **-d** option, which implies that the argument to follow specifies the name of the directory containing the files to be printed. If the **-d** option is not specified, the user's current working directory is used.

the **-f** option, which implies that the argument to follow specifies an argument for the *find* command. If **-f** is specified, a temporary directory called **temp** is created in the parent directory, and the *find* command is used to move all files of a certain modification date to **temp**, thus excluding them from the printing. The **-f** argument can have the following three forms:

- + *n* exclude those files modified more than *n* days ago;
- n* exclude those files modified less than *n* days ago;
- n* exclude those files modified exactly *n* days ago.

This argument is combined with the **-mtime** option of the *find* command (see *find(1)* in the HP-UX Reference manual). If the **-f** option is not specified, no files are excluded on the basis of modification date.

the **-r** option, which implies that the argument to follow specifies a string of literal and/or special characters. The string is used in the **in** clause of a **for** construct to print a subset of the files in the directory. If the **-r** option is not specified, an asterisk is used, causing all files to be printed.

The following examples show some of the valid ways to invoke **print**:

```
print -c
```

causes all the files in the current working directory to be printed in "raw" form using *cat*.

```
print -d /users/bill/Cprogs -r \[a-f\]\* -f +3 -c
```

does several things. First, the current working directory is changed to */users/bill/Cprogs*. Then, the directory */users/bill/temp* is created, and all files in **Cprogs** modified more than 3 days ago are moved to **temp**, thus excluding them from the printing. Finally, all files in **Cprogs** that begin with a lower-case letter in the range a through f are printed in "raw" form using *cat*. Note that the special characters in **[a-f]*** must be quoted to prohibit the shell from expanding the pat-

tern, and replacing it with the files that match it in the current working directory. When the printing is done, all the files in **temp** are moved back to **Cprogs**, and **temp** is removed.

```
print -r thesis -d /users/bill/school
```

prints the single file **thesis** in the directory */users/bill/school*. The *pr* command is used to produce a formatted printing.

Using the **case** construct to parse options enables you to specify them in any order on the command line. The only requirements are that the options be immediately followed by their implied arguments, and that all options and arguments be delimited by spaces.

For More Information

The *sh(1)* entry in the HP-UX Reference manual functions as a comprehensive, though somewhat cryptic, reference for the shell programming language. Some topics are not covered in this chapter because of infrequent use, or because one example is sufficient to give guidance in several areas. Most of the omitted topics are related to the shell's special commands, which are discussed under *Special Commands* in the *sh(1)* entry. You should read this section thoroughly to familiarize yourself with the many commands that are built directly into the shell.

Many HP-UX commands are actually shell programs. The HP-UX Reference manual specifies which commands are shell programs under the appropriate manual entries. The following is a list of some of them:

```
/etc/rc  
/etc/whodo  
/etc/mkdev
```

It is helpful to examine the contents of these commands to see how the shell programming language is used. Since these files contain ASCII data, you can print them out using *cat*, provided your system administrator has assigned permissions to these files that enable you to do so.

Table of Contents

UNIX Programming - Second Edition

Abstract.....	1
Introduction.....	2
Basics.....	2
Program Arguments.....	2
The Standard Input and Standard Output.....	2
The Standard I/O Library.....	4
File Access.....	4
Error Handling - Stderr and Exit.....	6
Miscellaneous I/O Functions.....	6
Low-Level I/O.....	6
File Descriptors.....	6
Read and Write.....	7
Open, Creat, Close, Unlink.....	8
Random Access - Seek and Lseek.....	10
Error Processing.....	10
Processes.....	11
The "System" Function.....	11
Low-Level Process Creation - Execl and Execv.....	11
Control of Processes - Fork and Wait.....	12
Pipes.....	13
Signals - Interrupts and All That.....	15
References.....	18
Appendix - The Standard I/O Library.....	19
General Usage.....	19
Calls.....	19

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX[†] system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

November 12, 1978

[†]UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the `<` convention: if `prog` uses `getchar`,

then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the “standard output,” which is also by default the terminal. The output can be captured on a file by using `>`: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main() /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (`/usr/include/stdio.h`) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the

program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The “Standard I/O Library” is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is `wc`, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.)

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("`r`"), write ("`w`"), or append ("`a`").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. `getc` and `putc` return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write `wc`. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — `stderr` and `Exit`

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `wc` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns EOF at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns NULL at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called “opening” the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user’s terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the “shell”) runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);
```

```
n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn’t equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time (“unbuffered”), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```

#define BUFSIZE 512 /* best size for PDP-11 UNIX */

main() /* copy input to output */
{
    char buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}

```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```

#define CMASK 0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}

```

c *must* be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```

#define CMASK 0377 /* for making char's > 0 */
#define BUFSIZE 512

getchar() /* buffered version */
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}

```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;
```

```
fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility `cp`, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2) /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0); /* get to pos */
    return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given `offset` by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of `-1`. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed

because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember that `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2. Low-Level Process Creation — `execl` and `execv`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);    /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the command and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to

return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `exec1`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `exec1`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.


```

#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);
    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);
        _exit(1); /* disaster has occurred if we get here */
    }
    if (popen_pid == -1)
        return(NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}

```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```

close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));

```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```

#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
    int status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}

```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```

#include <signal.h>
...
signal(SIGINT, SIG_IGN);

```

causes interrupts to be ignored, while

```

signal(SIGINT, SIG_DFL);

```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr()
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf  sjbuf;

main()
{
    int (*istat)(), onintr();

    istat = signal(SIGINT, SIG_IGN); /* save original status */
    setjmp(sjbuf); /* save current stack position */
    if (istat != SIG_IGN)
        signal(SIGINT, onintr);

    /* main processing loop */
}
```

```

onintr()
{
    printf("\nInterrupt\n");
    longjmp(sjbuf);    /* return to saved state */
}

```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```

if (getchar() == EOF)
    if (intflag)
        /* EOF caused by interrupt */
    else
        /* true end-of-file */

```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like `!` in the editor) whereby other programs can be executed. Then the code should look something like this:

```

if (fork() == 0)
    execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */

```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```

#include <signal.h>

system(s) /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)(), (*qstat)();

    if ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}

```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```

#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1

```

References

- [1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978.
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.
- [3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore `_` to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file
`stdout` The name of the standard output file
`stderr` The name of the standard error file
`EOF` is actually `-1`, and is the value returned by the read routines on end-of-file or error.
`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error
`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.
`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.
`getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `fileno`
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. Calls

```
FILE *fopen(filename, type) char *filename, *type;  
opens the file and, if needed, allocates a buffer for it. filename is a character string specifying the name. type is a character string (not a single character). It may be "r", "w", or "a" to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL the attempt to open failed.  
FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;
```

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

```
int getc(ioptr) FILE *ioptr;
    returns the next character from the stream named by ioptr, which is a pointer to a file
    such as returned by fopen, or the name stdin. The integer EOF is returned on end-of-
    file or when an error occurs. The null character \0 is a legal character.
```

```
int fgetc(ioptr) FILE *ioptr;
    acts like getc but is a genuine function, not a macro, so it can be pointed to, passed as an
    argument, etc.
```

```
putc(c, ioptr) FILE *ioptr;
    putc writes the character c on the output stream named by ioptr, which is a value
    returned from fopen or perhaps stdout or stderr. The character is returned as value,
    but EOF is returned on error.
```

```
fputc(c, ioptr) FILE *ioptr;
    acts like putc but is a genuine function, not a macro.
```

```
fclose(ioptr) FILE *ioptr;
    The file corresponding to ioptr is closed after any buffers are emptied. A buffer allocated
    by the I/O system is freed. fclose is automatic on normal termination of the program.
```

```
fflush(ioptr) FILE *ioptr;
    Any buffered information on the (output) stream named by ioptr is written out. Output
    files are normally buffered if and only if they are not directed to the terminal; however,
    stderr always starts off unbuffered and remains so unless setbuf is used, or unless it is
    reopened.
```

```
exit(errcode);
    terminates the process and returns its argument as status to the parent. This is a special
    version of the routine which calls fflush for each output file. To terminate without flush-
    ing, use _exit.
```

```
feof(ioptr) FILE *ioptr;
    returns non-zero when end-of-file has occurred on the specified input stream.
```

```
ferror(ioptr) FILE *ioptr;
    returns non-zero when an error has occurred while reading or writing the named stream.
    The error indication lasts until the file has been closed.
```

```
getchar();
    is identical to getc(stdin).
```

```
putchar(c);
    is identical to putc(c, stdout).
```

```
char *fgets(s, n, ioptr) char *s; FILE *ioptr;
    reads up to n-1 characters from the stream ioptr into the character pointer s. The read
    terminates with a newline character. The newline character is placed in the buffer followed
    by a null character. fgets returns the first argument, or NULL if error or end-of-file
    occurred.
```

```
fputs(s, ioptr) char *s; FILE *ioptr;
    writes the null-terminated string (character array) s on the stream ioptr. No newline is
    appended. No value is returned.
```

```
ungetc(c, ioptr) FILE *ioptr;
```

The argument character *c* is pushed back on the input stream named by *ioptr*. Only one character may be pushed back.

`printf(format, a1, ...) char *format;`

`fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;`

`sprintf(s, format, a1, ...) char *s, *format;`

`printf` writes on the standard output. `fprintf` writes on the named output stream.

`sprintf` puts characters in the character array (string) named by *s*. The specifications are as described in section `printf(3)` of the *UNIX Programmer's Manual*.

`scanf(format, a1, ...) char *format;`

`fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;`

`sscanf(s, format, a1, ...) char *s, *format;`

`scanf` reads from the standard input. `fscanf` reads from the named input stream.

`sscanf` reads from the character string supplied as *s*. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string *format*, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

`scanf` returns as its value the number of successfully matched and assigned input items.

This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

`fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`

reads *nitems* of data beginning at *ptr* from file *ioptr*. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the `fopen` call.

`fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`

Like `fread`, but in the other direction.

`rewind(ioptr) FILE *ioptr;`

rewinds the stream named by *ioptr*. It is not very useful except on input, since a rewound output file is still open only for output.

`system(string) char *string;`

The *string* is executed by the shell as if typed at the terminal.

`getw(ioptr) FILE *ioptr;`

returns the next word from the input stream named by *ioptr*. EOF is returned on end-of-file or error, but since this a perfectly good integer `feof` and `ferror` should be used. A "word" is 16 bits on the PDP-11.

`putw(w, ioptr) FILE *ioptr;`

writes the integer *w* on the named output stream.

`setbuf(ioptr, buf) FILE *ioptr; char *buf;`

`setbuf` may be used after a stream has been opened but before I/O has started. If *buf* is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char buf[BUFSIZ];
```

`fileno(ioptr) FILE *ioptr;`

returns the integer file descriptor associated with the file.

`fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;`

The location of the next byte in the stream named by *ioptr* is adjusted. *offset* is a long integer. If *ptrname* is 0, the offset is measured from the beginning of the file; if *ptrname* is 1, the offset is measured from the current read or write pointer; if *ptrname* is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When

this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the `ptrname` must be 0).

`long ftell(ioptr) FILE *ioptr;`

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`

allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`

allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`cfree(ptr) char *ptr;`

Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

Table of Contents

Make - A Program for Maintaining Computer Programs

Abstract.....	1
Introduction.....	2
Basic Features.....	2
Description Files and Substitutions.....	4
Command Usage.....	5
Implicit Rules.....	6
Example.....	7
Suggestions and Warnings.....	9
Acknowledgments.....	9
References.....	9
Appendix. Suffixes and Transformation Rules.....	10

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978

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 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.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences 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. *Make* was designed for use on Unix, but a version runs on GCOS.

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 *IS* 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 -IS -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 long-winded 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 -IS -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 just 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 name by 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. \$\$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: \$*, \$@, \$?, and \$<. 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. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and 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:

†UNIX is a Trademark of Bell Laboratories.

```
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.

1. 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.
2. 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.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
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 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. The contents of the description files override the built-in rules if they are present).

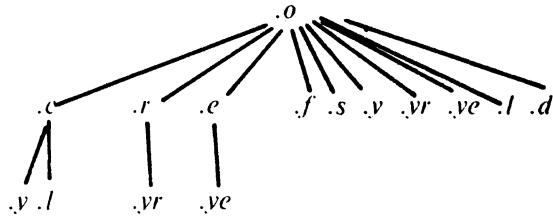
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is “made”.

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

<i>.o</i>	Object file
<i>.c</i>	C source file
<i>.e</i>	Efl source file
<i>.r</i>	Ratfor source file
<i>.f</i>	Fortran source file
<i>.s</i>	Assembler source file
<i>.y</i>	Yacc-C source grammar
<i>.yr</i>	Yacc-Ratfor source grammar
<i>.ye</i>	Yacc-Efl source grammar
<i>.l</i>	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*, *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.

Example

As an example of the use of *make*, we will 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.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES = -IS
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 -IS -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"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of 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. Obvious 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.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, “Yacc — Yet Another Compiler-Compiler”, Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, “Lex — A Lexical Analyzer Generator”, Computing Science Technical Report #39, October 1975.

Appendix. Suffixes and Transformation 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:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

Table of Contents

Source Code Control System User's Guide

General	1
SCCS for Beginners.....	1
Terminology.....	2
Creating an SCCS File via "Admin"	2
Retrieving a File via "Get"	2
Recording Changes via "Delta"	3
Additional Information About "Get"	4
The "Help" Command.....	5
Delta Numbering	5
SCCS Command Conventions	7
SCCS Commands	8
The "Get" Command.....	9
ID Keywords	10
Retrieval of Different Versions.....	10
Retrieval With Intent to Make a Delta	12
Concurrent Edits of Different SID.....	13
Concurrent Edits of Same SID	13
Keyletters That Affect Output.....	14
The "Delta" Command	15
The "Admin" Command.....	17
Creation of SCCS Files.....	17
Inserting Commentary for the Initial Delta	18
Initialization and Modification of SCCS File Parameters	18
The "Prs" Command	19
The "Help" Command.....	21
The "Rmdel" Command.....	21
The "Cdc" Command.....	22
The "What" Command.....	22
The "Sccsdiff" Command.....	23
The "Comb" Command.....	23
The "Val" Command	24
SCCS Files	24
Protection.....	24
Formatting.....	25
Auditing	25
An SCCS Interface Program.....	26
General.....	26
Function.....	26
A Basic Program.....	27
Linking and Use	27
Determination of New SID (Table 4.A)	28
SCCS Interface Program (Table 4.B).....	29



4. SOURCE CODE CONTROL SYSTEM USER'S GUIDE

GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX software commands which help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for the following:

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This section, together with relevant portions of the UNIX System User's Manual is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands, with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

Neither the implementation of SCCS nor the installation procedure for SCCS are described in this section.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section **N** of the UNIX System User's Manual.

SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.

To supplement the material in this section, the detailed SCCS command descriptions in the UNIX System User's Manual should be consulted.

A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *SCCS IDentification* string (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.1", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following **admin** command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*.

```
admin -i lang s.lang
```

All SCCS files must have names that begin with "s.", hence, *s.lang*. The **-i** keyletter, together with its value *lang*, indicates that **admin** is to create a new SCCS file and "initialize" the new SCCS file with the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the **get** command in the part "SCCS COMMANDS". In the following examples, this warning message is not shown, although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the **get** command) as follows:

```
rm lang
```

C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following **get** command:

```
get s.lang
```

The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

```
1.1
5 lines
```

This means that **get** retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get s.lang" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the **delta** command, the **get** command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The **-e** keyletter causes **get** to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the *p-file*, will be read by the **delta** command. The **get** command prints the same messages as before except that the SID of the version to be created through the use of **delta** is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file *lang* may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made. For example:

```
comments? added more languages
```

The **delta** command then reads the *p-file* and determines what changes were made to the file *lang*. The **delta** command does this by doing its own **get** to retrieve the original version and by applying the **diff(1)** command to the original version and the edited version.

When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, **delta** outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number “1.2” is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

E. Additional Information About “get”

As shown in the previous example, the command

```
get s.lang
```

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs. Note that omitting the level number of the SID (as in “get -r1 s.lang”) is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This is done with the **get** command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version *before* release 2. The **get** command also interprets this as a request to change the release number of the delta the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to **delta** via the *p-file*. The **get** command then outputs

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version **delta** will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and **delta** executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

the user will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

F. The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the **help** command:

```
help co1
```

This produces the following output:

```
co1:
" not an SCCS file "
A file that you think is an SCCS file
does not begin with the characters "s."
```

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Thus, the evolution of a particular file may be represented as in Fig. 4.1.

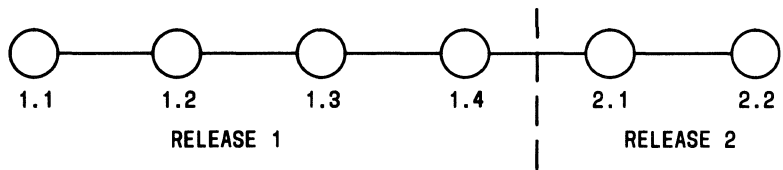


Fig. 4.1— Evolution of an SCCS File

Such a structure may be termed the “trunk” of the SCCS tree. Figure 4.1 represents the normal sequential development of an SCCS file in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are not dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas precisely as shown in Fig. 4.1. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a branch of the tree, and its name consists of four components; the release and level numbers, as with trunk deltas, plus the “branch” and “sequence” numbers. The delta name will appear as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Fig. 4.2.

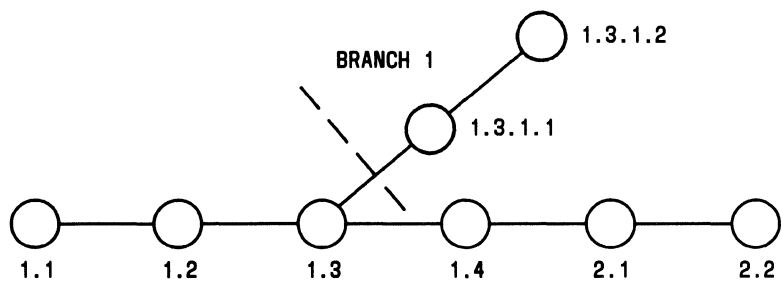


Fig. 4.2 — Tree Structure With Branch Deltas

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain

exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Fig. 4.3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

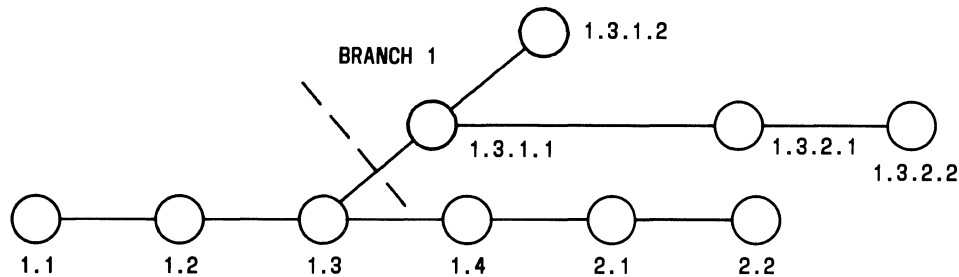


Fig. 4.3 — Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- keyletter arguments
- file arguments.

Keyletter arguments (hereafter called simply “keyletters”) begin with a minus sign (–), followed by a lower-case alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes via **chmod**(1)] in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name “–” (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line

as the name of an SCCS file to be processed. The standard input is read until end of file. This feature is often used in pipelines with, for example, the **find**(1) or **ls**(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Some-what different argument conventions apply to the **help**, **what**, **sccsdiff**, and **val** commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed in this part. For a complete description of all such flags, see **admin**(1) section in the UNIX System User's Manual.

The distinction between the real user [see **passwd**(1)] and the effective user of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system). This subject is discussed further in "SCCS FILES".

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, given the same mode [see **chmod**(1)] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files*. The files may be useful in the event of system crashes or similar situations.

The SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses may be used as an argument to the **help** command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the UNIX System User's Manual and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.

prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the get command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

A. The "get" Command

The **get** command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file*. The *g-file* name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the **get** command is invoked.

The most common invocation of **get** is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated *g-file* (file "abc") is given mode 444 (read-only) since this particular way of invoking **get** is intended to produce *g-files* only for inspection, compilation, etc., and not for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```


produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc. within the *g-file*, so this information will appear in a load module when one is eventually created. The SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and %M% is defined as the name of the *g-file*. Thus, executing **get** on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see **get(1)** in the UNIX System User's Manual.

Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **-r** keyletter of **get**.

The **-r** keyletter is used to specify a SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get -r1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3  
234 lines
```

When a 2- or 4-component SID is specified as a value for the `-r` keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. Thus, the above command might output:

```
3.7  
213 lines
```

If the given release does not exist, `get` retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file *s.abc* and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6  
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8  
89 lines
```

The `-t` keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no `-r` keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5  
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5  
46 lines
```

Retrieval With Intent to Make a Delta

Specification of the `-e` keyletter to the `get` command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes `get` to check:

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing `get` is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.
2. The release (R) of the version being retrieved satisfies the relation:

floor is $<$ or $=$ to R which is
 $<$ or $=$ to ceiling

to determine if the release being accessed is a protected release. The “floor” and “ceiling” are specified as flags in the SCCS file.

3. The release (R) is not locked against editing. The “lock” is specified as a flag in the SCCS file.
4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the `j` flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the `-e` keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable *g-file* already exists, `get` terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by `get` when the `-e` keyletter is specified because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, `get` does not need to check for the presence of ID keywords within the *g-file*, so the message

No id keywords (cm7)

is never output when `get` is invoked with the `-e` keyletter.

In addition, the `-e` keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the `delta` command.

The following is an example of the use of the `-e` keyletter:

```
get -e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the `-r` and/or `-t` keyletters are used together with the `-e` keyletter, the version retrieved for editing is a specified by the `-r` and/or `-t` keyletters.

The keyletters **-i** and **-x** may be used to specify a list [see **get(1)** in the UNIX System User's Manual for the syntax of such a list] of deltas to be included and excluded, respectively, by **get**. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be not applied. This may be used to undo in the version of the SCCS file to be created the effects of a previous delta. Whenever deltas are included or excluded, **get** checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

Warning: *The **-i** and **-x** keyletters should be used with extreme care.*

The **-k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of **get** with the **-e** keyletter or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **-k** keyletter is identical to one produced by **get** executed with the **-e** keyletter. However, no processing related to the *p-file* takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows a number of deltas to be “in progress” at any given time. This means that a number of **get** commands with the **-e** keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* (created by the **get** command invoked with the **-e** keyletter) is named by replacing the “s.” in the SCCS file name with “p.”. It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The *p-file* contains the following information for each delta that is still “in progress”:

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing **get**.

The first execution of **get -e** causes the creation of the *p-file* for the corresponding SCCS file. Subsequent executions only update the *p-file* with a line containing the above information. Before updating, however, **get** checks that no entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of **get** should be carried out from different directories. Otherwise, only the first execution will succeed since subsequent executions would attempt to overwrite a writable *g-file*, which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. See “Protection” under the part “SCCS FILES” for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 4.A shows, for the most useful cases, the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

Concurrent Edits of Same SID

Under normal conditions, **gets** for editing (**-e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, **delta** must be executed before a subsequent **get** for editing is executed at

the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

Keyletters That Affect Output

Specification of the **-p** keyletter causes **get** to write the retrieved text to the standard output rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get -p s.abc > arbitrary-file-name
```

The **-p** keyletter is particularly useful when used with the “!” or “\$” arguments of the **send(1C)** command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains:

```
//plicomp job job-card-information
//step1 exec plicke
//pli.sysin dd *
~s
~!get -p -rREL MOD
/*
//
```

will **send** the highest level of release 3 of file *s.abc*. Note that the line “~s”, which causes **send** to make ID keyword substitutions before detecting and interpreting control lines, is necessary if **send** is to substitute “s.abc” for MOD and “3” for REL in the line “~!get -p -rREL MOD”.

The **-s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user’s terminal and is often used in conjunction with the **-p** keyletter to “pipe” the output of **get**, as in:

```
get -p -s s.abc | nroff
```

The **-g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get -g -r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the **-g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get -e -g s.abc
```

The **-l** keyletter causes the creation of an *l-file*, which is named by replacing the “s.” of the SCCS file name with “l.”. This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table (whose format is described in **get(1)** in the UNIX System User’s Manual) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get -r2.3 -l s.abc
```

generates an *l-file* showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of “p” with the **-l** keyletter, as in:

```
get -lp -r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The **-g** keyletter may be used with the **-l** keyletter to suppress the actual retrieval of the text.

The **-m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated *g-file* to be preceded by the value of the **%M%** ID keyword and a tab character. The **-n** keyletter is most often used in a pipeline with **grep(1)**. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the **%M%** ID keyword and a tab (this is the effect of the **-n** keyletter) and followed by the line in the format produced by the **-m** keyletter. Because use of the **-m** keyletter and/or the **-n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must not be used for creating a delta. Therefore, neither the **-m** keyletter nor the **-n** keyletter may be specified together with the **-e** keyletter.

See **get(1)** in the UNIX System User’s Manual for a full description of additional **get** keyletters.

B. The “delta” Command

The **delta** command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the **delta** command requires the existence of a *p-file*. The **delta** command examines the *p-file* to verify the presence of an entry containing the user’s login name. If none is found, an error message results. The **delta** command also performs the same permission checks that **get** performs when invoked with the **-e**

keyletter. If all checks are successful, **delta** determines what has been changed in the *g-file* by comparing it via **diff(1)** with its own temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the “s.” of the SCCS file name with “d.”) and is obtained by performing an internal **get** at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing **delta** because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed **get** with the **-e** keyletter more than once on the same SCCS file), the **-r** keyletter must be used with **delta** to specify an SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new line character. The user’s response may be up to 512 characters long with new lines, not intended to terminate the response, escaped by backslash “\”.

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt “comments?”. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests [MRs]) and that it is desirable or necessary to record such MR number(s) within each delta.

The **-y** and/or **-m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input:

```
delta -y "descriptive comment" -m "mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell procedure [see **sh(1)** in the UNIX System User’s Manual.]

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that if **delta** is invoked with more than one file argument and the first file named has a **v** flag all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If in the process of making a delta **delta** finds no ID keywords in the edited *g-file*, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a **get** without the **-e** keyletter (recall that ID keywords are replaced by **get** in that case) or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file* which is described in the part "SCCS COMMAND CONVENTIONS". If there is only one entry in the *p-file*, then the *p-file* itself is removed.

In addition, **delta** removes the edited *g-file* unless the **-n** keyletter is specified. Thus:

```
delta -n s.abc
```

will keep the *g-file* upon completion of processing.

The **-s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by **diff(1)**.

C. The "admin" Command

The **admin** command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files. (Discussed in "Auditing" under the part "SCCS FILES".) Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command upon that file.

Creation of SCCS Files

An SCCS file may be created by executing the command

```
admin -ifirst s.abc
```


in which the value “first” of the `-i` keyletter specifies the name of a file from which the text of the initial delta of the SCCS file `s.abc` is to be taken. Omission of the value of the `-i` keyletter indicates that **admin** is to read the standard input for the text of the initial delta. Thus, the command

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the same invocation of the command also sets the `i` flag (not to be confused with the `-i` keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the `-i` keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally “1”, and its level number is always “1”. Thus, the first delta of an SCCS file is normally “1.1”. The `-r` keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named “3.1” rather than “1.1”. Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the `-i` keyletter.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (`-y` keyletter) and/or MR numbers (`-m` keyletter) in exactly the same manner as for **delta**. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (`-y` keyletter) are omitted, a comment line of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (`-m` keyletter), the `v` flag must also be set (using the `-f` keyletter described below). The `v` flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a “delta commentary” [see **scsfile(4)** in the UNIX System User’s Manual] in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that the `-y` and `-m` keyletters are only effective if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the `-t` keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the `-t` keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file *desc*.

When processing an *existing* SCCS file, the **-t** keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*; omission of the file name after the **-t** keyletter as in

```
admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted through the use of the **-f** and **-d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See **admin(1)** in the *UNIX System User's Manual* for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command. The **-f** keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fi -fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the **get** command will use to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the **%M%** ID keyword.) Note that several **-f** keyletters may be supplied on a single invocation of **admin** and that **-f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **-d** keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **-d** keyletters may be supplied on a single invocation of **admin** and may be intermixed with **-f** keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the **-a** keyletter is used. For example:

```
admin -axyz -awql -a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **-a** keyletter may be used whether **admin** is creating a new SCCS file or processing an existing one and may appear several times. The **-e** keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

D. The "prs" Command

The **prs** command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification," supplied by the user via the **-d** keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently being processed, and **:C:** is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see **prs(1)** in the UNIX System User's Manual.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs -d " :I: this is the top delta for :F: :I: " s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the **-r** keyletter. For example:

```
prs -d " :F: : :I: comment line is: :C: " -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the **-r** keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the **-l** or **-e** keyletters. The **-e** keyletter substitutes data keywords for the SID designated via the **-r** keyletter and all deltas created earlier. The **-l** keyletter substitutes data keywords for the SID designated via the **-r** keyletter and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4  
1.3  
1.2.1.1  
1.2  
1.1
```

and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3  
3.2  
3.1  
2.2.1.1  
2.2  
2.1  
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the `-e` and `-l` keyletters.

E. The “help” Command

The **help** command prints explanations of SCCS commands and of messages that these commands may print. Arguments to **help**, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. The **help** command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces

```
ge5:
" nonexistent sid "
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
rmdel -rSID name ...
```

F. The “rmdel” Command

The **rmdel** command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a “leaf” delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Fig. 4.3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed then deltas 1.3.2.1 and 2.1 can be removed, etc.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The `-r` keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta “2.3” of the SCCS file. Before removal of the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$

The **rmDEL** command also checks that the SID specified is not that of a version for which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified can not be locked against editing. That is, if the **l** flag is set [see **admin(1)** in the UNIX System User's Manual], the release specified *must* not be contained in the list]. If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" (delta) to "R" (removed).

G. The "cdc" Command

The **cdc** command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the **rmDEL** command, except that the delta to be processed is not required to be a leaf delta. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by **cdc** in the same manner as that of **delta**. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

H. The "what" Command

The **what** command is used to find identifying information within any UNIX system file whose name is given as an argument to **what**. Directory names and a name of "-" (a lone minus sign) are not treated specially, as they are by other SCCS commands, and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword [see **get(1)**], and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), new line, or (nonprinting) NUL character. For example, if the SCCS file *s.prog.c* (a C language program) contains the following line:

```
char id[] " %Z% %M%:%I%";
```

and then the command

```
get -r3.4 s.prog.c
```

is executed, the resulting *g-file* is compiled to produce "prog.o" and "a.out". Then the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:3.4
prog.o:
  prog.c:3.4
a.out:
  prog.c:3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

I. The “**sccsdiff**” Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **-r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr(1)** command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of “-” (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff(1)**. The following is an example of the invocation of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

J. The “**comb**” Command

The **comb** command generates a “shell procedure” [see **sh(1)** in the UNIX System User’s Manual] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining other specified deltas. The SCCS files that contain deltas no longer useful should be discarded. It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the “shape” of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Fig. 4.3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **-p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **-c** keyletter specifies a list [see **get(1)** in the UNIX System User’s Manual for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.

The **-s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

K. The "val" Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin(1)** in the UNIX System User's Manual for a description of the flags].

The **val** command treats the special argument "-" differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example:

```
val -  
-yc -mabc s.abc  
-mxyz -ypl1 s.xyz
```

first checks if file *s.abc* has a value "c" for its "type" flag and value "abc" for the "module name" flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error [see **val(1)** for a description of possible errors and the codes]. In addition, an appropriate diagnostic is printed unless suppressed by the **-s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

A. Protection

The SCCS relies on the capabilities of the UNIX software for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the "release lock" flag, the "release floor" and "ceiling" flags, and the "user list".

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode *not* be changed as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the *x-file*, see "SCCS COMMAND CONVENTIONS") and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s."

When only one user uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (e.g., in large software development projects), one user (equivalently, one user ID) must be chosen as the “owner” of the SCCS files and be the one who will “administer” them (e.g., by using the **admin** command). This user is termed the “SCCS administrator” for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the “set user ID on execution” bit “on” [see **chmod**(1) in the UNIX System User’s Manual], so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command’s execution. Thus, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the “user list” for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. These other users are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**. The project-dependent interface program, as its name implies, must be custom-built for each project.

B. Formatting

The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the “logical” sum of all the characters of the file (<i>not</i> including this checksum itself).
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in **sccsfile**(5). The checksum is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files they may be processed by various UNIX software commands, such as **ed**(1), **grep**(1), and **cat**(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

Caution: *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more “blocks”) can be destroyed. The SCCS commands (like most UNIX software commands)

issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with **ed(1)**]. No SCCS command will process a corrupted SCCS file except the **admin** command with the **-h** or **-z** keyletters, as described below.

It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the **admin** command with the **-h** keyletter on all SCCS files:

```
admin -h s.file1 s.file2 ...
      or
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the **ls(1)** command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request the file be restored from a backup copy. In the case of minor damage, repair through use of the editor **ed(1)** may be possible. In the latter case after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable.

AN SCCS INTERFACE PROGRAM

A. General

In order to permit UNIX system users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.

B. Function

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel**, **cdc**, and **unget** commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

The interface program must be owned by the SCCS administrator, must be executable by nonowners, and must have the “set user ID on execution” bit “on” [see **chmod**(1) in the UNIX System User’s Manual] so that, when executed, the effective user ID is the user ID of the administrator. This program’s function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command’s execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**.

C. A Basic Program

When a UNIX program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke [see **exec**(2) in the UNIX System User’s Manual].

A generic interface program (*inter.c*, written in C language) is shown in Table 4.B. Note the reference to the (unsupplied) function “filearg”. This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function “filearg” could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

D. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory “/x1/xyz/sccs”. Thus, the command sequence

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (the “...” represent other arguments that may be required). The proper mode and the “set user ID on execution” bit are set by executing:

```
chmod 4755 inter
```

For example, new links are created by:

```
ln inter get
ln inter delta
ln inter rmdel
```

The names of the links may be arbitrary provided the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter PATH [see **sh**(1) in the UNIX System User’s Manual] specifies directory “/x1/xyz/sccs” as the one to be searched first for executable commands may execute, e.g.:

```
get -e /x1/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link “get”). The interface program then executes “/usr/bin/get” (the actual SCCS **get** command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname “/x1/xyz/sccs” so that the user would only have to specify

```
get -e s.abc
```

to achieve the same results.

TABLE 4.A
DETERMINATION OF NEW SID

CASE	SID SPECIFIED*	-b KEYLETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
1	none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
2	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
3	R	no	R > mR	mR.mL	R.1§
4	R	no	R = mR	mR.mL	mR.(mL + 1)
5	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7	R	—	R < mR and R does not exist	hR.mL**	hR.mL.(mB + 1).1
8	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9	R.L	no	No trunk successor	R.L	R.(L + 1)
10	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11	R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mB + 1).1
12	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1)
15	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB + 1).1

* “R”, “L”, “B”, and “S” are the “release”, “level”, “branch”, and “sequence” components of the SID, respectively; “m” means “maximum”. Thus, for example, “R.mL” means “the maximum level number within release R”; “R.L.(mB + 1).1” means “the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R”. Note that if the SID specified is of the form “R.L”, “R.L.B”, or “R.L.B.S”, each of the specified components must exist.

† The -b keyletter is effective only if the b flag [see admin(1)] is present in the file. In this table, an entry of “—” means “irrelevant”.

‡ This case applies if the d (default SID) flag is not present in the file. If the d flag is present in the file, the SID obtained from the d flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the first delta in a new release.

** “hR” is the highest existing release that is lower than the specified, nonexistent, release R.

TABLE 4.B

SCCS INTERFACE PROGRAM "inter.c"

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i;
    char cmdstr[LENGTH]

    /*
    Process file arguments (those that don't begin with "-").
    */
    for (i = 1; i < argc; i++)
        if (argv[i][0] != '-')
            argv[i] = filearg(argv[i]);

    /*
    Get "simple name" of name used to invoke this program
    (i.e., strip off directory-name prefix, if any).
    */
    argv[0] = sname(argv[0]);

    /*
    Invoke actual SCCS command, passing arguments.
    */
    sprintf(cmdstr, "/usr/bin/%s", argv[0]);
    execv(cmdstr, argv);
}
```

NOTES

Table of Contents

Using C on HP9000 Series 500 Computers

Introduction.....	1
Data Types and Manipulations	1
Data Type Sizes.....	1
Char Data Type.....	1
Register Data Type	1
Integer Overflow.....	1
Division by Zero	2
Identifiers	2
Shift Operators	2
Bit Fields	2
Code/Data Limitations	2
Portability Considerations	3

Using C on HP9000 Series 500 Computers

Introduction

The purpose of this article is to describe the machine dependent features of the C programming language as it is implemented on the HP 9000 Series 500 computers. No attempt is made here to fully describe C. When applicable, page numbers are given that reference pages in the Kernighan and Ritchie text, *The C Programming Language*, which are related to the discussion.

Data Types and Manipulations

Data Type Sizes

The following table gives the sizes and alignment requirements of the six data types implemented in C (page 34):

Type	Size	Alignment Requirements
char	8 bits	byte boundary
short	16 bits	half word
int	32 bits	full word
long	32 bits	full word
float	32 bits	full word
double	64 bits	full word

Char Data Type

The **char** data type is treated as signed by default. This implies that, if a **char** is assigned to an **int**, sign extension will take place (page 40).

Register Data Type

Because the Series 500 computers are stack machines, declaring a variable to be **register** is ignored, and is treated as a no-op (page 81).

Integer Overflow

Integer overflow does *not* generate an error by default (page 185).

Division by Zero

Whenever division by zero occurs, you get the (somewhat misleading) error message "Floating exception" at run-time.

Identifiers

Internal identifiers have 16 significant characters. External identifiers have 15 significant characters (page 179).

Shift Operators

An arithmetic shift is performed if the left operand is signed. If the left operand is unsigned, a logical shift is performed (page 45). (Remember that integer constants are treated as signed unless cast to unsigned.)

Bit Fields

Bit fields are assigned left to right, and are treated as unsigned (page 138).

Code/Data Limitations

The following limitations exist on the Series 500 computers:

- a maximum of 2^{19} bytes of local variables in any procedure;
- a maximum of 2^{19} bytes of parameters in any function call;
- any branch instruction generated by a procedure must be within 2^{18} bytes of its target;
- structure functions cannot return a structure bigger than 2^{24} bytes.

If you violate any of the above limits, you get the message "impossible reach" from the assembly step of *cc*. Other limitations are:

- a maximum of 255 procedures in any single compilation (i.e. any single ".c" file and everything it **#includes**). If you exceed this, you get "proctable overflow" from the assembler;
- a maximum of 32 767 lines of assembly code generated by *cc*. If you exceed this, you get "too many lines" from the assembler. To work around this, break your program up into smaller pieces;
- a maximum of 2^{19} bytes of global scalar data (includes all global scalar variables, all static scalar variables, all global and static structures, and 4 bytes for each global or static array). If you exceed this, you get "byte offset too large" from the linker, *ld*.

When compiling with *cc*, you can recognize assembler errors by the fact that they make reference to a file called */tmp/ctm3x*, where *x* is a single letter. Also, you can use the *-v* option to watch the compilation process, and note where the error occurs.

Portability Considerations

The following list should be kept in mind when transporting C code to the Series 500 computers from other machines:

the Series 500 computers do *not* swap bytes;

dereferencing a null pointer for a read or write operation generates a run-time error. On some other machines, dereferencing a null pointer for a read operation returns zero;

beware of attempts to use absolute addressing. The use of hard-coded addresses is not likely to work on any machine to which you want to port code;

even though the stack grows toward higher memory addresses, parameters are stacked toward decreasing addresses. Thus, if you want to use a pointer to step through a variable length parameter list, you must *decrement* the pointer.

Table of Contents

Lint C Program Checker

Introduction.....	1
Error Detection.....	1
Problem Detection.....	2
Problem Code: Unused Variables and Functions.....	2
Suppressing Lint.....	3
Problem Code: Set/Used Information.....	3
Problem Code: Unreachable Code.....	4
Suppressing Lint.....	4
Problem Code: Function Value.....	4
Problem Code: Type Matching.....	5
Suppressing Lint.....	6
Problem Code: Portability.....	6
Suppressing Lint.....	7
Problem Code: Strange Constructions.....	7
Suppressing Lint.....	8
Problem Code: Obsolete Constructions.....	8
How to Use Lint.....	9
Directives.....	9
Option List.....	10



Lint

C Program Checker

Introduction

Lint is a program checker and verifier for C source code. Its main purpose is to supply the programmer with warning messages about problems with the source code's style, efficiency, portability, and consistency. Once the C code passes through the compiler with no errors, *lint* can be used to locate areas, undetected by the compiler, that may require corrections.

Error messages and *lint* warnings are sent to the standard error file (the terminal by default). Once the code errors are corrected, the C source file(s) should be run through the C compiler to produce the necessary object code.

Error Detection

Lint can detect all of the code errors that the C compiler detects. An example of an error message would be:

```
illegal initialization
```

These errors must be corrected before the compiler can be used to produce object code.

Although *lint* can be used for error detection, it cannot recover from all of the code errors it finds. If *lint* encounters an error that it can not recover from, it sends the message:

```
cannot recover from earlier errors – goodbye!
```

and then terminates.

Lint limits the number of code errors that it detects to 30. Once 30 errors have been found in the source file(s), any additional error causes the message:

```
too many errors
```

to be sent to the standard error file, and *lint* terminates. Because of this limitation and *lint*'s inability to recover from some errors, the compiler should be used for error detection. Once the error-causing code has been corrected, *lint* can be used on the source code for finding some of its inefficiencies and bugs.

Problem Detection

The main purpose of *lint* is to find problem areas in C source code. The detected code may not be considered an error by the C compiler; it can be converted into object code. However, *lint* considers the code to be inefficient, nonportable, bad style, or a possible bug.

Comments about problems that are local to a function are produced when they are detected. They have the form:

```
warning: <message text>
```

Information about external functions and variables is collected and analyzed after *lint* has processed the files handed to it. At that time, if a problem has been detected, it sends a warning message with the form:

```
<message text>
```

followed by a list of external names causing the message and the files where the problem occurred.

Code causing *lint* to issue a warning message should be analyzed to determine the source of the problem. Sometimes the programmer has a valid reason for writing the problem code. Usually, though, this is not the case. *Lint* can be very helpful in uncovering subtle programming errors.

Lint checks the source code for certain conditions, about which it issues warning messages. These can be grouped into the following categories:

1. variable or function is declared but not used;
2. variable is used before it is set;
3. portion of code is unreachable;
4. function values are used incorrectly;
5. type matching does not adhere strictly to C rules;
6. code has portability problems;
7. code construction is strange;
8. code construction is obsolete.

The code that you write may have constructions in it that *lint* objects to but that are necessary to its application. Warning messages about problem areas that you know about and do not plan to correct provide useless information and make helpful messages harder to find. There are two methods for suppressing warning messages from *lint* that you do not need to see. The use of *lint options* is one. The *lint* command can be called with any combination of its defined option set. Each option has *lint* ignore a different problem area. The other method is to insert *lint directives* into the source code. *Lint* directives are discussed later.

Problem Code: Unused Variables and Functions

Lint objects if source code declares a variable that is never used or defines a function that is never called. Unused variables and functions are considered bad style because their declarations clutter the code. They can also be the cause of a program bug if their use is essential.

An unused local variable can result in one of two *lint* warning messages. If a variable is defined to be static and is not used *lint* responds with:

```
warning: static variable <name> unused
```

Unused automatic variables cause the message:

```
warning: <name> unused in function <name>
```

A function or external variable that is unused causes the message:

```
name defined but never used
```

followed by the function or variable name and the file in which it was defined. *Lint* also looks at the special case where one of the parameters of a function is not used. The warning message is:

```
warning: argument unused in function: <arg_name> in <func_name>
```

If functions or external variables are declared but never used or defined *lint* responds with

```
name declared but never used or defined
```

followed by a list of variable and function names and the names of files where they were declared.

Suppressing Lint

Sometimes it is necessary to have unused function parameters to support consistent interfaces between functions. The `-v` option can be used with *lint* to have warnings about unused parameters suppressed. However, the `-v` option does not suppress comments when parameters are defined as register variables. Unused register variables result in an inefficient use of the computer's resources, since quick-access hardware is often allocated for their storage.

If *lint* is run on a file which is linked with other files at compile time, many external variables and functions can be defined but not used, as well used but not defined. If there is no guarantee that the definition of an external object is always seen before the object is used, it is declared **extern**. The `-u` option can be used to stop complaints about all external objects, whether or not they are declared **extern**. If you want to inhibit complaints about only the **extern** declared functions and variables, use the `-x` option.

Problem Code: Set/Used Information

A probable bug exists in a program if a variable's value is used before it is assigned. Although *lint* attempts to detect occurrences of this, it takes into account only the physical location of the code. If code using a static or external variable is located before the variable is given a value the message sent is:

```
warning: <name> may be used before set
```


Since static and external variables are always initialized to zero this may not point out a program bug. *Lint* also objects if automatic variables are set in a function but not used. The message given is:

```
warning: <name> set but not used in function
```

Problem Code: Unreachable Code

Lint checks for three types of unreachable code. Any statement following a **goto**, **break**, **continue**, or **return** statement must either be labeled or reside in an outer block for *lint* to consider it reachable. If neither is the case, *lint* responds with:

```
warning: statement not reached
```

The same message is given if *lint* finds an infinite loop. It only checks for the infinite loop cases of **while(1)** and **for(;;)**. The third item that *lint* looks for is a loop that cannot be entered from the top. If one is found then the message sent is:

```
warning: loop not entered from top
```

Lint's detection of unreachable code is by no means perfect. Warning messages can be sent about valid code. It can also overlook commenting on code that cannot be reached. An example of this is the fact that *lint* does not know if a called function ever returns to the calling function (e.g. **exit**). *Lint* does not identify code following such a function call as being unreachable.

Suppressing Lint

Programs that are generated by *yacc* or *lex* can have many unreachable **break** statements. Normally, each one causes a complaint from *lint*. The **-b** option can be used to force *lint* to ignore unreachable **break** statements.

Problem Code: Function Value

The C compiler allows a function containing both the statement

```
return();
```

and the statement

```
return(expression);
```

to pass through without complaint. *Lint*, however, detects this inconsistency and responds with the message:

```
warning: function <name> has return(e); and return;
```

Problem Code: Type Matching

The C compiler does not strictly enforce the C language's type matching rules. At the loss of some type checking, the C compiler gains speed. An important role of *lint* is to enforce the type checking that the compiler neglects. It does this in four areas:

1. pointer types;
2. **long** and **int** type matching;
3. enumerations;
4. operations on structures and unions.

The types of pointers used in assignment, conditional, relational, and initialization statements must agree exactly. For example, the code:

```
int *p;
char *q;
.
.
.
p = q;
```

would cause *lint* to respond with the message:

```
warning: illegal pointer combination
```

Adding and subtracting integers and pointers are legal. Any other binary operation on them results in the message:

```
warning: illegal combination of pointer and integer: op <operator>
```

An example of code causing this message would be:

```
int s, *t;
.
.
.
t = s;
```

Assignments of long integer variables to integer variables are possible in the C language. However, on some machines the amount of storage supplied for the two types differs, and so the accuracy of a value could be lost in the conversion. *Lint* detects these assignments as possible program bugs. If a long integer is assigned to an integer, *lint* responds with:

```
warning: conversion from long may lose accuracy
```

Lint checks enumerations to see that variables or members are all of one type. Also, the only enumeration operations it allows are assignment, initialization, equality, and inequality. If *lint* finds code breaking any of these guidelines, it sends the message:

warning: enumeration type clash, operator <operator>

Structure and union references are subject to more type checking by *lint* than by the C compiler. *Lint* requires that the left operand of \rightarrow be a pointer to a structure or a union. If it isn't a pointer, *lint*'s response is:

warning: struct/union or struct/union pointer required

The left operand of \cdot must be a structure or a union, which *lint* also indicates with the message above. The right operand of \rightarrow and \cdot must be a member of the structure or union implied by the left operand. If it isn't then *lint*'s message is:

warning: illegal member use <name>

where <name> is the right operand.

Suppressing Lint

You may have a legitimate reason for converting a long integer to an integer. *Lint*'s $-a$ option inhibits comments about these conversions.

Problem Code: Portability

Lint aids the programmer in writing portable code in five areas:

1. character comparisons;
2. pointer alignments;
3. uninitialized external variables;
4. length of external variables;
5. type casting.

Character representation varies on different machines. Characters may be implemented as signed values or as unsigned values. As a result, certain comparisons with characters give different results on different machines. The expression

$c < 0$

where c is defined as type character, is always true if characters are unsigned values. If, however, characters are signed values the expression could be either true or false. Where character comparisons could result in different values depending on the machine used, *lint* outputs the message:

warning: nonportable character comparison

Legal pointer assignments are determined by the alignment restrictions of the particular machine used. For example, one machine may allow double precision values to begin on any integer boundary, but another may restrict them to word boundaries. If integer and word boundaries are different, code containing an assignment of a double pointer to an integer pointer could cause problems. *Lint* attempts to detect where the effect of pointer assignments is machine dependent. The warning that it sends is:

warning: possible pointer alignment problem

Another machine dependent area is the treatment of uninitialized external variables. If two files both contain the declaration

```
int a;
```

either one word of storage is allocated or each occurrence receives its own word of storage, depending on the machine. If the files that *lint* is processing contain multiple definitions of the same uninitialized external variable, *lint* responds with:

warning: <name> redefinition hides earlier one

The amount of information about external symbols that is loaded depends on the machine being used: the number of characters saved and whether or not upper/lower case distinction is kept. *Lint* truncates all external symbols to six characters and allows only one case distinction. (It changes upper case characters to lower case.) This provides a worst-case analysis so that the uniqueness of an external symbol is not machine dependent.

The effectiveness of type casting in C programs can depend on the machine that is used. For this reason, *lint* ignores type casting code. All assignments that use it are subject to *lint*'s type checking (see *Problem Code: Type Matching*).

Suppressing Lint

The `-p` option stops comments about two types of portability problems:

1. pointer alignment problems,
2. multiple definitions of external variables.

Lint's objections to legal casts can also be suppressed. To do so, use its `-c` option.

Problem Code: Strange Constructions

A *strange construction* is code that *lint* considers to be bad style or a possible bug.

Lint looks for code that has no effect. An example is:

```
*p+ +;
```

where the `*` has no effect. The statement is equivalent to `"p+ +;"`. In cases like this the message:

warning: null effect

is sent.

The treatment of unsigned numbers as signed numbers in comparisons causes *lint* to report:

warning: degenerate unsigned comparison

The following code would produce such a message:

```
unsigned x;  
.  
.  
.  
if (x < 0) ...
```

Lint also objects if constants are treated as variables. If the boolean expression in a conditional has a set value due to constants, such as

```
if (1 != 0) ...
```

lint's response is:

warning: constant in conditional context

If the NOT operator is used on a constant value, the response is:

warning: constant argument to NOT

To avoid operator precedence confusion, *lint* encourages using parentheses in expressions by sending the message:

warning: precedence confusion possible; parenthesize!

Lint judges it bad style to redefine an outer block variable in an inner block. Variables with different functions should normally have different names. If variables are redefined, the message sent is:

warning: <name> redefinition hides earlier one

Suppressing Lint

To stop *lint*'s comments about strange constructions, use its **-h** option.

Problem Code: Obsolete Constructions

C contains two forms of old syntax which, through the evolution of the language, are now officially discouraged. One is a group of assignment operators. Previously acceptable `= +`, `= -`, `= *`, `= /`, `= %`, `= <<`, `= >>`, `= &`, `= ^`, and `= |` have been changed to `+ =`, `- =`, `* =`, `/ =`, `% =`, `<< =`, `>> =`, `& =`, `^ =`, and `| =`. If *lint* sees the older form, it responds with:

warning: old-fashioned assignment operator

The second syntax change deals with initialization. An older version of C allowed:

```
int a 0;
```

to initialize *a* to zero. Initialization now requires that an equals sign appear between the variable and the value it is to receive:

```
int a = 0;
```

Lint's response to the earlier version is:

```
warning: old-fashioned initialization: use =
```

How to Use Lint

To use *lint*, you must be logged into the HP-UX system and have a shell prompt on your screen. From here you can run *lint* on a single C source file:

```
$ lint filename.c
```

or on several source files which are to be linked together:

```
$ lint file1.c file2.c file3.c
```

The reappearance of your shell prompt after invoking *lint* tells you that *lint* has finished processing your files. If no messages were sent to your standard error file, *lint* found nothing wrong with your code.

Directives

The alternative to using options to suppress *lint*'s comments about problem areas is to use directives. Directives appear in the source code in the form of code comments. *Lint* recognizes five directives.

- `/*NOTREACHED*/` stops an unreachable code comment about the next line of code.
- `/*NOSTRICT*/` stops *lint* from strictly type checking the next expression.
- `/*ARGSUSED*/` stops a comment about any unused parameters for the following function.
- `/*VARARGSn*/` stops *lint* from reporting variable numbers of parameters in calls to a function. The function's declaration follows this comment. The first *n* parameters must be present in each call to the function; *lint* comments if they aren't. If `/*VARARGS*/` appears without the *n*, none of the parameters need be present.
- `/*LINTLIBRARY*/` must be placed at the beginning of a file. This directive tells *lint* that the file is a library file and to suppress comments about unused functions. *Lint* objects if other files redefine routines that are found there.

Option List

The following is a list of the options available when using *lint*:

- a suppress complaints about assignments of integers to longs and of longs to integers.
- b suppress complaints about unreachable break statements.
- c suppress complaints about legal casts. Without this option typecasting is ignored.
- h suppress complaints about legal but strange constructions (see *Problem Code: Strange Constructions*).
- n do not check the compatibility of code against any libraries (standard and portable *lint* libraries, directive-defined libraries).
- p suppress some portability checks (see *Problem Code: Portability*).
- u suppress complaints about externals (functions and variables) that are used but not defined, or that are defined but not used (see *Problem Code: Unused Variables and Functions, Problem Code: Set/Used Information*).
- v suppress complaints about unused function parameters. If a parameter is unused and is also declared as a register variable, the warning is not suppressed.
- x suppress complaints about unused variables with external declarations (see *Problem Code: Set/Used Information*).
- Dname[= def]
define the string *name* to *lint*, as if a **#define** control line were used. If no definition is given, then *name* is given the value 1. This option is also used by the C compiler.
- Uname
remove any initial definition of *name*, as if a **#undef** control line were used. This option is also used by the C compiler.
- ldir change the algorithm for searching for **#include** files whose names do not begin with "/". The *dir* directory is searched before the directories on the standard list. Thus, **#include** files whose names are enclosed in double quotes (" ") are searched for first in the directory of the source file, then in the directory specified by each **-I** option, and finally in the directories on the standard list. If a **#include** file's name is enclosed in angle brackets (<>), the source file's directory is not searched. This option is also used by the C compiler.

Table of Contents

Nroff/Troff User's Manual

Introduction.....	1
Usage.....	1
References.....	2
Summary and Index.....	3
Escape Sequences for Characters, Indicators, and Functions.....	6
Predefined General Number Registers.....	7
Predefined Read-Only Number Registers.....	7
Reference Manual.....	8
General Explanation.....	8
Form of input.....	8
Formatter and device resolution.....	8
Numerical parameter input.....	8
Numerical Expressions.....	9
Notation.....	9
Font and Character Size Control.....	9
Character set.....	9
Fonts.....	10
Character size.....	10
Page Control.....	11
Text Filling, Adjusting, and Centering.....	12
Filling and adjusting.....	12
Interrupted text.....	12
Vertical Spacing.....	13
Base-line spacing.....	13
Extra line-space.....	13
Blocks of vertical space.....	13
Line Length and Indenting.....	14
Macros, Strings, Diversion, and Position Traps.....	14
Macros and strings.....	14
Copy mode input interpretation.....	15
Arguments.....	15
Diversions.....	15
Traps.....	16
Number Registers.....	17
Tabs, Leaders, and Fields.....	18
Tabs and leaders.....	18
Fields.....	18
Input/Output Conventions and Character Translations.....	19
Input character translations.....	19
Ligatures.....	19
Backspacing, underlining, overstriking, etc.....	19
Control characters.....	20
Output translation.....	20
Transparent throughput.....	20
Comments and concealed newlines.....	20

Table of Contents (continued)

Local Horizontal and Vertical Motions, Width Function	20
Local Motions	20
Width Function	21
Mark horizontal place	21
Overstrike, Bracket, Line-drawing, and Zero-width Functions	21
Overstriking	21
Zero-width characters	21
Large Brackets	21
Line drawing	21
Hyphenation	22
Three Part Titles	23
Output Line Numbering	23
Conditional Acceptance of Input	24
Environment Switching	24
Insertions from the Standard Input	25
Input/Output File Switching	25
Miscellaneous	25
Output and Error Messages	26
Tutorial Examples	27
Introduction	27
Page Margins	27
Paragraphs and Headings	28
Multiple Column Output	28
Footnote Processing	29
The Last Page	30
Table 1 - Font Style Examples	31
Table 2 - Input Naming Conventions for Non-ASCII Characters	32
Summary of Changes to Nroff/Troff	34
Options	34
Old Requests	34
New Requests	34
New Predefined Number Registers	34

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System¹ that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	Effect
-olist	Print only pages whose page numbers appear in <i>list</i> , which consists of comma-separated numbers and number ranges. A number range has the form $N-M$ and means pages N through M ; a initial $-N$ means from the beginning to page N ; and a final $N-$ means from N to the end.
-nN	Number first generated page N .
-sN	Stop every N pages. NROFF will halt prior to every N pages (default $N=1$) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every N pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
-mname	Prepends the macro file <code>/usr/lib/tmac.name</code> to the input <i>files</i> .
-raN	Register a (one-character) is set to N .
-i	Read standard input after the input files are exhausted.
-q	Invoke the simultaneous input-output mode of the rd request.

NROFF Only

- Tname** Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).
- e** Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

- t** Direct output to the standard output instead of the phototypesetter.
- f** Refrain from feeding out paper and stopping phototypesetter at the end of the run.
- w** Wait until phototypesetter is available, if currently busy.
- b** TROFF will report whether the phototypesetter is busy or available. No text processing is done.
- a** Send a printable (ASCII) approximation of the results to the standard output.
- pN** Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elapsed time.
- g** Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

```
nroff -o4,8-10 -T300S -mabc file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN² (for NROFF and TROFF respectively), and the table-construction preprocessor TBL³. A reverse-line postprocessor COL⁴ is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK⁴ is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT⁴ is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

```
tbl files | eqn | troff -t options | tcat
```

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT⁴ can be used to send TROFF (**-g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

- [1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).
- [2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.
- [3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.
- [4] Internal on-line documentation, on UNIX.
- [5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

<i>Request Form</i>	<i>Initial Value*</i>	<i>If No Argument</i>	<i>Notes#</i>	<i>Explanation</i>
1. General Explanation				
2. Font and Character Size Control				
.ps $\pm N$	10 point	previous	E	Point size; also $\backslash s \pm N$.†
.ss N	12/36 em	ignored	E	Space-character size set to $N/36$ em.†
.cs FNM	off	-	P	Constant character space (width) mode (font F).†
.bd $F N$	off	-	P	Embolden font F by $N-1$ units.†
.bd S $F N$	off	-	P	Embolden Special Font when current font is F .†
.ft F	Roman	previous	E	Change to font $F = x, xx, \text{ or } 1-4$. Also $\backslash fx, \backslash f(xx, \backslash fN$.
.fp $N F$	R,I,B,S	ignored	-	Font named F mounted on physical position $1 \leq N \leq 4$.
3. Page Control				
.pl $\pm N$	11 in	11 in	v	Page length.
.bp $\pm N$	$N=1$	-	B‡,v	Eject current page; next page number N .
.pn $\pm N$	$N=1$	ignored	-	Next page number N .
.po $\pm N$	0; 26/27 in	previous	v	Page offset.
.ne N	-	$N=1 V$	D,v	Need N vertical space ($V =$ vertical spacing).
.mk R	none	internal	D	Mark current vertical place in register R .
.rt $\pm N$	none	internal	D,v	Return (<i>upward only</i>) to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad c	adj,both	adjust	E	Adjust output lines with mode c .
.na	adjust	-	E	No output line adjusting.
.ce N	off	$N=1$	B,E	Center following N input text lines.
5. Vertical Spacing				
.vs N	1/6in;12pts	previous	E,p	Vertical base line spacing (V).
.ls N	$N=1$	previous	E	Output $N-1$ V s after each text output line.
.sp N	-	$N=1 V$	B,v	Space vertical distance N in either direction.
.sv N	-	$N=1 V$	v	Save vertical distance N .
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll $\pm N$	6.5 in	previous	E,m	Line length.
.in $\pm N$	$N=0$	previous	B,E,m	Indent.
.ti $\pm N$	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de $xx yy$	-	$.yy=..$	-	Define or redefine macro xx ; end at call of yy .
.am $xx yy$	-	$.yy=..$	-	Append to a macro.
.ds $xx string$	-	ignored	-	Define a string xx containing $string$.
.as $xx string$	-	ignored	-	Append $string$ to string xx .

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of " " as control character (instead of ".") suppresses the break function.

Request Form	Initial Value	If No Argument	Notes	Explanation
.rm <i>xx</i>	-	ignored	-	Remove request, macro, or string.
.rn <i>xx yy</i>	-	ignored	-	Rename request, macro, or string <i>xx</i> to <i>yy</i> .
.di <i>xx</i>	-	end	D	Divert output to macro <i>xx</i> .
.da <i>xx</i>	-	end	D	Divert and append to <i>xx</i> .
.wh <i>N xx</i>	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch <i>xx N</i>	-	-	v	Change trap location.
.dt <i>N xx</i>	-	off	D,v	Set a diversion trap.
.it <i>N xx</i>	-	off	E	Set an input-line count trap.
.em <i>xx</i>	none	none	-	End macro is <i>xx</i> .
8. Number Registers				
.nr <i>R ± N M</i>	-	-	u	Define and set number register <i>R</i> ; auto-increment by <i>M</i> .
.af <i>R c</i>	arabic	-	-	Assign format to register <i>R</i> (<i>c</i> =1, i, I, a, A).
.rr <i>R</i>	-	-	-	Remove register <i>R</i> .
9. Tabs, Leaders, and Fields				
.ta <i>Nt ...</i>	0.8; 0.5in	none	E,m	Tab settings; <i>left</i> type, unless <i>t</i> =R(right), C(centered).
.tc <i>c</i>	none	none	E	Tab repetition character.
.lc <i>c</i>	.	none	E	Leader repetition character.
.fc <i>a b</i>	off	off	-	Set field delimiter <i>a</i> and pad character <i>b</i> .
10. Input and Output Conventions and Character Translations				
.ec <i>c</i>	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg <i>N</i>	-; on.	on	-	Ligature mode on if <i>N</i> >0.
.ul <i>N</i>	off	<i>N</i> =1	E	Underline (italicize in TROFF) <i>N</i> input lines.
.cu <i>N</i>	off	<i>N</i> =1	E	Continuous underline in NROFF; like ul in TROFF.
.uf <i>F</i>	Italic	Italic	-	Underline font set to <i>F</i> (to be switched to by ul).
.cc <i>c</i>	.	.	E	Set control character to <i>c</i> .
.c2 <i>c</i>	.	.	E	Set nobreak control character to <i>c</i> .
.tr <i>abcd....</i>	none	-	O	Translate <i>a</i> to <i>b</i> , etc. on output.
11. Local Horizontal and Vertical Motions, and the Width Function				
12. Overstrike, Bracket, Line-drawing, and Zero-width Functions				
13. Hyphenation.				
.nh	hyphenate	-	E	No hyphenation.
.hy <i>N</i>	hyphenate	hyphenate	E	Hyphenate; <i>N</i> = mode.
.hc <i>c</i>	\%	\%	E	Hyphenation indicator character <i>c</i> .
.hw <i>word1 ...</i>		ignored	-	Exception words.
14. Three Part Titles.				
.tl <i>'left'center'right'</i>		-	-	Three part title.
.pc <i>c</i>	%	off	-	Page number character.
.lt <i>± N</i>	6.5 in	previous	E,m	Length of title.
15. Output Line Numbering.				
.nm <i>± N M S I</i>		off	E	Number mode on or off, set parameters.
.nn <i>N</i>	-	<i>N</i> =1	E	Do not number next <i>N</i> lines.
16. Conditional Acceptance of Input				
.if <i>c anything</i>				If condition <i>c</i> true, accept <i>anything</i> as input, for multi-line use <code>\{anything\}</code> .

Request Form	Initial Value	If No Argument	Notes	Explanation
.if ! <i>c anything</i>		-	-	If condition <i>c</i> false, accept <i>anything</i> .
.if <i>N anything</i>		-	u	If expression <i>N</i> > 0, accept <i>anything</i> .
.if ! <i>N anything</i>		-	u	If expression <i>N</i> ≤ 0, accept <i>anything</i> .
.if ' <i>string1 string2</i> ' <i>anything</i>		-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
.if !' <i>string1 string2</i> ' <i>anything</i>		-	-	If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
.ie <i>c anything</i>		-	u	If portion of if-else; all above forms (like if).
.el <i>anything</i>		-	-	Else portion of if-else.
17. Environment Switching.				
.ev <i>N</i>	<i>N=0</i>	previous	-	Environment switched (<i>push down</i>).
18. Insertions from the Standard Input				
.rd <i>prompt</i>	-	<i>prompt=BEL</i>	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.
19. Input/Output File Switching				
.so <i>filename</i>		-	-	Switch source file (<i>push down</i>).
.nx <i>filename</i>		end-of-file	-	Next file.
.pi <i>program</i>		-	-	Pipe output to <i>program</i> (NROFF only).
20. Miscellaneous				
.mc <i>c N</i>	-	off	E,m	Set margin character <i>c</i> and separation <i>N</i> .
.tm <i>string</i>	-	newline	-	Print <i>string</i> on terminal (UNIX standard message output).
.ig <i>yy</i>	-	.yy=..	-	Ignore till call of <i>yy</i> .
.pm <i>t</i>	-	all	-	Print macro names and sizes; if <i>t</i> present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

- B** Request normally causes a break.
- D** Mode or relevant parameters associated with current diversion level.
- E** Relevant parameters are a part of the current environment.
- O** Must stay in effect until logical output.
- P** Mode must be still or again in effect at the time of physical output.
- v,p,m,u** Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ad 4	cc 10	ds 7	fc 9	ie 16	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	dt 7	fi 4	if 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	ec 10	fl 20	ig 20	lt 14	nn 15	pm 20	rs 5	ti 6	
as 7	cs 2	el 16	fp 2	in 6	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cu 10	em 7	ft 2	it 7	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	da 7	eo 10	hc 13	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	de 7	ev 17	hw 13	lg 10	ne 3	os 5	rd 18	ss 2	uf 10	
c2 10	di 7	ex 18	hy 13	li 10	nf 4	pc 14	rm 7	sv 5	ul 10	

Escape Sequences for Characters, Indicators, and Functions

<i>Section Reference</i>	<i>Escape Sequence</i>	<i>Meaning</i>
10.1	\\	\ (to prevent or delay the interpretation of \)
10.1	\e	Printable version of the <i>current</i> escape character.
2.1	\'	' (acute accent); equivalent to \(\aa
2.1	\`	` (grave accent); equivalent to \(\ga
2.1	\-	- Minus sign in the <i>current</i> font
7	\.	Period (dot) (see de)
11.1	\(space)	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	\	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.7	*	Beginning of comment
7.3	\\$N	Interpolate argument $1 \leq N \leq 9$
13	\%	Default optional hyphenation character
2.1	\(xx	Character named <i>xx</i>
7.1	*x, *(xx	Interpolate string <i>x</i> or <i>xx</i>
9.1	\a	Non-interpreted leader character
12.3	\b'abc...'	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\fx,\f(xx,\fN	Change to font named <i>x</i> or <i>xx</i> , or position <i>N</i>
11.1	\h'N'	Local horizontal motion; move right <i>N</i> (<i>negative left</i>)
11.3	\kx	Mark horizontal <i>input</i> place in register <i>x</i>
12.4	\l'Nc'	Horizontal line drawing function (optionally with <i>c</i>)
12.4	\L'Nc'	Vertical line drawing function (optionally with <i>c</i>)
8	\nx,\n(xx	Interpolate number register <i>x</i> or <i>xx</i>
12.1	\o'abc...'	Overstrike characters <i>a</i> , <i>b</i> , <i>c</i> , ...
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	\sN,\s±N	Point-size change function
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	\v'N'	Local vertical motion; move down <i>N</i> (<i>negative up</i>)
11.2	\w'string'	Interpolate width of <i>string</i>
5.2	\x'N'	Extra line-space function (<i>negative before, positive after</i>)
12.2	\zc	Print <i>c</i> with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
	\X	<i>X</i> , any character <i>not</i> listed above

The escape sequences \\, \., \", \\$, *, \a, \n, \t, and \(\newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
3	%	Current page number.
11.2	ct	Character type (set by <i>width</i> function).
7.4	dl	Width (maximum) of last completed diversion.
7.4	dn	Height (vertical size) of last completed diversion.
-	dw	Current day of the week (1-7).
-	dy	Current day of the month (1-31).
11.3	hp	Current horizontal place on <i>input</i> line.
15	ln	Output line number.
-	mo	Current month (1-12).
4.1	nl	Vertical position of last printed text base-line.
11.2	sb	Depth of string below base line (generated by <i>width</i> function).
11.2	st	Height of string above base line (generated by <i>width</i> function).
-	yr	Last two digits of current year.

Predefined Read-Only Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
7.3	.\$	Number of arguments available at the current macro level.
-	.A	Set to 1 in TROFF, if -a option used; always 1 in NROFF.
11.1	.H	Available horizontal resolution in basic units.
-	.T	Set to 1 in NROFF, if -T option used; always 0 in TROFF.
11.1	.V	Available vertical resolution in basic units.
5.2	.a	Post-line extra line-space most recently utilized using \x'N' .
-	.c	Number of <i>lines</i> read from current input file.
7.4	.d	Current vertical place in current diversion; equal to nl , if no diversion.
2.2	.f	Current font as physical quadrant (1-4).
4	.h	Text base-line high-water mark on current page or diversion.
6	.i	Current indent.
6	.l	Current line length.
4	.n	Length of text portion on previous output line.
3	.o	Current page offset.
3	.p	Current page length.
2.3	.s	Current point size.
7.5	.t	Distance to the next trap.
4.1	.u	Equal to 1 in fill mode and 0 in nofill mode.
5.1	.v	Current vertical line spacing.
11.2	.w	Width of previous character.
-	.x	Reserved version-dependent register.
-	.y	Reserved version-dependent register.
7.4	.z	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ` (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ` suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function `\nR` causes the interpolation of the contents of the *number register* *R* in place of the function; here *R* is either a single character name as in `\nx`, or left-parenthesis-introduced, two-character name as in `\n(xx)`.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions, of various typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the `-T` option (default Model 37 Teletype).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units	
		TROFF	NROFF
i	Inch	432	240
c	Centimeter	$432 \times 50 / 127$	$240 \times 50 / 127$
P	Pica = 1/6 inch	72	240/6
m	Em = <i>S</i> points	$6 \times S$	<i>C</i>
n	En = Em/2	$3 \times S$	<i>C, same as Em</i>
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	<i>V</i>	<i>V</i>
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as `->` (`→`) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions `ll`, `in`, `ti`, `ta`, `lt`, `po`, `mc`, `\h`, and `\l`; *V*s for the vertically-oriented requests and functions `pl`, `wh`, `ch`, `dt`, `sp`, `sv`, `ne`, `rt`, `\v`, `\x`, and `\L`; `p` for the `vs` request; and `u` for the requests `nr`, `if`, and `ie`. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator `u` may need to be appended to prevent an additional inappropriate default scaling.

The number, N , may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number N to generate the distance to the vertical or horizontal place N . For vertically-oriented requests and functions, | N becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the vertical place N . For *all* other requests and functions, | N becomes the distance from the current horizontal place on the *input* line to the horizontal place N . For example,

.sp |3.2c

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register x contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)/2u

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. $\pm N$ means that the argument may take the forms N , $+N$, or $-N$ and that the corresponding effect is to set the affected parameter to N , to increment it by N , or to decrement it by N respectively. Plain N means that an initial algebraic sign is *not* an increment indicator, but merely the sign of N . Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(\(xx where xx is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
'	acute accent	'	close quote
`	grave accent	'	open quote
—	minus	-	hyphen

The characters ' , ` , and — may be input by \' , ` , and \— respectively or by their names (Table II). The ASCII characters @ , # , " , ' , ` , < , > , \ , { , } , ~ , ^ , and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The

characters `'`, ```, and `_` print as themselves.

2.2. *Fonts*. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either `\fx`, `\f(xx)`, or `\fN` where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. TROFF can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register `.f`.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. *Character size*. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a `\sN` at the desired point to set the size to *N*, or a `\s±N` ($1 \leq N \leq 9$) to increment/decrement the size by *N*; `\s0` restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the `.s` register. NROFF ignores type size control.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes*</i>	<i>Explanation</i>
<code>.ps ±N</code>	10 point	previous	E	Point size set to $\pm N$. Alternatively imbed <code>\sN</code> or <code>\s±N</code> . Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence $+N, -N$ will work because the previous requested value is also remembered. Ignored in NROFF.
<code>.ss N</code>	12/36 em	ignored	E	Space-character size is set to $N/36$ ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
<code>.cs FNM</code>	off	-	P	Constant character space (width) mode is set on for font <i>F</i> (if mounted); the width of every character will be taken to be $N/36$ ems. If <i>M</i> is absent, the em is that of the character's point size; if <i>M</i> is given, the em is <i>M</i> -points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is <i>F</i> are also so treated. If <i>N</i> is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
<code>.bd FN</code>	off	-	P	The characters in font <i>F</i> will be artificially emboldened by printing each one twice, separated by $N-1$ basic units. A reasonable value for <i>N</i> is 3 when the character size is in the vicinity of 10 points. If <i>N</i> is missing the embolden mode is turned off. The column heads above were printed with <code>.bd I 3</code> . The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd <i>S FN</i>	off	-	P	The characters in the Special Font will be emboldened whenever the current font is <i>F</i> . This manual was printed with .bd S B 3 . The mode must be still or again in effect when the characters are physically printed.
.ft <i>F</i>	Roman	previous	E	Font changed to <i>F</i> . Alternatively, imbed <code>\fF</code> . The font name P is reserved to mean the previous font.
.fp <i>N F</i>	R,I,B,S	ignored	-	Font position. This is a statement that a font named <i>F</i> is mounted on position <i>N</i> (1-4). It is a fatal error if <i>F</i> is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (*N* from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	Initial Value	If No Argument	Notes	Explanation
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the .p register.
.bp $\pm N$	$N=1$	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request ns .
.pn $\pm N$	$N=1$	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A pn must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the % register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current <i>left margin</i> is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches. See §6. The current page offset is available in the .o register.
.ne <i>N</i>	-	$N=1$ V	D,v	Need <i>N</i> vertical space. If the distance, <i>D</i> , to the next trap position (see §7.5) is less than <i>N</i> , a forward vertical space of size <i>D</i> occurs, which will spring the trap. If there are no remaining traps on the page, <i>D</i> is the

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, D is the distance to the *diversion trap*, if any, or is very large.

.mk R	none	internal	D	Mark the <i>current</i> vertical place in an internal register (both associated with the current diversion level), or in register R , if given. See rt request.
.rt $\pm N$	none	internal	D,v	Return <i>upward only</i> to a marked vertical place in the current diversion. If $\pm N$ (w.r.t. current place) is given, the place is $\pm N$ from the top of the page or diversion or, if N is absent, to a place marked by a previous mk . Note that the sp request (§5.3) may be used in all cases instead of rt by spacing to the absolute place stored in an explicit register; e. g. using the sequence .mk Rsp $ \backslash n Ru$.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into an output text line until some word doesn't fit. An attempt is then made to hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the **ss** request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **-e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, or **!** is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using **tr** (§10.5).

4.2. Interrupted text. The copying of an input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a **\c**. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with **\c**; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	Initial Value	If No Argument	Notes	Explanation
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

.fi	fill on	-	B,E	Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are <i>neither</i> filled <i>nor</i> adjusted. Input text lines are copied directly to output lines <i>without regard</i> for the current line length.
.ad c	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator <i>c</i> is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for ad is not changed. Output line filling still occurs if fill mode is on.
.ce N	off	N=1	B,E	Center the next <i>N</i> input text lines within the current (line-length minus indent). If <i>N</i> =0, any residual count is cleared. A break occurs after each of the <i>N</i> input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing (*V*) between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function $\backslash\mathbf{x}'N'$ can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here \mathbf{x}'), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	Initial Value	If No Argument	Notes	Explanation
.vs N	1/6in;12pts	previous	E,p	Set vertical base-line spacing size <i>V</i> . Transient <i>extra</i> vertical space available with $\backslash\mathbf{x}'N'$ (see above).
.ls N	N=1	previous	E	<i>Line</i> spacing set to $\pm N$. <i>N</i> -1 <i>V</i> s (<i>blank lines</i>) are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line

				reached a trap position.	
.sp	<i>N</i>	-	$N=1 V$	B,v	Space vertically in <i>either</i> direction. If <i>N</i> is negative, the motion is <i>backward</i> (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see ns , and rs below).
.sv	<i>N</i>	-	$N=1 V$	v	Save a contiguous vertical block of size <i>N</i> . If the distance to the next trap is greater than <i>N</i> , <i>N</i> vertical space is output. No-space mode has <i>no</i> effect. If this distance is less than <i>N</i> , no vertical space is immediately output, but <i>N</i> is remembered for later output (see os). Subsequent sv requests will overwrite any still remembered <i>N</i> .
.os		-	-	-	Output saved vertical space. No-space mode has <i>no</i> effect. Used to finally output a block of vertical space requested by an earlier sv request.
.ns	space	-	-	D	No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests <i>without</i> a next page number. The no-space mode is turned off when a line of output occurs, or with rs .
.rs	space	-	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	-	B	Causes a break and output of a blank line exactly like sp 1 .

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ll $\pm N$	6.5 in	previous	E,m	Line length is set to $\pm N$. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches.
.in $\pm N$	$N=0$	previous	B,E,m	Indent is set to $\pm N$. The indent is prepended to each output line.
.ti $\pm N$	-	ignored	B,E,m	Temporary indent. The <i>next</i> output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a

control line beginning `.xx` will interpolate the contents of macro `xx`. The remainder of the line may contain up to nine *arguments*. The strings `x` and `xx` are interpolated at any desired point with `*x` and `*(xx` respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by `\n` are interpolated.
- Strings indicated by `*` are interpolated.
- Arguments indicated by `\$` are interpolated.
- Concealed newlines indicated by `\(newline)` are eliminated.
- Comments indicated by `\"` are eliminated.
- `\t` and `\a` are interpreted as ASCII horizontal tab and SOH respectively (§9).
- `\\` is interpreted as `\`.
- `\.` is interpreted as `"."`.

These interpretations can be suppressed by prepending a `\`. For example, since `\\` maps into a `\`, `\\n` will copy as `\n` which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with `\$N`, which interpolates the *N*th argument ($1 \leq N \leq 9$). If an invoked argument doesn't exist, a null string results. For example, the macro `xx` may be defined by

```
.de xx      \"begin definition
Today is \\$1 the \\$2.
..         \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that the `\$` was concealed in the definition with a prepended `\`. The number of currently available arguments is in the `.$` register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as an input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra `\`) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers `dn` and `dl` respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (`cs`) or emboldened (`bd`) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way

to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see it below.

Request Form	Initial Value	If No Argument	Notes	Explanation
.de <i>xx yy</i>	-	<i>.yy=..</i>	-	Define or redefine the macro <i>xx</i> . The contents of the macro begin on the next input line. Input lines are copied in <i>copy mode</i> until the definition is terminated by a line beginning with <i>.yy</i> , whereupon the macro <i>yy</i> is called. In the absence of <i>yy</i> , the definition is terminated by a line beginning with <i>..</i> . A macro may contain de requests provided the terminating macros differ or the contained definition terminator is concealed. <i>..</i> can be concealed as <i>\\.</i> which will copy as <i>\\.</i> and be reread as <i>..</i> .
.am <i>xx yy</i>	-	<i>.yy=..</i>	-	Append to macro (append version of de).
.ds <i>xx string</i>	-	ignored	-	Define a string <i>xx</i> containing <i>string</i> . Any initial double-quote in <i>string</i> is stripped off to permit initial blanks.
.as <i>xx string</i>	-	ignored	-	Append <i>string</i> to string <i>xx</i> (append version of ds).
.rm <i>xx</i>	-	ignored	-	Remove request, macro, or string. The name <i>xx</i> is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn <i>xx yy</i>	-	ignored	-	Rename request, macro, or string <i>xx</i> to <i>yy</i> . If <i>yy</i> exists, it is first removed.
.di <i>xx</i>	-	end	D	Divert output to macro <i>xx</i> . Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request di or da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

<code>.da xx</code>	-	end	D	Divert, appending to <i>xx</i> (append version of di).
<code>.wh N xx</code>	-	-	v	Install a trap to invoke <i>xx</i> at page position <i>N</i> ; a <i>negative N</i> will be interpreted with respect to the page <i>bottom</i> . Any macro previously planted at <i>N</i> is replaced by <i>xx</i> . A zero <i>N</i> refers to the <i>top</i> of a page. In the absence of <i>xx</i> , the first found trap at <i>N</i> , if any, is removed.
<code>.ch xx N</code>	-	-	v	Change the trap position for macro <i>xx</i> to be <i>N</i> . In the absence of <i>N</i> , the trap, if any, is removed.
<code>.dt N xx</code>	-	off	D,v	Install a diversion trap at position <i>N</i> in the <i>current</i> diversion to invoke macro <i>xx</i> . Another dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
<code>.it N xx</code>	-	off	E	Set an input-line-count trap to invoke the macro <i>xx</i> after <i>N</i> lines of <i>text</i> input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
<code>.em xx</code>	none	none	-	The macro <i>xx</i> will be invoked when all input has ended. The effect is the same as if the contents of <i>xx</i> had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
<code>\nx</code>	none	<i>N</i>
<code>\n(xx</code>	none	<i>N</i>
<code>\n+x</code>	<i>x</i> incremented by <i>M</i>	<i>N+M</i>
<code>\n-x</code>	<i>x</i> decremented by <i>M</i>	<i>N-M</i>
<code>\n+(xx</code>	<i>xx</i> incremented by <i>M</i>	<i>N+M</i>
<code>\n-(xx</code>	<i>xx</i> decremented by <i>M</i>	<i>N-M</i>

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.nr R ± N M</code>	-	-	u	The number register <i>R</i> is assigned the value $\pm N$ with respect to the previous value, if any. The increment for auto-incrementing is set to <i>M</i> .

.af R c arabic - Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

.rr R - ignored - Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with **ta**. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and *W* is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of <i>next-string</i>
Left	<i>D</i>	Following <i>D</i>
Right	<i>D</i> - <i>W</i>	Right adjusted within <i>D</i>
Centered	<i>D</i> - <i>W</i> /2	Centered on right end of <i>D</i>

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. **\t** and **\a** always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is **#** and the padding indicator is **^**, **#^xxx^right#** specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	Initial Value	If No Argument	Notes	Explanation
.ta <i>Nt</i> ...	0.8; 0.5in	none	E,m	Set tab stops and types. <i>t</i> = R , right adjusting; <i>t</i> = C , centering; <i>t</i> absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
.tc <i>c</i>	none	none	E	The tab repetition character becomes <i>c</i> , or is removed specifying motion.
.lc <i>c</i>	.	none	E	The leader repetition character becomes <i>c</i> , or is removed specifying motion.
.fc <i>a b</i>	off	off	-	The field delimiter is set to <i>a</i> ; the padding indicator is set to the <i>space</i> character or to <i>b</i> , if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All others are ignored.*

The *escape* character **** introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. **** should not be confused with the ASCII control character ESC of the same name. The escape character **** can be input with the sequence ****. The escape character can be changed with **ec**, and all that has been said about the default **** becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	Initial Value	If No Argument	Notes	Explanation
.ec <i>c</i>	\	\	-	Set escape character to \ , or to <i>c</i> , if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in NROFF) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	Initial Value	If No Argument	Notes	Explanation
.lg <i>N</i>	off; on	on	-	Ligature mode is turned on if <i>N</i> is absent or non-zero, and turned off if <i>N</i> =0. If <i>N</i> =2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in <i>copy mode</i> . No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\fF**, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	Initial Value	If No Argument	Notes	Explanation
.ul <i>N</i>	off	<i>N=1</i>	E	Underline in NROFF (italicize in TROFF) the next <i>N</i> input text lines. Actually, switch to <i>underline</i> font, saving the current font for later restoration; <i>other</i> font changes within the span of a ul will take effect, but the restoration will undo the last change. Output generated by tl (§14) is affected by the font change, but does <i>not</i> decrement <i>N</i> . If <i>N</i> >1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu <i>N</i>	off	<i>N=1</i>	E	A variant of ul that causes <i>every</i> character to be underlined in NROFF. Identical to ul in TROFF.
.uf <i>F</i>	Italic	Italic	-	Underline font set to <i>F</i> . In NROFF, <i>F</i> may <i>not</i> be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character **.** and the *no-break* control character **'** may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	Initial Value	If No Argument	Notes	Explanation
.cc <i>c</i>	.	.	E	The basic control character is set to <i>c</i> , or reset to ".".
.c2 <i>c</i>	'	'	E	The <i>nobreak</i> control character is set to <i>c</i> , or reset to "'".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	Initial Value	If No Argument	Notes	Explanation
.tr <i>abcd...</i>	none	-	O	Translate <i>a</i> into <i>b</i> , <i>c</i> into <i>d</i> , etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from <i>input</i> to <i>output</i> time.

10.6. Transparent throughput. An input line beginning with a **\!** is read in *copy mode* and *transparently* output (without the initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape ****. The sequence **\(newline)** is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with ****. The newline at the end of a comment cannot be concealed. A line beginning with **** will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with ****.

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v'*N*'** and **\h'*N*'** can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in		Horizontal Local Motion	Effect in	
	TROFF	NROFF		TROFF	NROFF
<code>\v'N'</code>	Move distance <i>N</i>		<code>\h'N'</code> <code>\(space)</code> <code>\0</code>	Move distance <i>N</i> Unpaddable space-size space Digit-size space	
<code>\u</code> <code>\d</code> <code>\r</code>	½ em up ½ em down 1 em up	½ line up ½ line down 1 line up	<code>\ </code> <code>\^</code>	1/6 em space 1/12 em space	ignored ignored

As an example, E^2 could be generated by the sequence `E\s-2\v'-0.4m'2\v'0.4m'\s+2`; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function `\w'string'` generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, `.ti-\w'1. 'u` could be used to temporarily indent leftward a distance equal to the size of the string "1. ".

The width function also sets three number registers. The registers `st` and `sb` are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is `\n(stu-\n(sbu)`. In TROFF the number register `ct` is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like `e`); 1 means that at least one character has a descender (like `y`); 2 means that at least one character is tall (like `H`); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence `\kx` will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction `\kx word\h'|nxu+2u' word` will embolden *word* by backing up to almost its beginning and overprinting it, resulting in **word**.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function `\o'string'`. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, `\o'e''` produces \acute{e} , and `\o'(mo)\(sl'` produces £ .

12.2. Zero-width characters. The function `\zc` will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, `\z(ci)\(pl` will produce \oplus , and `\(br)\z(rn)\(ul)\(br` will produce the smallest possible constructed box \square .

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (`{` `}` `[[` `]]` `[[` `]]`) that can be combined into various bracket styles. The function `\b'string'` may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, `\b'\(lc)\(lf'E'\)\b'\(rc)\(rf'\x'-0.5m'\x'0.5m'` produces $\left[E \right]$.

12.4. Line drawing. The function `\l'Nc'` will draw a string of repeated *c*'s towards the right for a distance *N*. (`\l` is `\(lower case L)`). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a `\&`. If *c* is not specified, the `_` (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule `_`, underrule `_`, and root-en `¯`, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'1'|0\ul'
..
```

or one to draw a box around a string

```
.de bx
\ (br\|\\$1\| (br\l'|0\ (rn'\l'|0\ (ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function $\backslash L' N c'$ will draw a vertical line consisting of the (optional) character c stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | ($\backslash (br)$); the other suitable character is the *bold vertical* | ($\backslash (bv)$). The line is begun without any initial motion relative to the current base line. A positive N specifies a line drawn downward and a negative N specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the 1/2-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \ "compensate for next automatic base-line spacing
.nf        \ "avoid possibly overflowing word buffer
\h'-.5n\L'\|\\nau-1'l'\n (.lu+1n\ (ul'\L'-|\\nau+1'l'|0u-.5n\ (ul'  \ "draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register a (e. g. using $.mk a$) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes ($\backslash (em)$), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hyN	on, $N=1$	on, $N=1$	E	Automatic hyphenation is turned on for $N \geq 1$, or off for $N=0$. If $N=2$, <i>last</i> lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. $N=14$ will invoke all three restrictions.
.hc c	\%	\%	E	Hyphenation indicator character is set to c or to the default \%. The indicator does not appear in the output.
.hw word1 ...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal s are

implied; i. e. *dig-it* implies *dig-its*. This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.tl 'left' center' right'		-	-	The strings <i>left</i> , <i>center</i> , and <i>right</i> are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter.
.pc <i>c</i>	%	off	-	The page number character is set to <i>c</i> , or removed. The page-number register remains %.
.lt $\pm N$	6.5 in	previous	E,m	Length of title set to $\pm N$. The line-length and the title-length are <i>independent</i> . Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nm $\pm N M S I$		off	E	Line number mode. If $\pm N$ is given, line numbering is turned on, and the next output line numbered is numbered $\pm N$. Default values are $M=1$, $S=1$, and $I=0$. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register ln .
.nn <i>N</i>	-	$N=1$	E	The next <i>N</i> text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with $M=3$: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by `\w'0000'u`) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with $M=5$, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, ! signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

Request Form	Initial Value	If No Argument	Notes	Explanation
<code>.if c anything</code>		-	-	If condition <i>c</i> true, accept <i>anything</i> as input; in multi-line case use <code>\{anything\}</code> .
<code>.if !c anything</code>		-	-	If condition <i>c</i> false, accept <i>anything</i> .
<code>.if N anything</code>		-	u	If expression $N > 0$, accept <i>anything</i> .
<code>.if !N anything</code>		-	u	If expression $N \leq 0$, accept <i>anything</i> .
<code>.if 'string1' string2' anything</code>		-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
<code>.if !'string1' string2' anything</code>		-	-	If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
<code>.ie c anything</code>		-	u	If portion of if-else; all above forms (like if).
<code>.el anything</code>		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter `{` and the last line must end with a right delimiter `}`.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp 0.5i
.tl 'Page %'''\
'sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting **E** in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters,

number registers, and macro and string definitions. All environments are initialized with default parameter values.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ev N</code>	$N=0$	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment <i>must</i> be done with <code>.ev</code> rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with `rd`, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.rd prompt</code>	-	<code>prompt=BEL</code>	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, <i>prompt</i> (or a BEL) is written onto the user's terminal. <code>rd</code> behaves like a macro, and arguments may be placed after <i>prompt</i> .
<code>.ex</code>	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option `-q` will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using `nx` (§19); the process would ultimately be ended by an `ex` in the insertion file.

19. Input/Output File Switching

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.so filename</code>		-	-	Switch source file. The top input (file reading) level is switched to <i>filename</i> . The effect of an <code>so</code> encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. <code>so</code> 's may be nested.
<code>.nx filename</code>		end-of-file	-	Next file is <i>filename</i> . The current file is considered ended, and the input is immediately switched to <i>filename</i> .
<code>.pi program</code>		-	-	Pipe output to <i>program</i> (NROFF only). This request must occur <i>before</i> any printing occurs. No arguments are transmitted to <i>program</i> .

20. Miscellaneous

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.mc c N</code>	-	off	E,m	Specifies that a <i>margin</i> character <i>c</i> appear a distance <i>N</i> to the right of the right margin after each non-empty text line (except those produced by <code>tl</code>). If the output line is too-long (as can happen in <code>nofill</code> mode) the character will

be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.

.tm	<i>string</i>	-	newline	-	After skipping initial blanks, <i>string</i> (rest of the line) is read in <i>copy mode</i> and written on the user's terminal.
.ig	<i>yy</i>	-	.yy=.	-	Ignore input lines. ig behaves exactly like de (§7) except that the input is discarded. The input is read in <i>copy mode</i> , and any auto-incremented registers will be affected.
.pm	<i>t</i>	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if <i>t</i> is given, only the total of the sizes is printed. The sizes is given in <i>blocks</i> of 128 characters.
.fl		-		B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in NROFF and a ■ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd      \"define header
'sp 1i
..          \"end definition
.de fo      \"define footer
'bp
..          \"end definition
.wh 0 hd
.wh -1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the

initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd      \"header
.if t .tl '\(rn''\ (rn' \"troff cut mark
.if \\n%>1 \\{
'sp |0.5i-1  \"tl base at 0.5i
.tl \"- % -\" \"centered page number
.ps        \"restore size
.ft        \"restore font
.vs \}     \"restore vs
'sp |1.0i  \"space to 1.0i
.ns        \"turn on no-space mode
..
.de fo      \"footer
.ps 10     \"set footer/header size
.ft R      \"set font
.vs 12p    \"set base-line spacing
.if \\n%=1 \\{
'sp |\\n(.pu-0.5i-1 \"tl base 0.5i up
.tl \"- % -\" \\} \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s  \\"current size
.ps
.nr s2 \\n(.s  \\"previous size
. ---        \\"rest of footer
..
.de hd
. ---        \\"header stuff
.ps \\n(s2    \\"restore previous size
.ps \\n(s1    \\"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn      \\"bottom number
.tl ""- % -" \\"centered page number
..
.wh -0.5i-1v bn \\"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg      \\"paragraph
.br         \\"break
.ft R       \\"force font,
.ps 10      \\"size,
.vs 12p     \\"spacing,
.in 0       \\"and indent
.sp 0.4     \\"prespace
.ne 1+\\n(.Vu \\"want more than 1 line
.ti 0.2i    \\"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once.

The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc      \\"section
. ---      \\"force font, etc.
.sp 0.4     \\"prespace
.ne 2.4+\\n(.Vu \\"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1   \\"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp      \\"labeled paragraph
.pg
.in 0.5i    \\"paragraph indent
.ta 0.2i 0.5i \\"label, paragraph
.ti 0
\\t\\$1\\t\\c \\"flow into paragraph
..
```

The intended usage is **.lp label**; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\c** so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```

.de hd      \"header
. ---
.nr cl 0 1  \"init column count
.mk        \"mark top of text
..
.de fo      \"footer
.ie \\n + (cl < 2 \\{
.po + 3.4i  \"next column; 3.1 + 0.3
.rt        \"back to mark
.ns \\}    \"no-space mode
.el \\{
.po \\nMu   \"restore left margin
. ---
'bp \\}
..
.ll 3.1i    \"column width
.nr M \\n(o \"save left margin

```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another `.mk` would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial `.fn` and a terminal `.ef`:

```

.fn
  Footnote text and control lines...
.ef

```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```

.de hd      \"header
. ---
.nr x 0 1   \"init footnote count
.nr y 0-\\nb \"current footer place
.ch fo -\\nbu \"reset footer trap
.if \\n(dn .fz \"leftover footnote
..
.de fo      \"footer
.nr dn 0    \"zero last diversion size
.if \\nx \\{
.ev 1      \"expand footnotes in ev1
.nf        \"retain vertical size
.FN        \"footnotes
.rm FN     \"delete it
.if "\\n(.z"fy" .di \"end overflow diversion
.nr x 0    \"disable fx

```

```

.ev \\}     \"pop environment
. ---
'bp
..
.de fx      \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn      \"start footnote
.da FN     \"divert (append) footnote
.ev 1      \"in environment 1
.if \\n + x = 1 .fs \"if first, include separator
.fi        \"fill mode
..
.de ef      \"end footnote
.br        \"finish output
.nr z \\n(.v \"save spacing
.ev        \"pop ev
.di        \"end diversion
.nr y -\\n(dn \"new footer position,
.if \\nx = 1 .nr y - (\\n(.v - \\nz) \\
           \"uncertainty correction
.ch fo \\nyu \"y is negative
.if (\\n(nl + 1v) > (\\n(.p + \\ny) \\
.ch fo \\n(nlu + 1v \"it didn't fit
..
.de fs      \"separator
\\' 1i'     \"1 inch rule
.br
..
.de fz      \"get leftover footnote
.fn        \"retain vertical size
.nf        \"where fx put it
.ef
..
.nr b 1.0i  \"bottom margin size
.wh 0 hd    \"header trap
.wh 12i fo  \"footer trap, temp position
.wh -\\nbu fx \"fx at footer position
.ch fo -\\nbu \"conceal fx with fo

```

The header `hd` initializes a footnote count register `x`, and sets both the current footer trap position register `y` and the footer trap itself to a nominal position specified in register `b`. In addition, if the register `dn` indicates a leftover footnote, `fz` is invoked to reprocess it. The footnote start macro `fn` begins a diversion (append) in environment 1, and increments the count `x`; if the count is one, the footnote separator `fs` is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro `ef` restores the previous environment and ends the diversion after saving the spacing size in register `z`. `y` is then decremented by the size of the

footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redirect the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, **NROFF** and **TROFF** invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en      \*end-macro
\c
`bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table II

**Input Naming Conventions for ' , ` , and –
and for Non-ASCII Special Characters**

Non-ASCII characters and *minus* on the standard fonts.

<i>Input Character</i>			<i>Input Character</i>		
<i>Char</i>	<i>Name</i>	<i>Name</i>	<i>Char</i>	<i>Name</i>	<i>Name</i>
'	'	close quote	fi	\(fi	fi
`	`	open quote	fl	\(fl	fl
–	\(em	3/4 Em dash	ff	\(ff	ff
-	–	hyphen or	ffi	\(Fi	ffi
-	\(hy	hyphen	ffl	\(Fl	ffl
–	\(–	current font minus	°	\(de	degree
•	\(bu	bullet	†	\(dg	dagger
□	\(sq	square	'	\(fm	foot mark
–	\(ru	rule	¢	\(ct	cent sign
¼	\(14	1/4	®	\(rg	registered
½	\(12	1/2	©	\(co	copyright
¾	\(34	3/4			

Non-ASCII characters and ' , ` , _ , + , – , = , and * on the special font.

The ASCII characters @ , # , " , ' , ` , < , > , \ , { , } , ~ , ^ , and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

<i>Input Character</i>			<i>Input Character</i>		
<i>Char</i>	<i>Name</i>	<i>Name</i>	<i>Char</i>	<i>Name</i>	<i>Name</i>
+	\(pl	math plus	κ	\(*k	kappa
–	\(mi	math minus	λ	\(*l	lambda
=	\(eq	math equals	μ	\(*m	mu
*	\(**	math star	ν	\(*n	nu
§	\(sc	section	ξ	\(*c	xi
'	\(aa	acute accent	ο	\(*o	omicron
`	\(ga	grave accent	π	\(*p	pi
–	\(ul	underrule	ρ	\(*r	rho
/	\(sl	slash (matching backslash)	σ	\(*s	sigma
α	\(*a	alpha	ς	\(ts	terminal sigma
β	\(*b	beta	τ	\(*t	tau
γ	\(*g	gamma	υ	\(*u	upsilon
δ	\(*d	delta	φ	\(*f	phi
ε	\(*e	epsilon	χ	\(*x	chi
ζ	\(*z	zeta	ψ	\(*q	psi
η	\(*y	eta	ω	\(*w	omega
θ	\(*h	theta	Α	\(*A	Alpha†
ι	\(*i	iota	Β	\(*B	Beta†

<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
Γ	<code>\(*G</code>	Gamma
Δ	<code>\(*D</code>	Delta
ϵ	<code>\(*E</code>	Epsilon†
ζ	<code>\(*Z</code>	Zeta†
η	<code>\(*Y</code>	Eta†
θ	<code>\(*H</code>	Theta
ι	<code>\(*I</code>	Iota†
κ	<code>\(*K</code>	Kappa†
λ	<code>\(*L</code>	Lambda
μ	<code>\(*M</code>	Mu†
ν	<code>\(*N</code>	Nu†
ξ	<code>\(*C</code>	Xi
\omicron	<code>\(*O</code>	Omicron†
π	<code>\(*P</code>	Pi
ρ	<code>\(*R</code>	Rho†
σ	<code>\(*S</code>	Sigma
τ	<code>\(*T</code>	Tau†
υ	<code>\(*U</code>	Upsilon
ϕ	<code>\(*F</code>	Phi
χ	<code>\(*X</code>	Chi†
ψ	<code>\(*Q</code>	Psi
ω	<code>\(*W</code>	Omega
$\sqrt{\quad}$	<code>\(sr</code>	square root
$\sqrt[n]{\quad}$	<code>\(rn</code>	root en extender
\geq	<code>\(> =</code>	$> =$
\leq	<code>\(< =</code>	$< =$
\equiv	<code>\(= =</code>	identically equal
\approx	<code>\(ˆ =</code>	approx =
\sim	<code>\(ap</code>	approximates
\neq	<code>\(! =</code>	not equal
\rightarrow	<code>\(-></code>	right arrow
\leftarrow	<code>\(<-</code>	left arrow
\uparrow	<code>\(ua</code>	up arrow
\downarrow	<code>\(da</code>	down arrow
\times	<code>\(mu</code>	multiply
\div	<code>\(di</code>	divide
\pm	<code>\(+-</code>	plus-minus
\cup	<code>\(cu</code>	cup (union)
\cap	<code>\(ca</code>	cap (intersection)
\subset	<code>\(sb</code>	subset of
\supset	<code>\(sp</code>	superset of
\subsetneq	<code>\(ib</code>	improper subset
\supsetneq	<code>\(ip</code>	improper superset
∞	<code>\(if</code>	infinity
∂	<code>\(pd</code>	partial derivative
∇	<code>\(gr</code>	gradient
\neg	<code>\(no</code>	not
\int	<code>\(is</code>	integral sign
\propto	<code>\(pt</code>	proportional to
\emptyset	<code>\(es</code>	empty set
\in	<code>\(mo</code>	member of

<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
	<code>\(br</code>	box vertical rule
‡	<code>\(dd</code>	double dagger
✋	<code>\(rh</code>	right hand
✋	<code>\(lh</code>	left hand
Ⓜ	<code>\(bs</code>	Bell System logo
	<code>\(or</code>	or
○	<code>\(ci</code>	circle
{	<code>\(lt</code>	left top of big curly bracket
{	<code>\(lb</code>	left bottom
}	<code>\(rt</code>	right top
}	<code>\(rb</code>	right bot
{	<code>\(lk</code>	left center of big curly bracket
}	<code>\(rk</code>	right center of big curly bracket
	<code>\(bv</code>	bold vertical
	<code>\(lf</code>	left floor (left bottom of big square bracket)
	<code>\(rf</code>	right floor (right bottom)
	<code>\(lc</code>	left ceiling (left top)
	<code>\(rc</code>	right ceiling (right top)

Summary of Changes to N/TROFF Since October 1976 Manual

Options

- h (Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.
- z Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

- .ad c The adjustment type indicator "c" may now also be a number previously obtained from the "j" register (see below).
- .so name The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

- .ab text Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.
- .fz F N forces font "F" to be in size N. N may have the form N, +N, or -N. For example,
.fz 3 -2
will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,
.fz S F N
may be used to specify the size treatment of special characters during font F. For example,
.fz 3 -3
.fz S 3 -0
will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers.

- .k Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.
- .j Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.
- .P Read-only. 1 if the current page is being printed, and zero otherwise.
- .L Read-only. Contains the current line-spacing parameter ("ls").
- c. General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

Table of Contents

Memorandum Macros

Introduction.....	1
Purpose.....	1
Conventions.....	1
Document Structure.....	1
Input Text Structure.....	2
Definitions.....	2
Usage.....	3
The Mm Command.....	3
The -cm or -mm Flag.....	4
Typical Command Lines.....	4
Parameters Set From Command Line.....	6
Omission of -cm or -mm Flag.....	8
Formatting Concepts.....	8
Basic Terms.....	8
Arguments and Double Quotes.....	9
Unpaddable Spaces.....	9
Hyphenation.....	10
Tabs.....	10
BEL Character.....	11
Bullets.....	11
Dashes, Minus Signs, and Hyphens.....	11
Trademark String.....	11
Use of Formatter Requests.....	12
Paragraphs and Headings.....	12
Paragraphs.....	12
Paragraph Indention.....	12
Numbered Paragraphs.....	13
Spacing Between Paragraphs.....	13
Numbered Headings.....	14
Normal Appearance.....	14
Altering Appearance.....	15
Unnumbered Headings.....	17
Headings and Table of Contents.....	18
First-Level Headings and Page Numbering Style.....	18
User Exit Macros.....	18
Hints for Large Documents.....	20
Lists.....	20
List Macros.....	20
List-Initialization Macros.....	20
List-Item Macro.....	23
List-End Macro.....	24
Example of Nested Lists.....	24

Table of Contents (continued)

List-Begin Macro and Customized Lists	26
User-Defined List Structures.....	27
Memorandum and Released-Paper Style Documents.....	30
Sequence of Beginning Macros	30
Title.....	30
Authors	31
TM Numbers	31
Abstract.....	32
Other Keywords	32
Memorandum Types	33
Date Changes.....	34
Alternate First-Page Format	34
Example.....	34
End of Memorandum Macros.....	35
Signature Block.....	35
"Copy to" and Other Notations	35
Approval Signature Line.....	37
One-Page Letter.....	37
Displays.....	37
Static Displays	37
Floating Displays	39
Tables	40
Equations.....	41
Figure, Table, Equation, and Exhibit Titles.....	42
List of Figures, Tables, Equations, and Exhibits.....	42
Footnotes	43
Automatic Numbering of Footnotes.....	43
Delimiting Footnote Text	43
Format Style of Footnote Text.....	43
Spacing Between Footnote Entries.....	44
Page Headers and Footers	45
Default Headers and Footers.....	45
Header and Footer Macros	45
Page Header.....	45
Even-Page Header	45
Odd-Page Header	46
Page Footer	46
Even-Page Footer	46
Odd-Page Footer	46
First Page Footer	46
Default Header and Footer With Section-Page Numbering	46
Strings and Registers in Header and Footer Macros.....	46
Header and Footer Example	47
Generalized Top-of-Page Processing.....	47

Table of Contents (continued)

Generalized Bottom-of-Page Processing.....	48
Top and Bottom (Vertical) Margins.....	48
Proprietary Marking.....	48
Private Documents.....	49
Table of Contents and Cover Sheet.....	49
Table of Contents.....	49
Cover Sheet.....	51
References.....	51
Automatic Numbering of References.....	51
Delimiting Reference Text.....	51
Subsequent References.....	52
Reference Page.....	52
Miscellaneous Features.....	52
Bold, Italic, and Roman Fonts.....	52
Justification of Right Margin.....	53
SCCS Release Identification.....	54
Two-Column Output.....	54
Column Headings for Two-Column Output.....	55
Vertical Spacing.....	55
Skipping Pages.....	55
Forcing an Odd Page.....	56
Setting Point Size and Vertical Spacing.....	56
Producing Accents.....	57
Inserting Text Interactively.....	57
Errors and Debugging.....	58
Error Terminations.....	58
Disappearance of Output.....	58
Extending and Modifying MM Macros.....	59
Naming Conventions.....	59
Names Used by Formatters.....	59
Names Used by MM.....	59
Names Used by CW, EQN/NEQN, and TBL Programs.....	60
Names Defined by User.....	60
Sample Extensions.....	60
Appendix Headings.....	60
Hanging Indent With Tabs.....	60
Summary.....	62
Examples.....	63
Useful Tables.....	67



IV. MEMORANDUM MACROS

1. Introduction

1.1 Purpose

This section is a guide and reference manual for users of Memorandum Macros (MM). These macros provide a general purpose package of text formatting macros for use with the UNIX operating system text formatters **nroff** and **troff** (refer to **troff**(1) in the User's Manual—UNIX Operating System for more details). A reference of the form **name**(*N*) points to page **name** in section *N* of the User's Manual.

1.2 Conventions

Each part of this section explains a single facility of MM. In general, the earlier a part occurs, the more necessary the information is for most users. Some of the later parts can be completely ignored if MM defaults are acceptable. Likewise, each part progresses from general case to special-case facilities. It is recommended that a user read a part in detail only to the point where there is enough information to obtain the desired format, then skim the rest of the part because some details may be of use to only a few.

Numbers enclosed in curly brackets ({}) refer to paragraph numbers within this section. For example, this is paragraph {1.2}.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

Figure 4.1 shows both **nroff** and **troff** formatter outputs (of files using MM macros) for a simple letter. In those cases in which the behavior of the two formatters is obviously different, the **nroff** formatter output is described first with the **troff** formatter output following in parentheses. For example:

The title is underlined (italic).

means that the title is underlined by the **nroff** formatter and italicized by the **troff** formatter.

1.3 Document Structure

Input for a document to be formatted with the MM text formatting macro package has four major segments, any of which may be omitted; if present, the segments must occur in the following order:

- *Parameter setting segment* sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for heading and lists, page headers and footers {9}, and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if the user is satisfied with default values; it produces no actual output, but performs only the formatter setup for the rest of the document.
- *Beginning segment* includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body segment* is the actual text of the document. It may be as small as a single paragraph or as large as hundreds of pages. It may have a hierarchy of headings up to seven levels deep {4}. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of list macros for automatic numbering, alphabetic sequencing, and "marking" of list items {5}. The body may also contain various types of displays, tables, figures, references, and footnotes {7, 8, 11}.
- *Ending segment* contains those items that occur only once at the end of a document. Included are signature(s) and lists of notations (e.g., "Copy to" lists) {6.11}. Certain macros may be invoked here to print

information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document {10}.

Existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author names, etc.) may differ depending on the document, there is a uniform way of typing it into an input text file.

1.4 Input Text Structure

In order to make it easy to edit or revise input file text at a later time.

- Input lines should be kept short
- Lines should be broken at the end of clauses
- Each new sentence should begin on a new line.

1.5 Definitions

Formatter refers to either the **nroff** or **troff** text-formatting program.

Requests are built-in commands recognized by the formatters. Although a user seldom needs to use these requests directly {3.10}, this section contains references to some of the requests. For example, the request

```
.sp
```

inserts a blank line in the output at the place the request occurs in the input text file.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. The **MM** package supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Table 4.A is an alphabetical list of macro names used by **MM**. The first line of each item lists the name of the macro, a brief description, and a reference to the paragraph in which the macro is described. The second line illustrates a typical call of the macro.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. These registers share the pool of names used by requests and macros. A string can be given a value via the **.ds** (define string) request; and its value can be obtained by referencing its name, preceded by “*” (for 1-character names) or “*(” (for 2-character names). For instance, the string **DT** in **MM** normally contains the current date, thus the input line

```
Today is \*(DT.
```

may result in the following output:

```
Today is December 16,1981.
```

The current date can be replaced, e.g.:

```
.ds DT 01/01/79
```

by invoking a macro designed for that purpose {6.8}. Table 4.B is an alphabetical list of string names used by **MM**. A brief description, paragraph reference, and initial (default) value(s) are given for each.

Number registers fill the role of integer variables. These registers are used for flags and for arithmetic and automatic numbering. A register can be given a value using a **.nr** request and be referenced by preceding its name by **\n** (for 1-character names) or **\n(** (for 2-character names). For example, the following sets the value of the register *d* to one more than that of the register *dd*:

```
.nr d 1+\n(dd
```

Table 4.C is an alphabetical list of number register names giving for each a brief description, paragraph reference, initial (default) value, and legal range of values (where [m:n] means values from m to n, inclusive).

Paragraph 14.1 contains naming conventions for requests, macros, strings, and number registers. Table 4.A, 4.B, and 4.C list all macros, strings, and number registers defined in MM.

2. Usage

This part describes how to access MM, illustrates UNIX operating system command lines appropriate for various output devices, and describes command line flags for the MM text formatting macro package.

2.1 The mm Command

The **mm(1)** command can be used to prepare documents using the **nroff** formatter and MM; this command invokes **nroff** with the **-cm** flag {2.2}. The **mm** command has options to specify preprocessing by **tbl** and/or by **neqn** and for postprocessing by various output filters. Any arguments or flags that are not recognized by the **mm** command, e.g., **-rC3**, are passed to the **nroff** formatter or to MM, as appropriate. Options, which can occur in any order but must appear before the file names, are:

OPTION	MEANING
-e	The neqn is to be invoked; also causes neqn to read <i>/usr/pub/eqnchar</i> [see eqnchar(7)].
-t	The tbl(1) is to be invoked.
-c	The col(1) is to be invoked.
-E	The -e option of the nroff formatter is to be invoked.
-y	The -mm (uncompacted macros) is to be used instead of -cm .
-12	The 12-pitch mode is to be used. The pitch switch on the terminal should be set to 12 if necessary.
-T450	Output is to a DASI 450. This is the default terminal type [unless \$TERM is set; see sh(1)]. It is also equivalent to -T1620 .
-T450-12	Output is to a DASI 450 in 12-pitch mode.
-T300	Output is to a DASI 300 terminal.
-T300-12	Output is to a DASI 300 in 12-pitch mode.
-T300s	Output is to a DASI 300S.
-T300s-12	Output is to a DASI 300S in 12-pitch mode.
-T4014	Output is to a Tektronix 4014.

OPTION	MEANING
-T37	Output is to a TELETYPE® Model 37.
-T382	Output is to a DTC-382.
-T4000a	Output is to a Trendata 4000A.
-TX	Output is prepared for an EBCDIC line printer.
-Thp	Output is to an HP264x (implies -c).
-T43	Output is to a TELETYPE® Model 43 (implies -c).
-T40/4	Output is to a TELETYPE® Model 40/4 (implies -c).
-T745	Output is to a Texas Instrument 700 series terminal (implies -c).
-T2631	Output is prepared for an HP2631 printer where -T2631-e and -T2631-c may be used for expanded and compressed modes, respectively (implies -c).
-Tlp	Output is to a device with no reverse or partial line motions or other special features (implies -c).

Any other -T option given does not produce an error; it is equivalent to -Tlp.

A similar command is available for use with the **troff** formatter [**mmt(1)**].

2.2 The -cm or -mm Flag

The MM package can also be invoked by including the -cm or -mm flag as an argument to the formatter. The -cm flag causes the precompact version of the macros to be loaded. The -mm flag causes the file `/usr/lib/tmac/tmac.m` to be read and processed before any other files. This action defines the MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the input text files.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4}):

- Text without tables or equations:

```
mm [options] file ...
or
nroff [options] -cm file ...
```

```
mmt [options] file ...
or
troff [options] -cm file ...
```

- Text with tables:

```
mm -t [options] file ...
or
tbl file ... | nroff [options] -cm
```

```
mmt -t [options] file ...
or
tbl file ... † troff [options] -cm
```

- Text with equations:

```
mm -e [options] file ...
or
neqn /usr/pub/eqnchar file ... † nroff [options] -cm
```

```
mmt -e [options] file ...
or
eqn /usr/pub/eqnchar file ... † troff [options] -cm
```

- Text with both tables and equations:

```
mm -t -e [options] file ...
or
tbl file ... † neqn /usr/pub/eqnchar — † nroff [options] -cm
```

```
mmt -t -e [options] file ...
or
tbl file ... † eqn /usr/pub/eqnchar — † troff \ [options] -cm
```

When formatting a document with the **nroff** processor, the output should normally be processed for a specific type of terminal because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly used terminal types and the command lines appropriate for them are given below. More information is found in paragraph {2.4} of this part, and **300(1)**, **450(1)**, **4014(1)**, **hp(1)**, **col(1)**, **termio(4)**, and **term(5)** of the User's Manual — UNIX Operating System.

- DASI 450 in 10-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (60 characters) where this is the default terminal type so no **-T** option is needed (unless **\$TERM** is set to another value):

```
mm file ...
or
nroff -T450 -h -cm file ...
```

- DASI 450 in 12-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (72 characters):

```
mm -12 file ...
or
nroff -T450-12 -h -cm file ...
```

or to increase the line length to 80 characters and decrease the offset to 3 characters:

```
mm -12 -rW80 -rO3 file ...
or
nroff -T450-12 -rW80 -rO3 -h -cm file ...
```

- Hewlett-Packard HP264x CRT family:

```
mm -Thp file ...
```

or
nroff -cm file ... | col | hp

- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 series, etc.):

mm -T745 file ...
or
nroff -cm file ... | col -x

The **tbl(1)** and **eqn(1)/neqn** formatters, if needed, must be invoked as shown in the command lines illustrated earlier.

If 2-column processing {12.4} is used with the **nroff** formatter, either the **-c** option must be specified to **mm(1)** [**mm(1)** uses **col(1)** automatically for many terminal types {2.1}] or the **nroff** formatter output must be postprocessed by **col(1)**. In the latter case, the **-T37** terminal type must be specified to the **nroff** formatter, the **-h** option must not be specified, and the output of **col(1)** must be processed by the appropriate terminal filter [e.g., **450(1)**]; **mm(1)** with the **-c** option handles all this automatically.

2.4 Parameters Set From Command Line

Number registers are commonly used within MM to hold parameter values that control various aspects of output style. Many of these values can be changed within the text files with **.nr** requests. In addition, some of these registers can be set from the command line. This is a useful feature for those parameters that should not be permanently embedded within the input text. If used, the number registers (with the possible exception of the register *P* below) must be set on the command line (or before the MM macro definitions are processed). The number register meanings are:

REGISTER	MEANING
-rAn	n = 1 has effect of invoking the .AF macro without an argument {6.9} . n = 2 permits use of Bell System logo, if available, on a printing device (currently available for Xerox 9700 only).
-rCn	sets type of copy (e.g., DRAFT) to be printed at bottom of each page {9.2.4} . n = 1 for OFFICIAL FILE COPY. n = 2 for DATE FILE COPY. n = 3 for DRAFT with single spacing and default paragraph style. n = 4 for DRAFT with double spacing and 10-space paragraph indent.
-rD1	sets debug mode . This flag requests formatter to continue processing even if MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2.1, 12.3} .

REGISTER

MEANING

- rE**n** controls font of Subject/Date/From fields.
n = 0, fields are bold (default for the **troff** formatter).
n = 1, fields are Roman font (regular text-default for the **nroff** formatter).
- rL**k** sets length of physical page to **k** lines.
For the **nroff** formatter, **k** is an unscaled number representing lines.
For the **troff** formatter, **k** must be scaled.
Default value is 66 lines per page.
This flag is used, for example, when directing output to a Versatec* printer.
- rN**n** specifies page numbering style.
n = 0 (default), all pages get the prevailing header {9.2.1} .
n = 1, page header replaces footer on page 1 only.
n = 2, page header is omitted from page 1.
n = 3, “section-page” numbering {4.5} occurs (.FD {8.3} and .RP {11.4} defines footnote and reference numbering in sections).
n = 4, default page header is suppressed; however, a user-specified header is not affected.
n = 5, “section-page” and “section-figure” numbering occurs.

n	PAGE 1	PAGES 2FF.
0	header	header
1	header replaces footer	header
2	no header	header
3	“section-page” as footer	same as page 1
4	no header	no header unless .PH defined
5	“section-page” as footer and “section-figure”	same as page 1

Contents of the prevailing header and footer do not depend on number register **N** value; **N** controls only whether the header (**N**=3) or the footer (**N**=5) is printed, as well as the page numbering style. If header and footer are null {9.2.1, 9.2.4} , the value of **N** is irrelevant.

- rO**k** offsets output **k** spaces to the right.
For the **nroff** formatter, **k** is an unscaled number representing lines or character positions.
For the **troff** formatter, **k** must be scaled.
This flag is helpful for adjusting output positioning on some terminals. The default offset, if this regular is not set on the command line, is 0.75 inches.
Note: Register name is the capital letter “O”.
- rP**n** specifies that pages of the document are to be numbered starting with **n**.
This register may also be set via a **.nr** request in the input text.
- rS**n** sets point size and vertical spacing for the document. The default **n** is 10, i.e., 10-point type on 12-point vertical spacing, giving six lines per inch {12.9} . This flag applies to the **troff** formatter only.

* Registered Trademark of Versatec, Inc.

REGISTER	MEANING
<code>-rTn</code>	provides register settings for certain devices. If n is 1, line length and page offset are set to 80 and 3, respectively. Setting n to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for n is 0. This flag applies to the nroff formatter only.
<code>-rU1</code>	controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This flag applies to the nroff formatter only.
<code>-rWk</code>	sets page width (line length and title length) to k . For the nroff formatter, k is an unscaled number representing character positions. For the troff formatter, k must be scaled. This flag can be used to change page width from the default value of 6 inches (60 characters in 10 pitch or 72 characters in 12 pitch).

2.5 Omission of `-cm` or `-mm` Flag

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

In this case, the user must not use the `-cm` or `-mm` flag [nor the `mm(1)` or `mmt(1)` command]; the `.so` request has the equivalent effect, but registers in {2.4} must be initialized before the `.so` request because their values are meaningful only if set before macro definitions are processed. When using this method, it is best to lock into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
.
.
.
```

specifies, for the **nroff** formatter, a line length (W) of 80, a page offset (O) of 10, and “section-page” (N) numbering.

3. Formatting Concepts

3.1 Basic Terms

Normal action of the formatters is to fill output lines from one or more input lines. Output lines may be

justified so that both the left and right margins are aligned. As lines are being filled, words may also be hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (**.SA** {12.2}, *Hy* {3.4}, and the **.nf** and **.fi** formatter requests). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause filling of the current output line to cease, the line (of whatever length) to be printed, and subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the MM macros cause a break.

Formatter requests {3.10} can be used with MM; however, there are consequences and side effects that each such request might have. A good rule is to use formatter requests only when absolutely necessary. The MM macros described herein should be used in most cases because:

- It is much easier to control (and change at any later point in time) overall style of the document.
- Complicated features (such as footnotes or tables of contents) can be obtained with ease.
- User is insulated from peculiarities of the formatter language.

3.2 Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is " ". Omitting an argument is not the same as supplying a null argument (e.g., the **.MT** macro in {6.7}). Omitted arguments can occur only at the end of an argument list; null arguments can occur anywhere in the list.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes. A double quote (") is a single character that must not be confused with two apostrophes or acute accents (') or with two grave accents (`). Otherwise, it will be treated as several separate arguments.

Double quotes are not permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If it is necessary to have a macro argument value, two grave accents (`) and/or two acute accents (') may be used instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times. For example, headings are first printed in the text and may be reprinted in the table of contents.

3.3 Unpaddable Spaces

When output lines are justified to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may distort the desired alignment of text. To avoid this distortion, it is necessary to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this:

- The user may type a backslash followed by a space (\). This pair of characters directly generates an unpaddable space.
- The user may sacrifice some seldom-used character to be translated into a space upon output.

Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used with the translation macro for this purpose. To use the tilde in this way, the following is inserted at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily “recovered” by inserting

```
.tr ~ ~
```


before the place where needed. Its previous usage is restored by repeating the `.tr ~` after a break or after the line containing the tilde has been forced out.

Note: Use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

Formatters do not perform hyphenation unless requested. Hyphenation can be turned on in the body of the text by specifying

```
.nr Hy 1
```

once at the beginning of the document input file. Paragraph 8.3 describes hyphenation within footnotes and across pages,

If hyphenation is requested, formatters will automatically hyphenate words if need be. However, the user may specify hyphenation points for a specific occurrence of any word with a special character known as a hyphenation indicator or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the 2-character sequence “\%”) appears at the beginning of a word, the word is not hyphenated. Alternatively, it can be used to indicate legal hyphenation points inside a word. All occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator.

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose by inserting the following at the beginning of a document input text file:

```
.HC ^
```

Note: Any word containing hyphens or dashes (also known as *em* dashes) will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, even if the formatter hyphenation function is turned off.

The user may supply, via the exception word `.hw` request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word “printout”, the user may specify

```
.hw print-out
```

3.5 Tabs

Macros `.MT` {6.7}, `.TC` {10.1}, and `.CS` {10.2} use the formatter tabs `.ta` request to set tab stops and then restore the default values of tab settings (every eight characters in the `nroff` formatter; every ½ inch in the `troff` formatter). Setting tabs to other than the default values is the user’s responsibility.

Default tab setting values are 9, 17, 25, ..., 161 for a total of 20 tab stops. Values may be separated by commas, spaces, or any other non-numeric character. A user may set tab stops at any value desired. For example:

```
.ta 9 17 25 33 41 49 57 ... 161
```

A tab character is interpreted with respect to its position on the input line rather than its position on the output line. In general, tab characters should appear only on lines processed in no-fill (`.nf`) mode {3.1}.

The **tbl(1)** program {7.3} changes tab stops but does not restore default tab settings.

3.6 BEL Character

The nonprinting character BEL is used as a delimiter in many macros to compute the width of an argument or to delimit arbitrary text, e.g., in page headers and footers {9}, headings {4}, and lists {5}. Users who include BEL characters in their input text file (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (●) is often obtained on a typewriter terminal by using an “o” overstruck by a “+”. For compatibility with the **troff** formatter, a bullet string is provided by MM with the following sequence:

```
\*(BU
```

The bullet list (**.BL**) macro {5.1.1.2} uses this string to generate automatically the bullets for bullet-listed items.

3.8 Dashes, Minus Signs, and Hyphens

The **troff** formatter has distinct graphics for a dash, a minus sign, and a hyphen; the **nroff** formatter does not.

- Users who intend to use the **nroff** formatter only may use the minus sign (–) for the minus, hyphen, and dash.
- Users who plan to use the **troff** formatter primarily should follow **troff** escape conventions.
- Users who plan to use both formatters must take care during input text file preparation. Unfortunately, these graphic characters cannot be represented in a way that is both compatible and convenient for both formatters.

The following approach is suggested:

SYMBOL	ACTION
Dash	Type *(EM for each text dash for both nroff and troff formatters. This string generates an em dash in the troff formatter and two dashes (—) in the nroff formatter. Dash list (.DL) macros {5.2.1.3} automatically generate the em dash for each list item.
Hyphen	Type - and use as is for both formatters. The nroff formatter will print it as is, and the troff formatter will print - (a true hyphen).
Minus	Type \- for a true minus sign regardless of formatter. The nroff formatter will effectively ignore the “\”; the troff formatter will print a true minus sign.

3.9 Trademark String

A trademark string ***(Tm** is available with MM. This places the letters “TM” one-half line above the text that it follows. For example:

```
The  
.I  
User’s Manual—UNIX  
.R
```

```
\h'-1'\*(Tm
.I
Operating System
.R
is available from the library.
```

yields:

The User's Manual—UNIX™ Operating System is available from the library.

3.10 Use of Formatter Requests

Most formatter requests should not be used with MM because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests. However, some formatter requests are useful with MM, namely the following:

.af	Assign format
.br	Break
.ce	Center
.de	Define macro
.ds	Define string
.fi	Fill output lines
.hw	Exception word
.ls	Line spacing
.nf	No filling of output lines
.nr	Define and set number register
.nx	Go to next file (does not return)
.rm	Remove macro
.rr	Remove register
.rs	Restore spacing
.so	Switch to source file and return
.sp	Space
.ta	Tab stop settings
.ti	Temporary indent
.tl	Title
.tr	Translate
!	Escape

The **.fp**, **.lg**, and **.ss** requests are also sometimes useful for the **troff** formatter. Use of other requests without fully understanding their implications very often leads to disaster.

4. Paragraphs and Headings

4.1 Paragraphs

```
.P [type]
one or more lines of text.
```

The **.P** macro is used to control paragraph style.

4.1.1 Paragraph Indentation

An indented or a nonindented paragraph is defined with the *type* argument.

<i>type</i>	<i>Result</i>
0	left justified
1	indent

In a left-justified paragraph, the first line begins at the left margin. In an indented paragraph, the paragraph is indented the amount specified in the *Pi* register (default value is 5). For example, to indent paragraphs by ten spaces, the following is entered at the beginning of the document input file:

```
.nr Pi 10
```

A document input file possesses a default paragraph type obtained by specifying “.P” before each paragraph that does not follow a heading {4.2}. Default paragraph type is controlled by the *Pt* number register. The initial value of *Pt* is 0, which provides left-justified paragraphs.

All paragraphs can be forced to be indented by inserting the following at the beginning of the document input file:

```
.nr Pt 1
```

All paragraphs can be indented except after headings, lists, and displays by entering the following at the beginning of the document input file:

```
.nr Pt 2
```

Both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

Note: Values that specify indentation must be unscaled and are treated as character positions, i.e., as a number of ens. In the **nroff** formatter, an en is equal to the width of a character. In the **troff** formatter, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size.

Regardless of the value of *Pt*, an individual paragraph can be forced to be left-justified or indented. The “.P 0” macro request forces left justification; “.P 1” causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a list, the indent (if any) of the paragraph is added to the current list indent {5}.

4.1.2 Numbered Paragraphs

Numbered paragraphs may be produced by setting the *Np* register to 1. This produces paragraphs numbered within first level headings, e.g., 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the **.nP** macro rather than the **.P** macro for paragraphs. This produces paragraphs that are numbered within second level headings.

```
.H 1 " FIRST HEADING "
.H 2 " Second Heading "
.nP
one or more lines of text
```

The paragraphs contain a “double-line indent” in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

4.1.3 Spacing Between Paragraphs

The *Ps* number register controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (one-half a vertical space).

4.2 Numbered Headings

.H level [heading-text] [heading-suffix]
zero or more lines of text

The **.H** macro provides seven levels of numbered headings. Level 1 is the highest; level 7 the lowest.

The *heading-suffix* argument is appended to the *heading-text* argument and may be used for footnote marks which should not appear with heading text in the table of contents.

Note: There is no need for a **.P** macro immediately after a **.H** or **.HU {4.3}** because the **.H** macro also performs the function of the **.P** macro. Any immediately following **.P** macro is ignored. It is, however, good practice to start every paragraph with a **.P** macro, thereby ensuring that all paragraphs uniformly begin with a **.P** throughout an entire document.

4.2.1 Normal Appearance

The effect of the **.H** macro varies according to argument level. First-level headings are preceded by two blank lines (one vertical space); all others are preceded by one blank line (one-half a vertical space). The following table describes the default effect of the level argument.

.H 1 heading-text	Produces a bold font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.
.H 2 heading-text	Produces a bold font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.
.H <i>n</i> heading-text	Produces an underlined (italicized) heading followed by two spaces ($3 < n < 7$). <i>The following text begins on the same line, i.e., these are run-in headings.</i>

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading-text argument is omitted from a **.H** macro call.

The following listing gives the first few **.H** calls used for this part.

```
.H 1 " paragraphs and headings "  
.H 2 " Paragraphs "  
.H 3 " Paragraph Indention "  
.H 3 " Numbered Paragraphs "  
.H 3 " Spacing Between Paragraphs "  
.H 2 " Numbered Headings "  
.H 3 " Normal Appearance "  
.H 3 " Altering Appearance "  
.H 4 " Prespacing and Page Ejection "  
.H 4 " Spacing After Headings "  
.H 4 " Centered Headings "
```

.H 4 " Bold, Italic, and Underlined Headings "
.H 5 "Control by Level"

Note: Users satisfied with the default appearance of headings may skip to the paragraph entitled "Un-numbered Headings" {4.3}.

4.2.2 Altering Appearance

The user can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document input text file. This permits quick alteration of a document's style because this style-control information is concentrated in a few lines rather than being distributed throughout the document.

4.2.2.1 Prespacing and Page Ejection

A first-level heading (.H 1) normally has two blank lines (one vertical space) preceding it, and all other headings are preceded by one blank line (one-half a vertical space). If a multiline heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting

```
.nr Ej 1
```

at the beginning of the document input text file. Long documents may be made more manageable if each section starts on a new page. Setting the *Ej* register to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to the *Ej* value.

4.2.2.2 Spacing After Headings

Three registers control the appearance of text immediately following a .H call. The registers are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (one-half a vertical space) is inserted after the heading. The default value for *Hb* and *Hs* is 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document while allowing easy alteration of white space and heading emphasis.

For any stand-alone heading, i.e., a heading not run into the following text, alignment of the next line of output is controlled by the *Hi* register.

- If *Hi* is 0, text is left-justified.
- If *Hi* is 1 (the default value), text is indented according to the paragraph type as specified by the *Pt* register {4.1}.
- If *Hi* is 2, text is indented to line up with the first word of the heading itself so that the heading number stands out more clearly.

To cause a blank line (one-half a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the beginning of the document input text file:

```
.nr Hs 3  
.nr Hb 7  
.nr Hi 0
```

4.2.2.3 Centered Headings

The *Hc* register can be used to obtain centered headings. A heading is centered if its level argument is less than or equal to *Hc* and if it is also a stand-alone heading {4.2.2.2}. The *Hc* register is 0 initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings

4.2.2.4.1 Control by Level: Any heading that is underlined by the **nroff** formatter is italicized by the **troff** formatter. The string *HF* (heading font) contains seven codes that specify fonts for heading levels 1 through 7. Legal codes, code interpretations, and defaults for *HF* codes are:

FORMATTER	HF CODE			DEFAULT HF CODE
	1	2	3	
nroff	no underline	underline	bold	3 3 2 2 2 2 2
troff	Roman	italic	bold	3 3 2 2 2 2 2

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined by the **nroff** formatter and italicized by the **troff** formatter. The user may reset *HF* as desired. Any value omitted from the right end of the list is assumed to be a 1. The following request would result in five bold levels and two Roman font levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 NROFF Underlining Style: The **nroff** formatter underlines in either of two styles:

- The normal style (**.ul** request) is to underline only letters and digits.
- The continuous style (**.cu** request) underlines all characters including spaces.

By default, **MM** attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined but is longer than a single line, the heading is underlined in the normal style.

All underlining of headings can be forced to the normal style by using the **-rU1** flag when invoking the **nroff** formatter {2.4}.

4.2.2.4.3 Heading Point Sizes: The user may specify the desired point size for each heading level with the *HP* string (for use with the **troff** formatter only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (**.H** and **.HU**) is printed in the same point size as the body except that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 10 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type with the remainder printed

in 10-point. Specified values may also be relative point-size changes, for example:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then absolute sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then point sizes for headings will be relative to the point size of the body even if the latter is changed.

Null or zero values imply that default size will be used for the corresponding heading level.

Note: Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via `.HX` and/or `.HZ`) may cause overprinting.

4.2.2.5 Marking Styles--Numerals and Concatenation

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Register values are normally printed using Arabic numerals. The `.HM` macro (heading mark style) allows this choice to be overridden thus providing “outline” and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal arguments and their meanings are:

ARGUMENT	MEANING
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Uppercase alphabetic
a	Lowercase alphabetic
I	Uppercase Roman
i	Lowercase Roman

Omitted arguments are interpreted as 1; illegal arguments have no effect.

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, the heading mark type register (*Ht*) is set to 1. For example, a commonly used “outline” style is obtained by:

```
.HM I A 1 a i  
.nr Ht 1
```

4.3 Unnumbered Headings

```
.HU heading-text
```

The `.HU` macro is a special case of `.H`; it is handled in the same way as `.H` except that no heading mark is printed. In order to preserve the hierarchical structure of headings when `.H` and `.HU` calls are intermixed, each `.HU` heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between

```
.HU heading-text
```


and

`.H 2 heading-text`

is the printing of the heading mark for the latter. Both macros have the effect of incrementing the numbering counter for level 2 and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

The `.HU` macro can be especially helpful in setting up appendices and other sections that may not fit well into the numbering scheme of the main body of a document {14.2.1}.

4.4 Headings and Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following:

- specifying in the contents level register, *Cl*, what level headings are to be saved
- invoking the `.TC` macro {10.1} at the end of the document.

Any heading whose level is less than or equal to the value of the *Cl* register is saved and later displayed in the table of contents. The default value for the *Cl* register is 2, i.e., the first two levels of headings are saved.

Due to the way headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" formatter error message (Table 4.D) will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the "section-page" form where "section" is the number of the current first-level heading. This page numbering style can be achieved by specifying the `-rN3` or `-rN5` flag on the command line {9.3}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each first level section begins on a new page. In this style, the page number is printed at the bottom of the page so that the correct section number is printed.

4.6 User Exit Macros

Note: This paragraph is intended primarily for users who are accustomed to writing formatter macros.

```
.HX dlevel rlevel heading-text
.HY dlevel rlevel heading-text
.HZ dlevel rlevel heading-text
```

The `.HX`, `.HY`, and `.HZ` macros are the means by which the user obtains a final level of control over the previously described heading mechanism. These macros are not defined by MM. These macros are intended to be defined by the user. The `.H` macro invokes `.HX` shortly before the actual heading text is printed; it calls `.HZ` as its last action. After `.HX` is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in `.HX`, such as `.ti` for temporary indent, to be lost. After the size calculation, `.HY` is invoked so that the user may respecify these features. All default actions occur if these macros are not defined. If `.HX`, `.HY`, or `.HZ` are defined by the user, user-supplied definition is interpreted at the appropriate point. These macros can therefore influence handling of all headings because the `.HU` macro is actually a special case of the `.H` macro.

If the user originally invoked the `.H` macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the `.H` invocation. If the user originally invoked the `.HU` macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time `.H` calls `.HX`, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank lines (vertical spaces) to precede the heading {4.2.2.1}, and accumulated the “heading mark”, i.e., the string of digits, letters, and periods needed for a numbered heading. When `.HX` is called, all user-accessible registers and strings can be referenced, as well as the following:

string ;0	If <i>rlevel</i> is nonzero, this string contains the “heading mark”. Two unpaddable spaces (to separate the <i>mark</i> from the <i>heading</i>) have been appended to this string. If <i>rlevel</i> is 0, this string is null.
register ;0	This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (one-half a vertical space) is to follow the heading.
string ;2	If “register ;0” is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) heading from the following text. If “register ;0” is nonzero, this string is null.
register ;3	This register contains an adjustment factor for a <code>.ne</code> request issued before the heading is actually printed. On entry to <code>.HX</code> , it has the value 3 if <i>dlevel</i> equals 1, and 1 otherwise. The <code>.ne</code> request is for the following number of lines: the contents of the “register ;0” taken as blank lines (halves of vertical space) plus the contents of “register ;3” as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of {0, ;2, and ;3 within `.HX`. The following are examples of actions that might be performed by defining `.HX` to include the lines shown:

- Change first-level heading mark from format *n*. to *n.0*:
`.if \ $1=1 .ds }0\n(H1.0\<sp>\<sp>`
 (where `<sp>` stands for a space)
- Separate run-in heading from the text with a period and two unpaddable spaces:
`.if \ n;0=0 .ds} 2 .\<sp>\<sp>`
- Assure that at least 15 lines are left on the page before printing a first-level heading:
`.if \ $1=1 .nr ;3 15-\n;0`
- Add three additional blank lines before each first-level heading:
`.if \ $1=1 .sp 3`
- Indent level 3 run-in headings by five spaces:
`.if \ $1=3 .ti 5n`

If temporary strings or macros are used within `.HX`, their names should be chosen with care {14.1}.

When the `.HY` macro is called after the `.ne` is issued, certain features requested in `.HX` must be repeated. For example:

```
.de HY
.if \ $1=3 .ti 5n
..
```

The `.HZ` macro is called at the end of `.H` to permit user-controlled actions after the heading is produced. In a large document, sections may correspond to chapters of a book; and the user may want to change a page header or footer, e.g.:

```
.de HZ
.if \\$1=1 .PF " Section \\$3 "
..
```

4.7 Hints for Large Documents

A large document is often organized for convenience into one input text file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register `H1` to one less than the number of the section just before the corresponding `.H\ 1` call. For example, at the beginning of Part 5, insert

```
.nr H1 4
```

Note: This is not good practice. It defeats the automatic (re)numbering of sections when sections are added or deleted. Such lines should be removed as soon as possible.

5. Lists

This part describes different styles of lists; automatically numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, i.e., with terms or phrases to be defined.

5.1 List Macros

In order to avoid repetitive typing of arguments to describe the style or appearance of items in a list, `MM` provides a convenient way to specify lists. All lists share the same overall structure and are composed of the following basic parts:

- A *list-initialization macro* (`.AL`, `.BL`, `.DL`, `.ML`, `.RL`, or `.VL`) determines the style of list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing of list items.
- One or more *list-item macros* (`.LI`) identifies each unique item to the system. It is followed by the actual text of the corresponding list item.
- The *list-end macro* (`.LE`) identifies the end of the list. It terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation marking style, etc.); the `.LE` macro restores it.

With this approach, the format of a list is specified only once at the beginning of the list. In addition by building onto the existing structure, users may create their own customized sets of list macros with relatively little effort ({5.3} and {5.4}).

5.1.1 List-Initialization Macros

List-initialization macros are implemented as calls to the more basic `.LB` macro {5.2}.

They are:

.AL	Automatically Numbered or Alphabetized List
.BL	Bullet List
.DL	Dash List
.ML	Marked List
.RL	Reference List
.VL	Variable-Item List

5.1.1.1 Automatically Numbered or Alphabetized List

`.AL [type] [text-indent] [1]`

The **.AL** macro is used to begin sequentially numbered or alphabetized lists. If there are no arguments, the list is numbered; and text is indented by *Li* (initially six) spaces from the indent in force when the **.AL** is called. This leaves room for a space, two digits, a period, and two spaces before the text. Values that specify indentation must be unscaled and are treated as “character positions”, i.e., number of ens.

Spacing at the beginning of the list and between items can be suppressed by setting the list space register (*Ls*). The *Ls* register is set to the innermost list level for which spacing is done. For example:

`.nr Ls 0`

specifies that no spacing will occur around any list items. The default value for *Ls* is six (which is the maximum list nesting level).

- The *type* argument may be given to obtain a different type of sequencing. Its value indicates the first element in the sequence desired. If *type* argument is omitted or null, the value 1 is assumed.

ARGUMENT	INTERPRETATION
1	Arabic (default for all levels)
A	Uppercase alphabetic
a	Lowercase alphabetic
I	Uppercase Roman
i	Lowercase Roman

- If *text-indent* argument is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* argument is null, the value of *Li* will be used.
- If the third argument is given, a blank line (one-half a vertical space) will not separate items in the list. A blank line will occur before the first item however.

5.1.1.2 Bullet List

`.BL [text-indent] [1]`

The **.BL** macro begins a bullet list. Each list item is marked by a bullet (●) followed by one space.

- If the *text-indent* argument is non-null, it overrides the default indentation (the amount of paragraph indentation as given in the *Pi* register {4.1}). In the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

- If the second argument is specified, no blank lines will separate items in the list.

5.1.1.3 Dash List

`.DL [text-indent] [1]`

The `.DL` macro is identical to `.BL` except that a dash is used as the list item mark instead of a bullet.

5.1.1.4 Marked List

`.ML mark [text-indent] [1]`

The `.ML` macro is much like `.BL` and `.DL` macros but expects the user to specify an arbitrary *mark* which may consist of more than a single character.

- Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*.
- If the third argument is specified, no blank lines will separate items in the list.

Note: The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

5.1.1.5 Reference List

`.RL [text-indent] [1]`

A `.RL` macro call begins an automatically numbered list in which the numbers are enclosed by square brackets ([]).

- If the *text-indent* argument is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If the *text-indent* argument is omitted or null, the value of *Li* is used.
- If the second argument is specified, no blank lines will separate the items in the list.

5.1.1.6 Variable-Item List

`.VL text-indent [mark-indent] [1]`

When a list begins with a `.VL` macro, there is effectively no current *mark*; it is expected that each `.LI` will provide its own mark. This form is typically used to display definitions of terms or phrases.

- *Text-indent* provides the distance from current indent to beginning of the text.
- *Mark indent* produces the number of spaces from current indent to beginning of the *mark*, and it defaults to 0 if omitted or null.
- If the third argument is specified, no blank lines will separate items in the list.

An example of `.VL` macro usage is shown below:

```
.tr ~
.VL 20 2
```

```
.LI mark~1
Here is a description of mark 1;
“mark 1” of the .LI line contains a tilde
translated to an unpaddable space in order
to avoid extra spaces between
“mark” and “1” {3.3}.
.LI second~mark.
This is the second mark also using a tilde translated to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space separates the mark from the text.
.LI ~
This item effectively has no mark because the tilde following the .LI is translated into a space.
.LE
```

when formatted yields:

```
mark 1          Here is a description of mark 1; “mark 1” of the .LI line contains a tilde translated to
                an unpaddable space in order to avoid extra spaces between “mark” and “1” {3.3}.

second mark     This is the second mark also using a tilde translated to an unpaddable space.

third mark longer than indent: This item shows the effect of a long mark; one space separates the mark
                from the text.

                This item effectively has no mark because the tilde following the .LI is translated into
                a space.
```

The tilde argument on the last .LI above is required; otherwise, a “hanging indent” would have been produced. A “hanging indent” is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

when formatted yields:

```
Here is some text to show a hanging indent. The first line of text is at the left margin. The second is
indented 10 spaces.
```

Note: The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

5.1.2 List-Item Macro

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The .LI macro is used with all lists and for each list item. It normally causes output of a single blank line (one-half a vertical space) before its list item although this may be suppressed.

- If no arguments are given, .LI labels the item with the current *mark* which is specified by the most recent list-initialization macro.

- If a single argument is given, that argument is output instead of the current *mark*.
- If two arguments are given, the first argument becomes a prefix to the current *mark* thus allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark.

For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a “plus”.
.LI + 1
This uses a “plus” as prefix to the bullet.
.LE
```

when formatted yields:

- This is a simple bullet item.
- + This replaces the bullet with a “plus”.
- + • This uses “plus” as prefix to the bullet.

Note: The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

If the current *mark* (in the current list) is a null string and the first argument of `.LI` is omitted or null, the resulting effect is that of a “hanging indent”, i.e., the first line of the following text is moved to the left starting at the same place where *mark* would have started {5.1.1.6}.

5.1.3 List-End Macro

```
.LE [1]
```

The `.LE` macro restores the state of the list to that existing just before the most recent list-initialization macro call. If the optional argument is given, the `.LE` outputs a blank line (one-half a vertical space). This option should generally be used only when the `.LE` is followed by running text but not when followed by a macro that produces blank lines of its own such as the `.P`, `.H`, or `.LI` macro.

The `.H` and `.HU` macros automatically clear all list information. The user may omit the `.LE` macros that would normally occur just before either of these macros and not receive the “LE:mismatched” error message. Such a practice is not recommended because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.1.4 Example of Nested Lists

An example of input for the several lists and the corresponding output is shown below. The `.AL` and `.DL` macro calls {5.1.1} contained therein are examples of list-initialization macros. Input text is:

```
.AL A
.LI
This is alphabetized list item A.
```

This text shows the alignment of the second line of the item.

Notice the text indentations and alignment of left and right margins.

.AL

.LI

This is numbered item 1.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.DL

.LI

This is a dash item.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LI + 1

This is a dash item with a "plus" as prefix.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LE

.LI

This is numbered item 2.

.LE

.LI

This is another alphabetized list item B.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LE

.P

This paragraph follows a list item and is aligned with the left margin.

A paragraph following a list resumes the normal line length and margins.

The output is:

- A. This is alphabetized list item A. This text shows the alignment of the second line of the item. Notice the text indentions and alignment of left and right margins.
 1. This is numbered item 1. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - + - This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 2. This is numbered item 2.
- B. This is another alphabetized list item B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph follows a list item and is aligned with the left margin. A paragraph following a list resumes the normal line length and margins.

5.2 List-Begin Macro and Customized Lists

`.LB text-indent mark-indent pad type [mark] [LI-space]\[LB-space]`

List-initialization macros described above suffice for almost all cases. However, if necessary, the user may obtain more control over the layout of lists by using the basic list-begin macro (**.LB**). The **.LB** macro is used by the other list-initialization macros. Its arguments are as follows:

- The *text-indent* argument provides the number of spaces that text is to be indented from the current indent. Normally, this value is taken from the *Li* register (for automatic lists) or from the *Pi* register (for bullet and dash lists).
- The combination of *mark-indent* and *pad* arguments determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent). The *mark-indent* argument is typically 0.
- Within the *mark area*, the mark is left justified if the *pad* argument is 0. If *pad* is a number *n* (greater than 0) then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. The *mark* is effectively right justified *pad* spaces immediately to the left of text.
- Arguments *type* and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done; and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.1.1.1}. This is summarized in the following table .

<i>type</i>	<i>mark</i>	<i>result</i>
0	omitted	hanging indent
0	<i>string</i>	<i>string</i> is the mark
>0	omitted	Arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each nonzero value of *type* from one to six selects a different way of displaying the marks. The following table shows the output appearance for each value of *type*:

VALUE	APPEARANCE
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x }

where *x* is the generated number or letter.

Note: The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

- The *LI-space* argument gives the number of blank lines (halves of a vertical space) that should be output by each *.LI* macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the *.LI* macro issues a *.ne* request for two lines just before printing the mark.
- The *LB-space* argument is the number of blank lines (one-half a vertical space) to be output by *.LB* itself. If omitted *LB-space* defaults to 0.

There are three combinations of *LI-space* and *LB-space*:

- The normal case is to set *LI-space* to 1 and *LB-space* to 0 yielding one blank line before each item in the list; such a list is usually terminated with a *.LE 1* macro to end the list with a blank line.
- For a more compact list, *LI-space* is set to 0, *LB-space* is set to 1, and the *.LE 1* macro is used at the end of the list. The result is a list with one blank line before and after it.
- If both *LI-space* and *LB-space* are set to 0 and the *.LE* macro is used to end the list, a list without any blank lines will result.

Paragraph {5.3} shows how to build upon the supplied list of macros to obtain other kinds of lists.

5.3 User-Defined List Structures

Note: This part is intended only for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to define the appearance for each list level only once instead of having to define the appearance at the beginning of each list. This permits consistency of

style in a large document. A generalized list-initialization macro might be defined in such a way that what the macro does depends on the list-nesting level in effect at the time the macro is called. Levels 1 through 5 of the lists to be formatted may have the following appearance:

A.

[1]

•

a)

+

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, the user should know that the number register :g is used by the MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments :g, and each .LE call decrements it.

```
\" register g is used as a local temporary to save
\" :g before it is changed below
.de aL
.nr g \n(:g
.if \ng=0 .AL A           \" produces an A.
.if \ng=1 .LB \n(Li 0 1 4 \" produces a [1]
.if \ng=2 .BL           \" produces a bullet

.if \ng=3 .LB \n(Li 0 2 2 a \" produces an a)
.if \ng=4 .ML +         \" produces a +

..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
First line.
.aL
.LI
Second line.
.LE
.LI
Third line.
.LE
```

when formatted yields

A. First line.

[1] Second line.

B. Third line.

There is another approach to lists that is similar to the .H mechanism. List-initialization, as well as the .LI and the .LE macros, are all included in a single macro. That macro (defined as .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item.

```
.de bL
.ie \n(.$ .nr g \\\$1      \" if there is an argument,
.                          \" that is the level
.el .nr g \n(:g          \" if no argument, use current level
.if \\\ng-\n(:g>1 .)D     \" **ILLEGAL SKIPPING OF LEVEL
.                          \" increasing level by more than 1
.if \\\ng>\n(:g \{.aL \\\ng-1 \" if g > :g, begin new list
.nr                          \" and reset g to current level
.                          \" (.aL changes g)
.if \n(:g>\nng .LC \nng    \" if :g > g, prune back to
.                          \" correct level
.                          \" if :g = g, stay within
.                          \" current list
.LI                          \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of g from its argument rather than from :g. Invoking .bL without arguments causes it to stay at the current list level. The .LC (List Clear) macro removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call “.LC 0”. If text is to be resumed at the end of a list, insert the call “.LC 0” to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
First line.
.bL 2
Second line.
.bL 1
Third line.
.bL
Fourth line.
.LC 0
Fifth line.
```

when formatted yields

The quick brown fox jumped over the lazy dog's back.

A. First line.

[1] Second line.

B. Third line.

C. Fourth line.
Fifth line.

6. Memorandum and Released-Paper Style Documents

One use of MM is for the preparation of memoranda and released-paper documents (a documentation style used by Bell Laboratories, Inc.) which have special requirements for the first page and for the cover sheet. Data needed (title, author, date, case numbers, etc.) is entered the same for both styles; an argument to the **.MT** macro indicates which style is being used.

6.1 Sequence of Beginning Macros

Macros, if present, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg]
.AT [title] ...
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.NS [arg]
one or more lines of text
.NE
.OK [keyword] ...
.MT [type] [addressee]
```

The only required macros for a memorandum for file or a released-paper document are **.TL**, **.AU**, and **.MT**; all other macros (and their associated input lines) may be omitted if the features are not needed. Once **.MT** has been invoked, none of the above macros (except **.NS** and **.NE**) can be reinvoked because they are removed from the table of defined macros to save memory space.

If neither the memorandum nor released-paper document style is desired, the **TL**, **AU**, **TM**, **AE**, **OK**, **MT**, **ND**, and **AF** macros should be omitted from the input text. If these macros are omitted, the first page will have only the page header followed by the body of the document.

6.2 Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

Arguments to the **.TL** macro are the charging-case number(s) and filing-case number(s).

- The *charging-case* argument is the case number to which time was charged for the development of the project described in the memorandum. Multiple charging - case numbers are entered as “subarguments” by separating each from the previous with a comma and a space and enclosing the entire argument within double quotes.
- The *filing-case* argument is a number under which the memorandum is to be filed. Multiple filing case members are entered similarly. For example:

```
.TL “1 2 3 4 5, 6 7 8 9 0” 9 8 7 6 5 4 3 2 1
Construction of a Table of all Even Prime numbers
```

The title of the memorandum or released-paper document follows the **.TL** macro and is processed in fill mode.

The .br request may be used to break the title into several lines as follows:

```
.TL 12345
First Title Line
.br
\!.br
Second Title Line
```

On output, the title appears after the word “subject” in the memorandum style and is centered and bold in the released-paper document style.

If only a charging case number or only a filing case number is given, it will be separated from the title in the memorandum style by a dash and will appear on the same line as the title. If both case numbers are given and are the same, then “Charging and Filing Case” followed by the number will appear on a line following the title. If the two case numbers are different, separate lines for “Charging Case” and “File Case” will appear after the title.

6.3 Authors

```
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg]
.AT [title] ...
```

The **.AU** macro receives as arguments information that describes an author. If any argument contains blanks, that argument must be enclosed within double quotes. The first six arguments must appear in the order given. A separate **.AU** macro is required for each author.

The **.AT** macro is used to specify the author’s title. Up to nine arguments may be given. Each will appear in the signature block for memorandum style {6.11} on a separate line following the signer’s name. The **.AT** must immediately follow the **.AU** for the given author. For example:

```
.AU " J. J. Jones " JJJ PY 9876 5432 1Z-234
.AT Director " Materials Research Laboratory "
```

In the “from” portion in the memorandum style, the author’s name is followed by location and department number on one line and by room number and extension number on the next. The “x” for the extension is added automatically. Printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9 of the **.AU** macro, if present, will follow this normal author information in the “from” portion, each on a separate line. These last three arguments may be used for organizational numbering schemes, etc. For example:

```
.AU " S. P. Lename " SPL IH 9988 7766 5H-444 9876-543210.01MF
```

The name, initials, location, and department are also used in the signature block. Author information in the “from” portion, as well as names and initials in the signature block will appear in the same order as the **.AU** macros.

Names of authors in the released-paper style are centered below the title. Following the name of the last author, “Bell Laboratories” and the location are centered. The paragraph on memorandum types {6.7} contains information regarding authors from different locations.

6.4 TM Numbers

```
.TM [number] ...
```

If the memorandum is a technical memorandum, the TM numbers are supplied via the **.TM** macro. Up to

nine numbers may be specified. For example:

```
.TM 7654321 77777777
```

This macro call is ignored in the released-paper and external-letter styles {6.7}.

6.5 Abstract

```
.AS [arg] [indent]
text of abstract
.AE
```

If a memorandum has an abstract, the input is identified with the **.AS** (abstract start) and **.AE** (abstract end) delimiters. Abstracts are printed on page 1 of a document and/or on its cover sheet. There are three styles of cover sheet:

- released paper
- technical memorandum
- memorandum for file {10.2} (also used for engineer's notes, memoranda for record, etc.).

Cover sheets for released papers and technical memoranda are obtained by invoking the **.CS** macro {10.2}.

In released-paper style (first argument of the **.MT** macro {6.7} is 4) and in technical memorandum style if the first argument of **.AS** is:

- 0 Abstract will be printed on page 1 and on the cover sheet (if any).
- 1 Abstract will appear only on the cover sheet (if any).

In memoranda for file style and in all other documents (other than external letters) if the first argument of **.AS** is:

- 0 Abstract will appear on page 1 and there will be no cover sheet printed.
- 2 Abstract will appear only on the cover sheet which will be produced automatically (i.e., without invoking the **.CS** macro).

It is not possible to get either an abstract or a cover sheet with an external letter (first argument of the **.MT** macro is 5).

Notations such as a “copy to” list {6.11} are allowed on memorandum for file cover sheets; the **.NS** and **.NE** macros must appear after the **.AS** and **.AE** macros. Headings {4.2, 4.3} and displays {7} are not permitted within an abstract.

The abstract is printed with ordinary text margins; an indentation to be used for both margins can be specified as the second argument of **.AS**. Values that specify indentation must be unscaled and are treated as “character positions”, i.e., as the number of *ens*.

6.6 Other Keywords

```
.OK [keyword] ...
```

Topical keywords should be specified on a technical memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the **.OK** macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.7 Memorandum Types

`.MT [type] [addressee]`

The `.MT` macro controls the format of the top part of the first page of a memorandum or of a released-paper document and the format of the cover sheets. The *type* arguments and corresponding values are:

<i>type</i>	<i>Value</i>
""	no memorandum type printed
0	no memorandum type printed
<i>none</i>	MEMORANDUM FOR FILE
1	MEMORANDUM FOR FILE
2	PROGRAMMER'S NOTES
3	ENGINEER'S NOTES
4	released-paper style
5	external-letter style
" <i>string</i> "	<i>string</i> (enclosed in quotes)

If the *type* argument indicates a memorandum style document, the corresponding statement indicated under "Value" will be printed after the last line of author information. If *type* is longer than one character, then the string itself will be printed. For example:

```
.MT " Technical Note #5"
```

A simple letter is produced by calling `.MT` with a null (but not omitted) or 0 argument.

The second argument to `.MT` is the name of the addressee of a letter. If present, that name and the page number replace the normal page header on the second and following pages of a letter. For example:

```
.MT 1 " John Jones"
```

produces

```
John Jones — 2
```

The *addressee* argument may not be used if the first argument is 4 (released-paper style document).

The released-paper style is obtained by specifying

```
.MT 4 [1]
```

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless the `.AF` macro specifies a different company). If the optional second argument to `.MT 4` is given, then the name of each author is followed by the respective company

name and location. Information necessary for the memorandum style document but not for the released-paper style document is ignored.

If the released-paper style document is utilized, most BTL location codes are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the `.MT` macro is invoked. Thus, following the `.MT` macro, the user may reuse these string names. In addition, the macros for the end of a memorandum {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate `.AF` macro {6.9} and defining a string (with a 2-character name such as `ZZ`) for the address before each `.AU`. For example:

```
.TL
A Learned Treatise
.AF " Getem Inc."
.ds ZZ " 22 Maple Avenue, Sometown 09999"
.AU " F. Swatter" "" ZZ
.AF " Bell Laboratories"
.AU " Sam P. Lename" "" CB
.MT 4 1
```

In the external-letter style document (`.MT 5`), only the title (without the word “subject:”) and the date are printed in the upper left and right corners, respectively, on the first page. It is expected that preprinted stationery will be used with the company logo and address of the author.

6.8 Date Changes

```
.ND new-date
```

The `.ND` macro alters the value of the string `DT`, which is initially set to produce the current date. If the argument contains spaces, it must be enclosed within double quotes.

6.9 Alternate First-Page Format

```
.AF [company-name]
```

An alternate first-page format can be specified with the `.AF` macro. The words “subject”, “date”, and “from” (in the memorandum style) are omitted and an alternate company name is used.

If an argument is given, it replaces “Bell Laboratories” without affecting other headings. If the argument is null, “Bell Laboratories” is suppressed; and extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp.

The `.AF` with no argument suppresses “Bell Laboratories” and the “Subject/Date/From” headings, thus allowing output on preprinted stationery. The use of `.AF` with no arguments is equivalent to the use of `-rA1` {2.4}, except that the latter must be used if it is necessary to change the line length and/or page offset (which default to 5.8i and 1i, respectively, for preprinted forms). The command line options `-rOk` and `-rWk` {2.4} are not effective with `.AF`. The only `.AF` use appropriate for the `troff` formatter is to specify a replacement for “Bell Laboratories”.

The command line option `-rEn` {2.4} controls the font of the “Subject/Date/From” block.

6.10 Example

Input text for a document may begin as follows:

```
.TL
```

```

MM\*(EMMemorandum Macros
.AU " D. W. Smith" DWS PY
.AU " J. R. Mashey" JRM PY
.AU " E. C. Pariser (January 1980 Revision)" ECP PY
.AU " N. W. Smith (June 1980 Revision)" NWS PY
.MT 4

```

Figure 4.1 shows the input text file and both the **nroff** and **troff** formatter outputs for a simple letter.

6.11 End of Memorandum Macros

At the end of a memorandum document (but not of a released-paper document), signatures of authors and a list of notations can be requested. The following macros and their input are ignored if the released-paper style document is selected.

6.11.1 Signature Block

```

.FC [closing]
.SG [arg] [1]

```

The **.FC** macro prints “Yours very truly,” as a formal closing, if no argument is used. It must be given before the **.SG** macro. A different closing may be specified as an argument to **.FC**.

The **.SG** macro prints the author’s name(s) after the formal closing, if any. Each name begins at the center of the page. Three blank lines are left above each name for the actual signature.

- If no arguments are given, the line of reference data (location code, department number, author’s initials, and typist’s initials, all separated by hyphens) will not appear.
- A non-null first argument is treated as the typist’s initials and is appended to the reference data.
- A null first argument prints reference data without the typist’s initials or the preceding hyphen.
- If there are several authors and if the second argument is given, reference data is placed on the line of the first author.

Reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the **.SG** macro. For example:

```

.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen

```

6.11.2 “Copy to” and Other Notations

```

.NS [arg]
zero or more lines of the notation
.NE

```

Many types of notations (such as a list of attachments or “Copy to” lists) may follow signature and reference data. Various notations are obtained through the **.NS** macro, which provides for proper spacing and for breaking notations across pages, if necessary.

Codes for *arg* and the corresponding notations are:

<i>arg</i>	<i>Notations</i>
<i>none</i>	Copy to
" "	Copy to
0	Copy to
1	Copy (with att.) to
2	Copy (without att.) to
3	Att.
4	Atts.
5	Enc.
6	Encs.
7	Under Separate Cover
8	Letter to
9	Memorandum to
" <i>string</i> "	Copy (string) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to". For example:

.NS " with att. 1 only "

will generate

Copy (with att. 1 only) to

as the notation. More than one notation may be specified before the .NE macro because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as

```
Atts.
Attachment 1—List of register names
Attachment 2—List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz
```

The .NS and .NE macros may also be used at the beginning following .AS 2 and .AE to place the notation list on the memorandum for file cover sheet {6.5}. If notations are given at the beginning without .AS 2, they will be saved and output at the end of the document.

6.11.3 Approval Signature Line

`.AV approver's-name`

The `.AV` macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example:

`.AV " Jane Doe "`

produces

APPROVED:

Jane Doe

Date

6.12 One-Page Letter

At times, the user may like more space on the page forcing the signature or items within notations to the bottom of the page so that the letter or memo is only one page in length. This can be accomplished by increasing the page length with the `-rL n` option, e.g., `-rL90`. This has the effect of making the formatter believe that the page is 90 lines long and therefore providing more space than usual to place the signature or the notations.

Note: This will work only for a single-page letter or memo.

7. Displays

Displays are blocks of text that are to be kept together on a page and not split across pages. They are processed in an environment that is different from the body of the text (see the `.ev` request). The MM package provides two styles of displays—a *static* (`.DS`) style and a *floating* (`.DF`) style.

- In the *static* style, the display appears in the same relative position in the output text as it does in the input text. This may result in extra white space at the bottom of the page if the display is too long to fit in the remaining page space.
- In the *floating* style, the display “floats” through the input text to the top of the next page if there is not enough space on the current page. Thus input text that follows a floating display may precede it in the output text. A queue of floating displays is maintained so that their relative order of appearance in the text is not disturbed.

By default, a display is processed in no-fill mode with single spacing and is not indented from the existing margins. The user can specify indentation or centering as well as fill-mode processing.

Note: Displays and footnotes {8} may never be nested in any combination. Although lists {5} and paragraphs {4.1} are permitted, no headings (`.H` or `.HU`) {4.2, 4.3} can occur within displays or footnotes.

7.1 Static Displays

```
.DS [format] [fill] [rindent]
one or more lines of text
.DE
```

A static display is started by the **.DS** macro and terminated by the **.DE** macro. With no arguments, **.DS** accepts lines of text exactly as typed (no-fill mode) and will not indent lines from the prevailing left margin indentation or from the right margin.

- The *format* argument is an integer or letter used to control the left margin indentation and centering with the following meanings:

<i>format</i>	<i>Meaning</i>
" "	no indent
0 or L	no indent
1 or I	indent by standard amount
2 or C	center each line
3 or CB	center as a block
omitted	no indent

- The *fill* argument is an integer or letter and can have the following meanings:

<i>fill</i>	<i>Meaning</i>
" "	no-fill mode
0 or N	no-fill mode
1 or F	fill mode
omitted	no-fill mode

- The *rindent* argument is the number of characters that the line length should be decreased, i.e., an indentation from the right margin. This number must be unscaled in the **nroff** formatter and is treated as *ens*. It may be scaled in the **troff** formatter or else defaults to *ems*.

The standard amount of static display indentation is taken from the *Si* register, a default value of five spaces. Thus, text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same (default values), these two registers are independent.

The display *format* argument value 3 (or CB) centers (horizontally) the entire display as a block (as opposed to **.DS 2** and **.DF 2** which center each line individually). All collected lines are left justified, and the display is centered based on width of the longest line. This format must be used in order for the **eqn/neqn** “mark” and “lineup” feature to work with centered equations {7.4}.

By default, a blank line (one-half a vertical space) is placed before and after *static* and *floating* displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

The following example shows usage of all three arguments for *static* displays. This block of text will be indented five spaces from the left margin, filled, and indented five spaces from the right margin (i.e., centered). The input text

```
.DS I F 5
"We the people of the United States,
in order to form a more perfect union,
establish justice, ensure domestic tranquillity,
provide for the common defense,
and secure the blessings of liberty to
ourselves and our posterity,
do ordain and establish this Constitution for the
```

United States of America.”
.DE

produces

“We the people of the United States, in order to form a more perfect union, establish justice, ensure domestic tranquillity, provide for the common defense, and secure the blessings of liberty to ourselves and our posterity, do ordain and establish this Constitution to the United States of America.”

7.2 Floating Displays

```
.DF [format] [fill] [rindent]  
one or more lines of text  
.DE
```

A floating display is started by the **.DF** macro and terminated by the **.DE** macro. Arguments have the same meanings as static displays described above, except indent, no indent, and centering are calculated with respect to the initial left margin. This is because prevailing indent may change between the time the formatter first reads the floating display and when the display is printed. One blank line (one-half a vertical space) occurs before and after a floating display.

The user may exercise precise control over the output positioning of floating displays through the use of two number registers, *De* and *Df* (see below). When a floating display is encountered by the **nroff** or **troff** formatter, it is processed and placed onto a queue of displays waiting to be output. Displays are removed from the queue and printed in the order entered, which is the order they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, the new display is a candidate for immediate output on the current page. Immediate output is governed by size of display and the setting of the *Df* register code. The *De* register code controls whether text will appear on the current page after a floating display has been produced.

As long as the display queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in 2-column mode), the next display from the queue becomes a candidate for output if the *Df* register code has specified “top-of-page” output. When a display is output, it is also removed from the queue.

When the end of a section (using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This occurs before a **.SG**, **.CS**, or **.TC** macro is processed.

A display will fit on the current page if there is enough room to contain the entire display or if the display is longer than one page in length and less than half of the current page has been used. A wide (full-page width) display will not fit in the second column of a 2-column document.

The *De* and *Df* number register code settings and actions are as follows:

De Register

CODE	ACTION
0	No special action occurs (also the default condition).

CODE	ACTION
1	A page eject will always follow the output of each floating display, so only one floating display will appear on a page and no text will follow it.

Note: For any other code, the action performed is the same as for code 1.

Df Register

CODE	ACTION
0	Floating displays will not be output until end of section (when section-page numbering) or end of document.
1	Output new floating display on current page if there is space; otherwise, hold it until end of section or document.
2	Output exactly one floating display from queue to the top of a new page or column (when in 2-column mode).
3	Output one floating display on current page if there is space; otherwise, output to the top of a new page or column.
4	Output as many displays as will fit (at least one) starting at the top of a new page or column.

Note: If *De* register is set to 1, each display will be followed by a page eject causing a new top of page to be reached where at least one more display will be output (this also applies to code 5).

5	Output a new floating display on the current page if there is room (also the default condition). Output as many displays (at least one) as will fit on the page starting at the top of a new page or column.
---	--

Note: For any code greater than 5, the action performed is the same as for code 5.

The `.WC` macro {12.4} may also be used to control handling of displays in double-column mode and to control the break in text before floating displays.

7.3 Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The `.TS` (table start) and `.TE` (table end) macros make possible the use of the `tbl(1)` program. These macros are used to delimit text to be examined by `tbl` and to set proper spacing around the table. The display function and the `tbl` delimiting function are independent. In order to permit the user to keep together blocks that contain any mixture of tables, equations, filled text, unfilled text, and caption lines, the `.TS/.TE` block should be enclosed within a display (`.DS/.DE`). Floating tables may be enclosed inside floating displays (`.DF/.DE`).

Macros `.TS` and `.TE` permit processing of tables that extend over several pages. If a table heading is needed for each page of a multipage table, the "H" argument should be specified to the `.TS` macro as above. Following

the options and format information, table title is typed on as many lines as required and is followed by the `.TH` macro. The `.TH` macro must occur when “`.TS H`” is used for a multipage table. This is not a feature of `tbl` but of the definitions provided by the `MM` macro package.

The `.TH` (table header) macro may take as an argument the letter `N`. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller `.TS H/.TE` segments. For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (e.g., too many blocks of filled text). If each segment had its own `.TS H/.TH` sequence, it would have its own header. However, if each table segment after the first uses `.TS H/.TH N`, the table header will appear only at the beginning of the table and the top of each new page or column that the table continues onto.

For the `nroff` formatter, the `-e` option [`-E` for `mm(1) {2.1}`] may be used for terminals, such as the 450, that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. The `-e` is not effective with `col(1)`.

7.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

Mathematical typesetting programs `eqn(1)` and `neqn` expect to use the `.EQ` (equation start) and `.EN` (equation end) macros as delimiters in the same way that `tbl(1)` uses `.TS` and `.TE`; however, `.EQ` and `.EN` must occur inside a `.DS/.DE` pair. There is an exception to this rule — if `.EQ` and `.EN` are used to specify only the delimiters for in-line equations or to specify `eqn/neqn` defines, the `.DS` and `.DE` macros must not be used; otherwise, extra blank lines will appear in the output.

The `.EQ` macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the “vertical center” of the general equation. The `Eq` register may be set to 1 to change labeling to the left margin.

The equation will be centered for centered displays; otherwise, the equation will be adjusted to the opposite margin from the label.

7.5 Figure, Table, Equation, and Exhibit Titles

```
.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
.EX [title] [override] [flag]
```

The **.FG** (figure title), **.TB** (table title), **.EC** (equation caption), and **.EX** (exhibit caption) macros are normally used inside **.DS/.DE** pairs to automatically number and title figures, tables, and equations. These macros use registers *Fg*, *Tb*, *Ec*, and *Ex*, respectively (see paragraph {2.4} on `-rN5` to reset counters in sections). For example:

```
.FG " This is a Figure Title "
```

yields

Figure 1. This is a Figure Title

The **.TB** macro replaces “Figure” with “TABLE”, the **.EC** macro replaces “Figure” with “Equation”, and the **.EX** macro replaces “Figure” with “Exhibit”. The output title is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the **.af** request of the formatter. The format of the caption may be changed from

Figure 1. Title

to

Figure 1—Title

by setting the *Of* register to 1.

The *override* argument is used to modify normal numbering. If *flag* argument is omitted or 0, *override* is used as a prefix to the number; if the *flag* argument is 1, *override* is used as a suffix; and if the *flag* argument is 2, *override* replaces the number. If `-rN5 {2.4}` is given, “section-figure” numbering is set automatically and user-specified *override* string is ignored.

As a matter of formatting style, table headings are usually placed above the text of tables, while figure, equation, and exhibit titles are usually placed below corresponding figures and equations.

7.6 List of Figures, Tables, Equations, and Exhibits

A list of figures, tables, exhibits, and equations are printed following the table of contents if the number registers *Lf*, *Lt*, *Lx*, and *Le* (respectively) are set to 1. The *Lf*, *Lt*, and *Lx* registers are 1 by default; *Le* is 0 by default.

Titles of these lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES
.ds Lt LIST OF TABLES
.ds Lx LIST OF EXHIBITS
.ds Le LIST OF EQUATIONS
```

8. Footnotes

There are two macros (.FS and .FE) that delimit text of footnotes, a string that automatically numbers footnotes, and a macro (.FD) that specifies the style of footnote text. Footnotes are processed in an environment different from that of the body of text. Refer to .ev request.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters “*F” (i.e., invoking the string *F*) immediately after the text to be footnoted without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half line above the text to be footnoted.

8.2 Delimiting Footnote Text

```
.FS [label]
one or more lines of footnote text
.FE
```

There are two macros that delimit the text of each footnote. The .FS (footnote start) macro marks the beginning of footnote text, and the .FE (footnote end) macro marks the end. The *label* on the .FS, if present, will be used to mark footnote text. Otherwise, the number retrieved from the string *F* will be used. Automatically numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted must be followed by *label*, rather than by “*F”. Text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are not permitted between .FS and .FE macros. If footnotes are required in the title, abstract, or table {7.3} only labeled footnotes will appear properly. Everywhere else automatically numbered footnotes work correctly. For example:

Automatically numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

Labeled footnote:

```
This is a labeled*
.FS *

The footnote is labeled with an asterisk.
.FE
footnote.
```

Text of the footnote (enclosed within the .FS/.FE pair) should immediately follow the word to be footnoted in the input text, so that “*F” or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append an unpaddable space {3.3} to “*F” or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Figure 4.2 illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format Style of Footnote Text

```
.FD [arg] [1]
```

Within footnote text, the user can control formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left or right justification of the label when text indenting is used. The **.FD** macro is invoked to select the appropriate style.

The first argument is a number from the left column of the following table. Formatting style for each number is indicated in the remaining four columns. Further explanation of the first two of these columns is given in the definitions of the **.ad**, **.hy**, **.na**, and **.nh** (adjust, hyphenation, no adjust, and no hyphenation, respectively) requests in the **nroff** part of this document.

<u>arg</u>	<u>HYPHENATION</u>	<u>ADJUST</u>	<u>TEXT INDENT</u>	<u>LABEL JUSTIFICATION</u>
0	.nh	.ad	yes	left
1	.hy	.ad	yes	left
2	.nh	.na	yes	left
3	.hy	.na	yes	left
4	.nh	.ad	no	left
5	.hy	.ad	no	left
6	.nh	.na	no	left
7	.hy	.na	no	left
8	.nh	.ad	yes	right
9	.hy	.ad	yes	right
10	.nh	.na	yes	right
11	.hy	.na	yes	right

If the argument to **.FD** is greater than 11, the effect is as if “.FD 0” were specified. If the first argument is omitted or null, the effect is equivalent to “.FD 10” in the **nroff** formatter and to “.FD 0” in the **troff** formatter; these are also the respective initial default values.

If the second argument is specified, then when a first-level heading is encountered, automatically numbered footnotes begin again with 1. This is most useful with the “section-page” page numbering scheme. As an example, the input line

```
.FD " " 1
```

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading in a document.

Hyphenation across pages is inhibited by **MM** except for long footnotes that continue to the following page. If hyphenation is permitted, it is possible for the last word on the last line on the current page footnote to be hyphenated. The user has control over this situation by specifying an even **.FD** argument.

Footnotes are separated from the body of the text by a short line rule. Those that continue to the next page are separated from the body of the text by a full-width rule. In the **troff** formatter, footnotes are set in type two points smaller than the point size in the body of text.

8.4 Spacing Between Footnote Entries

Normally, one blank line (a 3-point vertical space) separates footnotes when more than one occurs on a page. To change this spacing, the *Fs* number register is set to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a 6-point vertical space) to occur between footnotes.

9. Page Headers and Footers

Text printed at the top of each page is called *page header*. Text printed at the bottom of each page is called *page footer*. There can be up to three lines of text associated with the header — every page, even page only, and odd page only. Thus the page header may have up to two lines of text — the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This part describes the default appearance of page headers and page footers and ways of changing them. The term *header* (not qualified by *even* or *odd*) is used to mean the page header line that occurs on every page, and similarly for the term *footer*.

9.1 Default Headers and Footers

By default, each page has a centered page number as the header. There is no default footer and no even/odd default headers or footers except as specified in paragraph {9.3}.

In a memorandum or a released-paper style document, the page header on the first page is automatically suppressed provided a break does not occur before the `.MT` macro is called. Macros and text in the following categories do not cause a break and are permitted before the memorandum types (`.MT`) macro:

- Memorandum and released-paper style document macros (`.TL`, `.AU`, `.AT`, `.TM`, `.AS`, `.AE`, `.OK`, `.ND`, `.AF`, `.NS`, and `.NE`)
- Page headers and footers macros (`.PH`, `.EH`, `.OH`, `.PF`, `.EF`, and `.OF`)
- The `.nr` and `.ds` requests.

9.2 Header and Footer Macros

For header and footer macros (`.PH`, `.EH`, `.OH`, `.PF`, `.EF`, and `.OF`), the argument `[arg]` is of the following form:

```
" 'left-part'center-part'right-part' "
```

If it is inconvenient to use apostrophe (') as the delimiter because it occurs within one of the parts, it may be replaced uniformly by any other character. In formatted output, the parts are left justified, centered, and right justified, respectively.

9.2.1 Page Header

```
.PH [arg]
```

The `.PH` macro specifies the header that is to appear at the top of every page. The initial value is the default centered page number enclosed by hyphens. The page number contained in the `P` register is an Arabic number. The format of the number may be changed by the `.af` macro request.

If “*debug mode*” is set using the flag `-rD1` on the command line {2.4}, additional information printed at the top left of each page is included in the default header. This consists of the Source Code Control System (SCCS) release and level of MM (thus identifying the current version {12.3}) followed by the current line number within the current input file.

9.2.2 Even-Page Header

```
.EH [arg]
```

The **.EH** macro supplies a line to be printed at the top of each even-numbered page immediately following the header. Initial value is a blank line.

9.2.3 Odd-Page Header

.OH [arg]

The **.OH** macro is the same as the **.EH** except that it applies to odd-numbered pages.

9.2.4 Page Footer

.PF [arg]

The **.PF** macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the **-rCn** flag is specified on the command line {2.4}, the type of copy follows the footer on a separate line. In particular, if **-rC3** or **-rC4** (DRAFT) is specified, the footer is initialized to contain the date {6.8} instead of being a blank line.

9.2.5 Even-Page Footer

.EF [arg]

The **.EF** macro supplies a line to be printed at the bottom of each even-numbered page immediately preceding the footer. Initial value is a blank line.

9.2.6 Odd-Page Footer

.OF [arg]

The **.OF** macro is the same as **.EF** except that it applies to odd-numbered pages.

9.2.7 First Page Footer

By default, the first page footer is a blank line. If, in the input text file, the user specifies **.PF** and/or **.OF** before the end of the first page of the document, these lines will appear at the bottom of the first page.

The header (whatever its contents) replaces the footer on the first page only if the **-rN1** flag is specified on the command line {2.4}.

9.3 Default Header and Footer With Section-Page Numbering

Pages can be numbered sequentially within sections by “section-number page-number” {4.5}. To obtain this numbering style, **-rN3** or **-rN5** is specified on the command line. In this case, the default *footer* is a centered “section-page” number, e.g., 7-2; and the default page header is blank.

9.4 Strings and Registers in Header and Footer Macros

String and register names may be placed in arguments to header and footer macros. If the value of the string or register is to be computed when the respective header or footer is printed, invocation must be escaped by four backslashes. This is because string or register invocation will be processed three times:

1. As the argument to the header or footer macro
2. In a formatting request within the header or footer macro

3. In a `.tl` request during header or footer processing.

For example, page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH " "Page \\\nP "
```

creates a right-justified header containing the word “Page” followed by the page number. Similarly, to specify a footer with the “section-page” style, the user specifies (see paragraph {4.2.2.5} for meaning of *H1*):

```
.PF " "– \\\n(H1–\\n –' "
```

If the user arranges for the string *a*] to contain the current section heading which is to be printed at the bottom of each page, the `.PF` macro call would be:

```
.PF " "\\\*(a) "
```

If only one or two backslashes were used, the footer would print a constant value for *a*], namely, its value when `.PF` appeared in the input text.

9.5 Header and Footer Example

The following sequence specifies blank lines for header and footer lines, page numbers on the outside margin of each page (i.e., top left margin of even pages and top right margin of odd pages), and “Revision 3” on the top inside margin of each page (nothing is specified for the center):

```
.PH " "  
.PF " "  
.EH " "\\\nP”Revision 3’ "  
.OH " ’Revision 3’\\nP’ "
```

9.6 Generalized Top-of-Page Processing

Note: This part is intended only for users accustomed to writing formatter macros.

During header processing, `MM` invokes two user-definable macros:

- The `.TP` (top of page) macro is invoked in the environment (refer to `.ev` request) of the header.
- The `.PX` is a page header user-exit macro that is invoked (without arguments) when the normal environment has been restored and with the “no-space” mode already in effect.

The effective initial definition of `.TP` (after the first page of a document) is

```
.de TP  
.sp 3  
.tl \\\*(}t  
.if e ’tl \\\*(}e  
.if o ’tl \\\*(}o  
.sp 2  
..
```

The string `}t` contains the header, the string `}e` contains the even-page header, and the string `}o` contains the odd-page header as defined by the `.PH`, `.EH`, and `.OH` macros, respectively. To obtain more specialized page titles, the user may redefine the `.TP` macro to cause the desired header processing {12.5}. Formatting done within

the .TP macro is processed in an environment different from that of the body. For example, to obtain a page header that includes three centered lines of data, i.e., document number, issue date, and revision date, the user could define the .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.7 Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the **.BS** (bottom-block start) and **.BE** (bottom-block end) macros will be printed at the bottom of each page after the footnotes (if any) but before the page footer. This block of text is removed by specifying an empty block, i.e.:

```
.BS
.BE
```

The bottom block will appear on the table of contents, pages, and the cover sheet for memorandum for file, but not on the technical memorandum or released-paper cover sheets.

9.8 Top and Bottom (Vertical) Margins

```
.VM [top] [bottom]
```

The **.VM** (vertical margin) macro allows the user to specify additional space at the top and bottom of the page. This space precedes the page header and follows the page footer. The .VM macro takes two unscaled arguments that are treated as v's. For example:

```
.VM 10 15
```

adds 10 blank lines to the default top of page margin and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by redefining .TP).

9.9 Proprietary Marking

```
.PM [code]
```

The **.PM** (proprietary marking) macro appends to the page footer a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer. The *code* argument may be:

```
code    Disclaimer
none    turn off previous disclaimer, if any
```

<i>code</i>	<i>Disclaimer</i>
P	PRIVATE
N	NOTICE
BP	BELL LABORATORIES PROPRIETARY
BR	BELL LABORATORIES RESTRICTED

These disclaimers are in a form approved for use by the Bell System. The user may alternate disclaimers by use of the .BS/.BE macro pair.

9.10 Private Documents

.nr Pv value

The word “PRIVATE” may be printed, centered, and underlined on the second line of a document (preceding the page header). This is done by setting the *Pv* register *value*:

<i>value</i>	<i>Meaning</i>
0	do not print PRIVATE (default)
1	PRIVATE on first page only
2	PRIVATE on all pages

If *value* is 2, the user definable .TP macro may not be used because the .TP macro is used by MM to print “PRIVATE” on all pages except the first page of a memorandum on which .TP is not invoked.

10. Table of Contents and Cover Sheet

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively.

Note: This section refers to cover sheets for technical memoranda and released papers only. The mechanism for producing a memorandum for file cover sheet was discussed earlier {6.5}.

These macros normally appear once at the end of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros, and may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet may not be desired by a user and is therefore produced at the end.

10.1 Table of Contents

.TC [*slevel*] [*spacing*] [*tlevel*] [*tab*] [*head1*] [*head2*] [*head3*] [*head4*] [*head5*]

The .TC macro generates a table of contents containing heading levels that were saved for the table of contents as determined by the value of the *Cl* register {4.4}. Arguments to .TC control spacing before each entry, placement of associated page number, and additional text on the first page of the table of contents before the word “CONTENTS”.

Spacing before each entry is controlled by the first and second arguments (*slevel* and *spacing*). Headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (one-half a vertical space). The *slevel* argument does not control what levels of heading have been saved; saving of headings is the function of the *Cl* register.

The third and fourth arguments (*[tlevel]* and *[tab]*) control placement of associated page number for each heading. Page numbers can be justified at the right margin with either blanks or dots (called leaders) separating the heading text from the page number, or the page numbers can follow the heading text.

For headings whose level is less than or equal to *tlevel* (default 2), page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate heading text from page number. If *tab* is 0 (default value), dots (i.e., leaders) are used. If *tab* is greater than 0, spaces are used.

For headings whose level is greater than *tlevel*, page numbers are separated from heading text by two spaces (i.e., page numbers are “ragged right”, not right justified).

Additional arguments (*[head1]* ... *[head5]*) are horizontally centered on the page and precede the table of contents.

If the *.TC* macro is invoked with at most four arguments, the user-exit macro *.TX* is invoked (without arguments) before the word “CONTENTS” is printed, or the user-exit macro *.TY* is invoked and the word “CONTENTS” is not printed.

By defining *.TX* or *.TY* and invoking *.TC* with at most four arguments, the user can specify what needs to be done at the top of the first page of the table of contents. For example:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields the following output when the file is formatted

```
Special Application
Message Transmission

Approved: _____

CONTENTS
.
```

If the *.TX* macro were defined as *.TY*, the word “CONTENTS” would be suppressed. Defining *.TY* as an empty macro will suppress “CONTENTS” with no replacement:

```
.de TY
..
```

By default, the first level headings will appear in the table of contents left justified. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the

Ci string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the *Cl* register. Arguments must be scaled; for example, with “*Cl* = 5”:

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci 0 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents — *Oc* and *Cp*. By default, table of contents pages will have lowercase Roman numeral page numbering. If the *Oc* register is set to 1, the *.TC* macro will not print any page number but will instead reset the *P* register to 1. It is the user’s responsibility to give an appropriate page footer to specify the placement of the page number. Ordinarily, the same *.PF* macro (page footer) used in the body of the document will be adequate.

The list of figures, tables, etc. pages will be produced separately unless *Cp* is set to 1, which causes these lists to appear on the same page as the table of contents.

10.2 Cover Sheet

```
.CS [pages] [other] [total] [figs] [tbls] [refs]
```

The *.CS* macro generates a cover sheet in either the released paper or technical memorandum style (see paragraph {6.5} for details of the memorandum for file cover sheet). All other information for the cover sheet is obtained from data given before the *.MT* macro call {6.1}. If the technical memorandum style is used, the *.CS* macro generates the “Cover Sheet for Technical Memorandum”. The data that appear in the lower left corner of the technical memorandum cover sheet (counts of: pages of text, other pages, total pages, figures, tables, and references) are generated automatically (0 is used for “other pages”). These values may be changed by supplying the corresponding arguments to the *.CS* macro. If the released-paper style is used, all arguments to *.CS* are ignored.

11. References

There are two macros (*.RS* and *.RF*) that delimit the text of references, a string that automatically numbers the subsequent references, and an optional macro (*.RP*) that produces reference pages within the document.

11.1 Automatic Numbering of References

Automatically numbered references may be obtained by typing $\backslash*(Rf$ (invoking the string *Rf*) immediately after the text to be referenced. This places the next sequential reference number (in a smaller point size) enclosed in brackets one-half line above the text to be referenced. Reference count is kept in the *Rf* number register.

11.2 Delimiting Reference Text

```
.RS [string-name]  
.RF
```

The *.RS* and *.RF* macros are used to delimit text of each reference as shown below:

```
A line of text to be referenced.\*(Rf  
.RS  
reference text  
.RF
```

11.3 Subsequent References

The **.RS** macro takes one argument, a *string-name*. For example:

```
.RS aA
reference text
.RF
```

The string *aA* is assigned the current reference number. This string may be used later in the document as the string call, $\backslash*(aA$, to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets one-half line above the text to be referenced. No **.RS/.RF** pair is needed for subsequent references.

11.4 Reference Page

```
.RP [arg1] [arg2]
```

A reference page, entitled by default "References", will be generated automatically at the end of the document (before table of contents and cover sheet) and will be listed in the table of contents. This page contains the reference items (i.e., reference text enclosed within **.RS/.RF** pairs). Reference items will be separated by a space (one-half a vertical space) unless the *Ls* register is set to 0 to suppress this spacing. The user may change the reference page title by defining the *Rp* string:

```
.ds Rp " New Title "
```

The **.RP** (reference page) macro may be used to produce reference pages anywhere else within a document (i.e., after each major section). It is not needed to produce a separate reference page with default spacings at the end of the document.

Two **.RP** macro arguments allow the user to control resetting of reference numbering and page skipping.

<i>arg1</i>	<i>Meaning</i>
0	reset reference counter (default)
1	do not reset reference counter

<i>arg2</i>	<i>Meaning</i>
0	put on separate page (default)
1	do not cause a following .SK
2	do not cause a preceding .SK
3	no .SK before or after

If no **.SK** is issued by the **.RP** macro, a single blank line will separate the references from the following/preceding text. The user may wish to adjust spacing. For example, to produce references at the end of each major section:

```
.sp 3
.RP 1 2
.H 1 " Next Section "
```

12. Miscellaneous Features

12.1 Bold, Italic, and Roman Fonts

```
.B [bold-arg] [previous-font-arg] ...
```

```
.I [italic-arg] [previous-font-arg] ...  
.R
```

When called without arguments, the **.B** macro changes the font to bold and the **.I** macro changes to underlining (italic). This condition continues until the occurrence of the **.R** macro which causes the Roman font to be restored. Thus:

```
.I  
here is some text.  
.R
```

yields underlined text via the **nroff** and italic text via the **troff(1)** formatter.

If the **.B** or **.I** macro is called with one argument, that argument is printed in the appropriate font (underlined in the **nroff** formatter for **.I**). Then the previous font is restored (underlining is turned off in the **nroff** formatter). If two or more arguments (maximum six) are given with a **.B** or **.I** macro call, the second argument is concatenated to the first with no intervening space (1/12 space if the first font is italic) but is printed in the previous font. Remaining pairs of arguments are similarly alternated. For example:

```
.I italic " text " right-justified
```

produces

```
italic text right -justified
```

The **.B** and **.I** macros alternate with the prevailing font at the time the macros are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB .BI .IR .RI .RB .BR
```

Each macro takes a maximum of six arguments and alternates arguments between specified fonts.

When using a terminal that cannot underline, the following can be inserted at the beginning of the document to eliminate all underlining:

```
.rm ul  
.rm cu
```

Note: Font changes in headings are handled separately {4.2.2.4.1}.

12.2 Justification of Right Margin

```
.SA [arg]
```

The **.SA** macro is used to set right-margin justification for the main body of text. Two justification flags are used—*current* and *default*. The “.SA 0” call sets both flags to no justification; it acts like the **.na** request. The “.SA 1” call sets both flags to cause both right and left justification, the same as the **.ad** request. However, calling **.SA** without an argument causes the current flag to be copied from the default flag, thus performing either a **.na** or **.ad** depending on the *default*. Initially, both flags are set for no justification in the **nroff** formatter and for justification in the **troff** formatter.

In general, the no adjust request (**.na**) can be used to ensure that justification is turned off, but **.SA** should be used to restore justification, rather than the **.ad** request. In this way, justification or no justification for the remainder of the text is specified by inserting “.SA 0” or “.SA 1” once at the beginning of the document.

12.3 SCCS Release Identification

The *RE* string contains the SCCS release and the MM text formatting macro package current version level. For example:

```
This is version \*(RE of the macros.
```

produces

```
This is version 10.129 of the macros.
```

This information is useful in analyzing suspected bugs in MM. The easiest way to have the release identification number appear in the output is to specify `-rD1 {2.4}` on the command line. This causes the *RE* string to be output as part of the page header {9.2.1}.

12.4 Two-Column Output

```
.2C
text and formatting requests (except another .2C)
.1C
```

The MM text formatting macro package can format two columns on a page. The `.2C` macro begins 2-column processing which continues until a `.1C` macro (1-column processing) is encountered. In 2-column processing, each physical page is thought of as containing 2-columnar “pages” of equal (but smaller) “page” width. Page headers and footers are not affected by 2-column processing. The `.2C` macro does not balance 2-column output.

It is possible to have full-page width footnotes and displays when in 2-column mode, although default action is for footnotes and displays to be narrow in 2-column mode and wide in 1-column mode. Footnote and display width is controlled by the `.WC` (width control) macro, which takes the following arguments:

<i>arg</i>	<i>Meaning</i>
N	Default mode (<code>-WF</code> , <code>-FF</code> , <code>-WD</code> , <code>FB</code>).
WF	Wide footnotes (even in 2-column mode).
<code>-WF</code>	DEFAULT: Turn off WF. Footnotes follow column mode; wide in 1-column mode (1C), narrow in 2-column mode (2C), unless FF is set.
FF	First footnote. All footnotes have same width as first footnote encountered for that page.
<code>-FF</code>	DEFAULT: Turn off FF. Footnote style follows settings of WF or <code>-WF</code> .
WD	Wide displays (even in 2-column mode).
<code>-WD</code>	DEFAULT: Displays follow the column mode in effect when display is encountered.
FB	DEFAULT: Floating displays cause a break when output on the current page.
<code>-FB</code>	Floating displays on current page do not cause a break.

Note: The `“.WC WD FF”` command will cause all displays to be wide and all footnotes on a page to be the same width while `“.WC N”` will reinstate default actions. If conflicting settings are given to `.WC`, the last one is used. A `“.WC WF -WF”` command has the effect of a `“.WC -WF”`.

12.5 Column Headings for Two-Column Output

Note: This section is intended only for users accustomed to writing formatter macros.

In 2-column processing output, it is sometimes necessary to have headers over each column, as well as headers over the entire page. This is accomplished by redefining the `.TP` macro {9.6} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP'OVERALL"
.tl "TITLE"
.sp
.nf
.ta 16C 31R 34 50C 65R
left center right left center right
first column second column
.fi
.sp 2
..
```

where `Ø` stands for the tab character.

The above example will produce two lines of page header text plus two lines of headers over each column. Tab stops are for a 65-en overall line length.

12.6 Vertical Spacing

`.SP [lines]`

There exists several ways of obtaining vertical spacing, all with different effects. The `.sp` request spaces the number of lines specified unless the no space (`.ns`) mode is on, then the `.sp` request is ignored. The no space mode is set at the end of a page header to eliminate spacing by a `.sp` or `.bp` request that happens to occur at the top of a page. This mode can be turned off by the `.rs` (restore spacing) request.

The `.SP` macro is used to avoid the accumulation of vertical space by successive macro calls. Several `.SP` calls in a row will not produce the sum of the arguments but only the maximum argument. For example, the following produces only three blank lines:

```
.SP 2
.SP 3
.SP
```

Many MM macros utilize `.SP` for spacing. For example, “.LE 1” {5.1.3} immediately followed by “.P” {4.1} produces only a single blank line (one-half a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (one vertical space). Negative arguments are not permitted. The argument must be unscaled but fractional amounts are permitted. The `.SP` macro (as well as `.sp`) is also inhibited by the `.ns` request.

12.7 Skipping Pages

`.SK [pages]`

The `.SK` macro skips pages but retains the usual header and footer processing. If the *pages* argument is omitted, null, or 0, `.SK` skips to the top of the next page unless it is currently at the top of a page (then it does

nothing). The “.SK *n*” skips *n* pages. The .SK macro always positions text that follows it at the top of a page, while “.SK 1” always leaves one page blank except for the header and footer.

12.8 Forcing an Odd Page

.OP

The .OP macro is used to ensure that formatted output text following the macro begins at the top of an odd-numbered page. If currently at the top of an odd-numbered page, text output begins on that page (no motion takes place). If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page), one blank page is produced, and printing resumes on the next odd-numbered page after that.

12.9 Setting Point Size and Vertical Spacing

.S [point size] [vertical spacing]

In the **troff** formatter, the default point size (obtained from the MM register $S\{2.4\}$) is 10 points, and the vertical spacing is 12 points (six lines per inch). Prevailing point size and vertical spacing may be changed by invoking the .S macro.

The mnemonics D (default value), C (current value), and P (previous value) may be used for both arguments.

- If an argument is *negative*, current value is decremented by the specified amount.
- If an argument is *positive*, current value is incremented by the specified amount.
- If an argument is *unsigned*, it is used as the new value.
- If there are no arguments, the .S macro defaults to P.
- If the first argument is specified but the second is not, then (default) D is used for the vertical spacing.

Default value for vertical spacing is always two points greater than the current point size. Footnotes {8} are two points smaller than the body with an additional 3-point space between footnotes. A null (" ") value for either argument defaults to C (current value). Thus, if *n* is a numeric value:

```
.S          =.S P P
.S " " n   =.S C n
.S n " "   =.S n C
.S n       =.S n D
.S " "     =.S C D
.S " " "   =.S C C
.S n n     =.S n n
```

If the first argument is greater than 99, the default point size (10 points) is restored. If the second argument is greater than 99, the default vertical spacing (current point size plus two points) is used. For example:

```
.S 100     =.S 10 12
.S 14 111  =.S 14 16
```

The .SM macro allows the user to reduce by one point the size of a string:

.SM string1 [string2] [string3]

If the third argument is omitted, the first argument is made smaller and is concatenated with the second argument if the latter is specified. If all three arguments are present (even if any are null), the second argument is made smaller and all three arguments are concatenated. For example:

<i>INPUT</i>	<i>OUTPUT</i>
.SM X	X
.SM X Y	XY
.SM Y X Y	YXY
.SM YXYX	YXYX
.SM YXYX)	YXYX)
.SM (YXYX)	(YXYX)
.SM Y XYX " "	YXYX

12.10 Producing Accents

The following strings may be used to produce accents for letters:

	<i>INPUT</i>	<i>OUTPUT</i>
Grave accent	c*^	è
Acute accent	e*'	é
Circumflex	o*^	ô
Tilde	n*~	ñ
Cedilla	c*,	ç
Lower-case umlaut	u*:	ü
Upper-case umlaut	U*;	Û

12.11 Inserting Text Interactively

`.RD [prompt] [diversion] [string]`

The **.RD** (read insertion) macro allows a user to stop the standard output of a document and to read text from the standard input until two consecutive newline characters are found. When newline characters are encountered, normal output is resumed.

- The *prompt* argument will be printed at the terminal. If not given, .RD signals the user with a BEL on terminal output.
- The *diversion* argument allows the user to save all text typed in after the prompt in a macro whose name is that of the diversion.
- The *string* argument allows the user to save for later reference the first line following the prompt in the named string.

The .RD macro follows the formatting conventions in effect. Thus, the following examples assume that the .RD is invoked in no-fill mode (**.nf**):

`.RD Name aA bB`

produces

```
Name: J. Jones   (user types name)
16 Elm Rd.,
Piscataway
```

The diverted macro `.aA` will contain

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string `bB (*bB)` contains “J. Jones”.

A newline character followed by an EOF (user specifiable CONTROL d) also allows the user to resume normal output.

13. Errors and Debugging

13.1 Error Terminations

When a macro detects an error, the following actions occur:

- A break occurs.
- The formatter output buffer (which may contain some text) is printed to avoid confusion regarding location of the error.
- A short message is printed giving the name of the macro that detected the error, type of error, and approximate line number in the current input file of the last processed input line. Error messages are explained in Table 4.D.
- Processing terminates unless register D {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

The error message is printed by outputting the message directly to the user terminal. If an output filter, such as `300(1)`, `450(1)`, or `hp(1)` is being used to post-process the `nroff` formatter output, the message may be garbled by being intermixed with text held in that filter’s output buffer.

Note: If any of `cw(1)`, `eqn(1)/neqn`, and `tbl(1)` programs are being used and if the `-olist` option of the formatter causes the last page of the document not to be printed, a harmless “broken pipe” message may result.

13.2 Disappearance of Output

Disappearance of output usually occurs because of an unclosed diversion (e.g., a missing `.DE` or `.FE` macro). Fortunately, macros that use diversions are careful about it, and these macros check to make sure that illegal nestings do not occur. If any error message is issued concerning a missing `.DE` or `.FE`, the appropriate action is to search backwards from the termination point looking for the corresponding associated `.DF`, `.DS`, or `.FS` (since these macros are used in pairs).

The following command:

```
grep -n '^\. [EDFRT][EFNQS]' files ...
```

prints all the .DF, .DS, .DE, .EQ, .EN, .FS, .FE, .RS, .RF, .TS, and .TE macros found in *files ...*, each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

14. Extending and Modifying MM Macros

14.1 Naming Conventions

In this part, the following conventions are used to describe names:

- n: Digit
- a: Lowercase letter
- A: Uppercase letter
- x: Any alphanumeric character (: a, n, A, or 1, ie., letter or digit)
- s: any nonalphanumeric character (special character)

All other characters are literals (i.e., characters stand for themselves).

Request, macro, and string names are kept by the formatters in a single internal table; therefore, there must be no duplication among such names. Number register names are kept in a separate table.

14.1.1 Names Used by Formatters

requests: aa (most common)
 an (only one, currently: c2)

registers: aa (normal)
 .x (normal)
 .s (only one, currently: . $\$$)
 a. (only one, currently: c.)
 % (page number)

14.1.2 Names Used by MM

macros and strings: A, AA, Aa (accessible to users; e.g., macros P and HU, strings F, BU, and Lt)
 nA (accessible to users; only two, currently: 1C and 2C)
 aA (accessible to users; only one, currently: nP)
 s (accessible to users; only the seven accents, currently {12.10})
)x, }x,]x, >x, ?x (internal)

registers: An, Aa (accessible to users; e.g., H1, Fg)
 A (accessible to users; meant to be set on the command line; e.g., C)
 :x, ;x, #x, ?x, !x (internal)

14.1.3 Names Used by CW, EQN/NEQN, and TBL Programs

The **cw**(1) program is the constant-width font preprocessor for the **troff** formatter. It uses the following five macro names:

.CD, .CN, .CP, .CW, and .PC.

This preprocessor also uses the number register names *cE* and *cW*. Mathematical equation preprocessors, **eqn**(1) and **neqn**, use registers and string names of the form *nn*. The table preprocessor, **tbl**(1), uses T&, T#, and TW, and names of the form:

a- a+ a! nn na ^a #a #s

14.1.4 Names Defined by User

Names that consist either of a single lowercase letter or a lowercase letter followed by a character other than a lowercase letter (names .c2 and .nP are already used) should be used to avoid duplication with already used names. The following is a possible naming convention:

macros: aA (e.g., bG, kW)
strings: as (e.g., c, f], p})
registers: a (e.g., f, t)

14.2 Sample Extensions

14.2.1 Appendix Headings

The following is a way of generating and numbering appendix headings:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH " "Appendix \\na-\\\\\\\\\\\\\\\\nP' "
.SK
.HU " \\$1 "
..
```

After the above initialization and definition, each call of the form **aH " title"** begins a new page (with the page header changed to "Appendix *a-n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish appendix titles to be centered must, in addition, set the register *Hc* to 1 {4.2.2.3}.

14.2.2 Hanging Indent With Tabs

The following example illustrates the use of the hanging indent feature of variable-item lists {5.1.1.6}. A user-defined macro is defined to accept four arguments that make up the *mark*. In the output, each argument is to be separated from the previous one by a tab; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro call.

Note: The 2-character sequence "\&" is understood by formatters to be a "zero-width" space. It causes no output characters to appear, but it removes the special meaning of a leading period or apostrophe.

The “\t” is translated by the formatter into a tab. The “\c” is used to concatenate the input text that follows the macro call to the line built by the macro. The macro and an example of its use are:

```
.de aX
.LI
\&\$1\t\\$2\t\\$3\t\\$4\t\c
..
.
.
.ta 8 14 20 24
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\<sp>c none none no
Hyphenation indicator character is set to “c” or
removed.
During text processing, the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

where <sp> stands for a space.

The resulting output is:

.nh	off	—	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	—	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to “c” or removed. During text processing, the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

15. Summary

The following are qualities of MM that have been emphasized in its design in approximate order of importance:

- *Robustness in the face of error*—A user need not be an **nroff**/**troff** expert to use MM macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error or an error message describing the error is produced. An effort has been made to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce documents. Reasonable macro argument default values are provided where possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt input text files to produce output documents to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—A strong effort has been made to use mnemonic naming conventions and consistent techniques in construction of macros. Naming conventions are given so that a user can add new macros or redefine existing ones if necessary.
- *Device independence*—A common use of MM is to produce documents on hard copy via teletypewriter terminals using the **nroff** formatter. Macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be displayed on an appropriate CRT terminal. Macros have been constructed to allow compatibility with the **troff**(1) formatter so that output can be produced on both a phototypesetter and a teletypewriter/CRT terminal.
- *Minimization of input*—The design of macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, the user need only set a specific parameter once at the beginning of a document rather than type a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text although the user may obtain a number of output styles by setting a few global flags. For example, the **.H** macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or within a single document.

INPUT:

.ND "May 31, 1979"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the out-of-hours course
.I
Document Preparation on the UNIX*
.R
.FS *
Trademark of Bell Laboratories.
.FE
.I "Time-Sharing Operating System:"
.P
The course is intended for clerks, typists, and others
who intend to use the UNIX system for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system; using appropriate output terminals.
.LI Files:
how text is stored on the system;
directories; manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier to make;
how to use the editing system to add, delete, and move lines of text;
how to make corrections.
.LI "Text processing:"
basic concepts;
use of general purpose formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist such as the
.I "spell, diff,"
and
.I grep
commands, and a desk-calculator package.
.LE
.SG jrm
.NS 0
S. P. I ename
H. O. Del
M. Hill
.NE

Fig. 4.1 — Examples of a Simple Letter (Sheet 1 of 3)

nreff OUTPUT:

Bell Laboratories

subject: Out-of-Hours Course Description -
Case 334455

date: May 31, 1979

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the out-of-hours course
Document Preparation on the UNIX* Time-Sharing Operating System:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

Other facilities: additional capabilities useful to the typist such as the spell, diff, and grep commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* Trademark of Bell Laboratories.

Fig. 4.1 — Examples of a Simple Letter (Sheet 2 of 3)



Bell Laboratories

subject: **Out-of-Hours Course Description - Case 334455**

date: **May 31, 1979**

from: **D. W. Stevenson**
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course *Document Preparation on the UNIX* Time-Sharing System*:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

- Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
- Files: how text is stored on the system; directories; manipulating files.
- Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
- Text processing: basic concepts; use of general-purpose formatting packages.
- Other facilities: additional capabilities useful to the typist such as the *spell*, *diff*, and *grep* commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* Trademark of Bell Laboratories.

INPUT:

```
.P
.FD 10
This example illustrates several footnote styles
for both labeled and automatically numbered footnotes.
With the footnote style set to the NROFF default,
process the first footnote\*F
.FS
This is the first footnote text example (.FD 10).
This is the default style for NROFF.
The right margin is not justified.
Hyphenation is not permitted.
Text is indented, and the automatically generated label is
right justified in the text-indent space.
.FE
and follow it by a second footnote.*****
.FS *****
This is the second footnote text example (.FD 10).
This is also the default NROFF style but with a long
footnote label (*****) provided by the user.
.FE
.FD 1
Footnote style is changed by using the .FD macro to
specify hyphenation, right margin justification,
indentation, and left justification of the label.
This produces the third footnote,\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified, the footnote text is indented,
and the label is left justified in the text-indent space.
Although not necessarily illustrated by this example,
hyphenation is permitted.
.FE
and then the fourth footnote.\(dg
.FS †
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
Footnote style is set again via the .FD macro for no hyphenation,
no right margin justification,
no indentation, and with the label left justified.
This produces the fifth footnote.\*F
.FS
This is the fifth footnote example (.FD 6).
The right margin is not justified, hyphenation is not permitted,
footnote text is not indented,
and the label is placed at the beginning of the first line.
.FE
```

Fig. 4.2 — Examples of Footnotes (Sheet 1 of 2)

OUTPUT:

This example illustrates several footnote styles for both labeled and automatically numbered footnotes. With the footnote style set to the NROFF default, process the first footnote¹ and follow it by a second footnote.***** Footnote style is changed by using the .FD macro to specify hyphenation, right margin justification, indentation, and left justification of the label. This produces the third footnote,² and then the fourth footnote.† Footnote style is set again via the .FD macro for no hyphenation, no right margin justification, no indentation, and with the label left justified. This produces the fifth footnote.³

1. This is the first footnote text example (.FD 10). This is the default style for NROFF. The right margin is not justified. Hyphenation is not permitted. Text is indented, and the automatically generated label is right justified in the text-indent space.

***** This is the second footnote text example (.FD 10). This is also the default NROFF style but with a long footnote label (*****) provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, and the label is left justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is not justified, hyphenation is not permitted, footnote text is not indented, and the label is placed at the beginning of the first line.

Fig. 4.2 — Examples of Footnotes (Sheet 2 of 2)

TABLE 4.A

MM MACRO NAMES SUMMARY

MACRO	DESCRIPTION {PARAGRAPH}
1C	1-column processing {12.4} .1C
2C	2-column processing {12.4} .2C
AE	Abstract end {6.5} .AE
AF	Alternate format of "Subject/Date/From" block {6.9} .AF [company-name]
AL	Automatically incremented list start {5.1.1.1} .AL [type] [text-indent] [1]
AS	Abstract start {6.5} .AS [arg] [indent]
AT	Author's title {6.3} .AT [title] ...
AU	Author information {6.3} .AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
AV	Approval signature {6.11.3} .AV [name]
B	Bold {12.1} .B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]
BE	Bottom block end {9.7} .BE
BI	Bold/Italic {12.1} .BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]
BL	Bullet list start {5.1.1.2} .BL [text-indent] [1]
BR	Bold/Roman {12.1} .BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]
BS	Bottom block start {9.7} .BS
CS	Cover sheet {10.2} .CS [pages] [other] [total] [figs] [tbls] [refs]
DE	Display end {7.1} .DE
DF	Display floating start {7.2} .DF [format] [fill] [right-indent]
DL	Dash list start {5.1.1.3} .DL [text-indent] [1]

TABLE 4.A (Contd)

MM MACRO NAMES SUMMARY

MACRO	DESCRIPTION PARAGRAPH
DS	Display static start {7.1} .DS [format] [fill] [right-indent]
EC	Equation caption {7.5} .EC [title] [override] [flag]
EF	Even-page footer {9.2.5} .EF [arg]
EH	Even-page header {9.2.2} .EH [arg]
EN	End equation display {7.4} .EN
EQ	Equation display start {7.4} .EQ [label]
EX	Exhibit caption {7.5} .EX [title] [override] [flag]
FC	Formal closing {6.11} .FC [closing]
FD	Footnote default format {8.3} .FD [arg] [1]
FE	Footnote end {8.2} .FE
FG	Figure title {7.5} .FG [title] [override] [flag]
FS	Footnote start {8.2} .FS [label]
H	Heading—numbered {4.2} .H level [heading-text] [heading-suffix]
HC	Hyphenation character {3.4} .HC [hyphenation-indicator]
HM	Heading mark style {4.2.2.5} (Arabic or Roman numerals, or letters) .HM [arg1] ... [arg7]
HU	Heading—unnumbered {4.3} .HU heading-text
HX*	Heading user exit X (before printing heading) {4.6} .HX dlevel rlevel heading-text

*See note at end of table.

TABLE 4.A (Contd)

MM MACRO NAMES SUMMARY

MACRO	DESCRIPTION PARAGRAPH
HY*	Heading user exit Y (before printing heading) {4.6} .HY dlevel rlevel heading-text
HZ*	Heading user exit Z (after printing heading) {4.6} .HZ dlevel rlevel heading-text
I	Italic (underline in the nroff formatter) {12.1} .I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]
IB	Italic/Bold {12.1} .IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]
IR	Italic/Roman {12.1} .IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]
LB	List begin {5.2} .LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
LC	List-status clear {5.3} .LC [list-level]
LE	List end {5.1.3} .LE [1]
LI	List item {5.1.2} .LI [mark] [1]
ML	Marked list start {5.1.1.4} .ML mark [text-indent] [1]
MT	Memorandum type {6.7} .MT [type] [addressee] or .MT [4] [1]
ND	New date {6.8} .ND new-date
NE	Notation end {6.11.2} .NE
NS	Notation start {6.11.2} .NS [arg]
nP	Double-line indented paragraphs {4.1} .nP
OF	Odd-page footer {9.2.6} .OF [arg]
OH	Odd-page header {9.2.3} .OH [arg]
OK	Other keywords for the Technical Memorandum cover sheet {6.6} .OK [keyword] ...

*See note at end of table.

TABLE 4.A (Contd)

MM MACRO NAMES SUMMARY

MACRO	DESCRIPTION	PARAGRAPH
OP	Odd page {12.8} .OP	
P	Paragraph {4.1} .P [type]	
PF	Page footer {9.2.4} .PF [arg]	
PH	Page header {9.2.1} .PH [arg]	
PM	Proprietary marking {9.9} .PM [code]	
PX*	Page-header user exit {9.6} .PX	
R	Return to regular (Roman) font {12.1} .R	
RB	Roman/bold {12.1} .RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]	
RD	Read insertion from terminal {12.11} .RD [prompt] [diversion] [string]	
RF	Reference end {11.2} .RF	
RI	Roman/Italic {12.1} .RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]	
RL	Reference list start {5.1.1.5} .RL [text-indent] [1]	
RP	Produce reference page {11.4} .RP [arg] [arg]	
RS	Reference start {11.2} .RS [string-name]	
S	Set troff formatter point size and vertical spacing {12.9} .S [size] [spacing]	
SA	Set adjustment (right-margin justification) default {12.2} .SA [arg]	
SG	Signature line {6.11.1} .SG [arg] [1]	
SK	Skip pages {12.7} .SK [pages]	

*See note at end of table.

TABLE 4.A (Contd)

MM MACRO NAMES SUMMARY

MACRO	DESCRIPTION PARAGRAPH
SM	Make a string smaller {12.9} .SM string1 [string2] [string3]
SP	Space vertically {12.6} .SP [lines]
TB	Table title {7.5} .TB [title] [override] [flag]
TC	Table of contents {10.1} .TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]
TE	Table end {7.3} .TE
TH	Table header {7.3} .TH [N]
TL	Title of memorandum {6.2} .TL [charging-case] [filing-case]
TM	Technical Memorandum number(s) {6.4} .TM [number] ...
TP*	Top-of-page macro {9.6} .TP
TS	Table start {7.3} .TS [H]
TX*	Table of contents user exit {10.1} .TX
TY*	Table of contents user exit {10.1} (suppresses "CONTENTS") .TY
VL	Variable-item list start {5.1.1.6} .VL text-indent [mark-indent] [1]
VM	Vertical margins {9.8} .VM [top] [bottom]
WC	Footnote and Display Width control {12.4} .WC [format]

*Macros marked with an asterisk are not, in general, called (invoked) directly by the user. They are "user exits" defined by the user and called by the MM macros from inside header, footer, or other macros.

TABLE 4.B

STRING NAMES SUMMARY

STRING NAME	DESCRIPTION { PARAGRAPH }
BU	Bullet {3.7} NROFF: ● TROFF: •
Ci	Table of contents indent list {10.1} Up to seven args (must be scaled) for heading levels
DT	Date {6.8} Current date, unless overridden Month, day, year (e.g., July 16, 1982)
EM	Em dash string {3.8} Produces an em dash in the troff formatter and a double hyphen in nroff
F	Footnote numberer {8.1} NROFF:\u\\n+(;p\d TROFF:\v'-.4m'\s-3\\n+(;p\s0\v'.4m'
HF	Heading font list {4.2.2.4.1} Up to seven codes for heading levels 1 through 7 3 3 2 2 2 2 2 (levels 1 and 2 bold, 3 through 7 underlined in the nroff formatter and italic in troff)
HP	Heading point size list {4.2.2.4.3} Up to seven codes for heading levels 1 through 7
Le	Title for LIST OF EQUATIONS {7.6}
Lf	Title for LIST OF FIGURES {7.6}
Lt	Title for LIST OF TABLES {7.6}
Lx	Title for LIST OF EXHIBITS {7.6}
RE	SCCS Release and Level of MM {12.3} Release.Level (e.g., 10.129)
Rf	Reference numberer {11.1}
Rp	Title for references {11.4}
Tm	Trademark string {3.9} . Places the letters "TM" one-half line above the text that it follows Seven accent strings are also available {12.10}.

Note 1: If the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.7} . Currently, the following are recognized:

AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

Note 2: Paragraph 1.5 has notes on setting and referencing strings.

TABLE 4.C

NUMBER REGISTER NAMES SUMMARY

REGISTER	DESCRIPTION {PARAGRAPH}
A*†	Handles preprinted forms and Bell System logo {2.4} 0, [0:2]
Au	Inhibits printing author's location, department, room, and extension in "from" portion of a memorandum {6.3} 1, [0:1]
C*†	Copy type {2.4} Original, Draft, etc. 0 (Original), [0:4]
C1	Contents level {4.4} Level of headings saved for table of contents 2, [0:7]
Cp	Placement of list of figures, etc. {10.1} 1 (on separate pages), [0:1]
D*†	Debug flag {2.4} 0, [0:1]
De	Display eject register for floating displays {7.2} 0, [0:1]
Df	Display format register for floating displays {7.2} 5, [0:5]
Ds	Static display pre- and post-space {7.1} 1, [0:1]
E*†	Controls font of the Subject/Date/From fields {2.4} 1 (nroff) 0 (troff), [0:1]
Ec	Equation counter, used by .EC macro {7.5} 0, [0:?], incremented by one for each .EC call.
Ej	Page-ejection flag for headings {4.2.2.1} 0 (no eject), [0:7]
Eq	Equation label placement {7.4} 0 (right-adjusted), [0:1]
Ex	Exhibit counter, used by .EX macro {7.5} 0, [0:?], incremented by one for each .EX call.
Fg	Figure counter, used by .FG macro {7.5} 0, [0:?], incremented by one for each .FG call.
Fs	Footnote space (i.e., spacing between footnotes) {8.4} 1, [0:?]
H1 through H7	Heading counters for levels 1 through 7 {4.2.2.5} 0, [0:?], incremented by .H of corresponding level or .HU if at level given by register Hu. H2 through H7 are reset to 0 by any heading at a lower-numbered level.

*†See notes at end of table.

TABLE 4.C (Contd)

NUMBER REGISTER NAMES SUMMARY

REGISTER	DESCRIPTION {PARAGRAPH}
Hb	Heading break level (after .H and .HU) {4.2.2.2}2, [0:7]
Hc	Heading centering level for .H and .HU {4.2.2.3} 0 (no centered headings), [0:7]
Hi	Heading temporary indent (after .H and .HU) {4.2.2.2} 1 (indent as paragraph), [0:2]
Hs	Heading space level (after .H and .HU) {4.2.2.2} 2 (space only after .H 1 and .H 2), [0:7]
Ht	Heading type {4.2.2.5} For .H: single or concatenated numbers 0 (concatenated numbers: 1.1.1, etc.), [0:1]
Hu	Heading level for unnumbered heading (.HU) {4.3} 2 (.HU at the same level as .H 2), [0:7]
Hy	Hyphenation control for body of document {3.4} 0 (automatic hyphenation off), [0:1]
L*†	Length of page {2.4} 66, [20:?] (11i, [2i:?] in troff formatter) For nroff formatter, these values are unscaled numbers representing lines or character positions; for troff formatter, these values must be scaled..
Le	List of equations {7.6} 0 (list not produced) [0:1]
Lf	List of figures {7.6} 1 (list produced) [0:1]
Li	List indent {5.1.1.1} 6 (nroff) 5 (troff), [0:?]
Ls	List spacing between items by level {5.1.1.1} 6 (spacing between all levels) [0:6]
Lt	List of tables {7.6} 1 (list produced) [0:1]
Lx	List of exhibits {7.6} 1 (list produced) [0:1]
N*†	Numbering style {2.4} 0, [0:5]
Np	Numbering style for paragraphs {4.1} 0 (unnumbered) [0:1]

*†See notes at end of table.

TABLE 4.C (Contd)

NUMBER REGISTER NAMES SUMMARY

REGISTER	DESCRIPTION {PARAGRAPH}
O*†	Offset of page {2.4} .75i, [0:?] (0.5i, [0i:?] in troff formatter) For nroff formatter, these values are unscaled numbers representing lines or character positions; for troff formatter, these values must be scaled.
Oc	Table of contents page numbering style {10.1} 0 (lowercase Roman), [0:1]
Of	Figure caption style {7.5} 0 (period separator), [0:1]
P†	Page number manager by MM {2.4} 0, [0:?]
Pi	Paragraph indent {4.1} 5 (nroff) 3 (troff), [0:?]
Ps	Paragraph spacing {4.1} 1 (one blank space between paragraphs), [0:?]
Pt	Paragraph type {4.1} 0 (paragraphs always left justified), [0:2]
Pv	“PRIVATE” header {9.10} 0 (not printed), [0:2]
Rf	Reference counter, used by .RS macro {11.1} 0, [0:?], incremented by one for each .RS call.
S*†	The troff formatter default point size {2.4} 10, [6:36]
Si	Standard indent for displays {7.1} 5 (nroff) 3 (troff), [0:?]
T*†	Type of nroff output device {2.4} 0, [0:2]
Tb	Table counter, used by .TB macro {7.5} 0, [0:?], incremented by one for each .TB call.
U*†	Underlining style (nroff) for .H and .HU {2.4} 0 (continuous underline when possible), [0:1]
W*†	Width of page (line and title length) {2.4} 6i, [10:1365] (6i, [2i:7.54i] in the troff formatter)

*An asterisk attached to a register name indicates that this register can be set only from the command line or before the MM macro definitions are read by the formatter {2.4, 2.5} .

†Paragraph 1.5 has notes on setting and referencing registers. Any register having a single-character name can be set from the command line.

TABLE 4.D
ERROR MESSAGES

ERROR MESSAGE	DESCRIPTION
MM Error Messages	
<p>An MM error message has a standard part followed by a variable part. The standard part has the form:</p> <p style="text-align: center;">ERROR:(<i>filename</i>)input line <i>n</i>:</p> <p>Variable parts consist of a descriptive message usually beginning with a macro name. They are listed below in alphabetical order by macro name, each with a more complete explanation.</p>	
Check TL, AU, AS, AE, MT sequence	The correct order of macros at the start of a memorandum is shown in {6.1} . Something has disturbed this order.
Check TL, AU, AS, AE, NS, NE, MT sequence	The correct order of macros at the start of a memorandum is shown in {6.1} . Something has disturbed this order. Occurs if the .AS 2 {6.5}macro was used.
CS:cover sheet too long	Text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.5} .
DE:no DS or DF active	A .DE macro has been encountered, but there has not been a previous .DS or .DF macro to match it.
DF:illegal inside TL or AS	Displays are not allowed in the title or abstract.
DF:missing DE	A .DF macro occurs within a display, i.e., a .DE macro has been omitted or mistyped.
DF:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE macro to end a previous footnote.
DF:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:illegal inside TL or AS	Displays are not allowed in the title or abstract.
DS:missing DE	A .DS macro occurs within a display, i.e., a .DE has been omitted or mistyped.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
FE:no FS active	A .FE macro has been encountered with no previous .FS to match it.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
FS:missing FE	A previous .FS macro was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
H:bad arg: <i>value</i>	The first argument to the .H macro must be a single digit from one to seven, but <i>value</i> has been supplied instead.
H:missing arg	The .H macro needs at least one argument.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.

TABLE 4.D (Contd)

ERROR MESSAGES

ERROR MESSAGE	DESCRIPTION
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
HU:missing arg	The .HU macro needs one argument.
LB:missing arg(s)	The .LB macro requires at least four arguments.
LB:too many nested lists	Another list was started when there were already six active lists.
LE:mismatched	The .LE macro has occurred without a previous .LB or other list-initialization macro {5.1.1} . Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	The .LI macro occurred without a preceding list-initialization macro. The latter has probably been omitted or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	The .ML macro requires at least one argument.
ND:missing arg	The .ND macro requires one argument.
RF:no RS active	The .RF macro has been encountered with no previous .RS to match it.
RP:missing RF	A previous .RS macro was not matched by a closing .RF.
RS:missing RF	A previous .RS macro was not matched by a closing .RF.
S:bad arg:value	The incorrect argument <i>value</i> has been given for the .S macro {12.9} .
SA:bad arg:value	The argument to the .SA macro (if any) must be either 0 or 1. the incorrect argument is shown as <i>value</i> .
SG:missing DE	The .SG macro occurred inside a display.
SG:missing FE	The .SG macro occurred inside a footnote.
SG:no authors	The .SG macro occurred without any previous .AU macro(s).
VL:missing arg	The .VL macro requires at least one argument.
WC:unknown option	An incorrect argument has been given to the .WC macro {12.4} .
<p>Formatter Error Messages</p> <p>Most messages issued by the formatter are self-explanatory. Those error messages over which the user has some control are listed below. Any other error messages should be reported to the local system support group.</p>	
Cannot do ev	<p>Caused by:</p> <ol style="list-style-type: none"> a. setting a page width that is negative or extremely short b. setting a page length that is negative or extremely short c. reprocessing a macro package (e.g., performing a .so request on a macro package that was already requested on the command line) d. requesting the troff formatter (an option on a document that is longer than ten pages).

TABLE 4.D (Contd)

ERROR MESSAGES

ERROR MESSAGE	DESCRIPTION
Cannot execute <i>filename</i>	Given by the <code>!</code> request if the <i>filename</i> is not found.
Cannot open <i>filename</i>	Indicates one of the files in the list of files to be processed cannot be opened.
Exception word list full	Indicates too many words have been specified in the hyphenation exception list (via <code>.hw</code> requests).
Line overflow	Indicates output line being generated was too long for the formatter line buffer capacity. The excess was discarded. Likely causes for this message are very long lines or words generated through the misuse of <code>\c</code> of the <code>.cu</code> request or very long equations produced by <code>eqn(1)/neqn</code> .
Nonexistent font type	Indicates a request has been made to mount an unknown font.
Nonexistent macro file	Indicates the requested macro package does not exist.
Nonexistent terminal type	Indicates the terminal options refer to an unknown terminal type.
Out of temp file space	Indicates additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing <code>.FE</code> or <code>.DE</code>), unclosed macro definitions (e.g., missing <code>..</code>), or a huge table of contents.
Too many page numbers	Indicates the list of pages specified to the <code>-o</code> formatter option is too long.
Too many number registers	Indicates the pool of number register names is full. Unneeded registers can be deleted by using the <code>.rr</code> request.
Too many string/macro names	Indicates the pool of string and macro names is full. Unneeded strings and macros can be deleted using the <code>.rm</code> request.
Word overflow	Indicates a word being generated exceeds the formatter word buffer capacity. Excess characters were discarded. Likely causes for this message are very long lines, words generated through the misuse of <code>\c</code> of the <code>.cu</code> request, or very long equations produced by <code>eqn(1)/neqn</code> .

Table of Contents

Lex - A Lexical Analyzer Generator

Introduction.....	1
Lex Source.....	3
Lex Regular Expressions.....	3
Operators.....	3
Character Classes.....	4
Arbitrary Character.....	4
Optional Expressions.....	4
Repeated Expressions.....	4
Alternation and Grouping.....	4
Context Sensitivity.....	4
Repetitions and Definitions.....	5
Lex Actions.....	5
Example.....	6
Ambiguous Source Rules.....	7
Lex Source Definitions.....	8
Usage.....	8
UNIX.....	9
GCOS.....	9
TSO.....	9
Lex and Yacc.....	9
Examples.....	9
Left Context Sensitivity.....	11
Character Set.....	12
Summary of Source Format.....	12
Caveats and Bugs.....	13
Acknowledgments.....	13
References.....	13

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

Bell Laboratories

Murray Hill, New Jersey 07974

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Contents

1. Introduction.
2. Lex Source.
3. Lex Regular Expressions.
4. Lex Actions.
5. Ambiguous Source Rules.
6. Lex Source Definitions.
7. Usage.
8. Lex and Yacc.
9. Examples.
10. Left Context Sensitivity.
11. Character Set.
12. Summary of Source Format.
13. Caveats and Bugs.
14. Acknowledgments.
15. References.

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 user in the source specifications given to Lex. The Lex written code 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 asso-

ciates 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 his 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

Source → Lex → yylex

Input → yylex → Output

An overview of Lex

Figure 1

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 there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%  
[ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule.

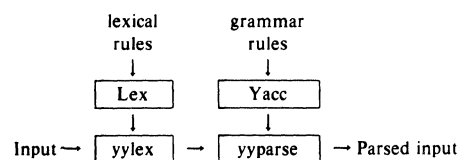
This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or 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]+$ ;  
[ \t]+ printf(" ");
```

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. 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 in Figure 2. 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



Lex with Yacc

Figure 2

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 re-scanning. 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 (in C) or branches of a computed GOTO (in Ratfor). 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*. . . Such backup is more costly than the processing of simpler languages.

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* (see section 3) 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.

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*.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. 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. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex 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.

Character classes. Classes of characters can be specified using the operator pair `[]`. The construction `[ab]` 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 it is desired 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; or

`[^a-zA-Z]`

is any character which is not a letter. The `\` character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

is the class of 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).

Optional expressions. The operator `?` indicates an optional element of an expression. Thus

`ab?c`

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators `*` and `+`.

`a*`

is any number of consecutive *a* characters, including zero; while

`a+`

is 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.

Alternation and Grouping. The operator `|` indicates alternation:

`(ab|cd)`

matches either *ab* or *cd*. Note that parentheses are used 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 *dddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize 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* as explained in section 10. 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>

Start conditions are explained more fully later.

Repetitions and Definitions. The 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, initial % is special, being the separator for Lex source segments.

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 that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, 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.

One of the simplest things that can be done 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 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, one might ask 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 which 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.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yylen* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+ {words++; chars += yylen;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yylen-1]
```

in C or

```
yytext(yylen)
```

in Ratfor.

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, *yyless(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* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[\"]* {
    if (yytext[yytext-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(yytext-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(yytext-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

```
==/[^\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

- 1) *input(c)* which returns the next input character;
- 2) *output(c)* which writes the character *c* on the output; and
- 3) *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 the user can override them and supply private versions. There is another important routine in Ratfor, named *lexshf*, which is described below under "Character Set". 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 strange 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. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes 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, the user should provide a *yywrap* which 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.

In Ratfor all of the standard I/O library routines, *input*,

output, *unput*, *yywrap*, and *lexshf*, are defined as integer functions. This requires *input* and *yywrap* to be called with arguments. One dummy argument is supplied and ignored.

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.

Thus, suppose the rules

```
integer  keyword action ...;
[a-z]+  identifier action ...;
```

to be 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. But 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 *.* operator will not match newline. Thus expressions like *.** stop on the current line. Don't try to defeat this with expressions like *[\\n]+* or equivalents; the Lex 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 *.* 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.

Sometimes the user would like to 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 the user really wants 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, the user 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 *acdb* 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* 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.

6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program 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. There are three classes of such things.

- 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 beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the 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      [TEde][-+]?{D}+
%%
{D}+   printf("integer");
{D}+"."{D}*({E})? |
{D}*."{D}+({E})? |
{D}+{E}
```

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. These possibilities are discussed below under "Summary of Source Format," section 12.

7 Usage.

There are two steps in 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* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file %R.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file %C.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll*. So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

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. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexlib h =	./rc a = lex.yy.r ./lexlib h =

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(clud)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text*. The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
```

```
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

- a. Edit the Ratfor program.
1. Remove all tabs.
2. Change all lower case letters to upper case letters.
3. Convert the file to an 80-column card image file.
- b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.
- c. Compile the Fortran.
- d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. **Warning:** Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

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. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll -lS
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9 Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```

%%
int k;
[0-9]+ {
scanf(-1, yytext, "%d", &k);
if (k%7 == 0)
printf("%d", k+3);
else
printf("%d",k);
}

```

to do just that. The rule `[0-9]+` recognizes strings of digits; `scanf` 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 objected that this program will alter 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, as here:

```

%%
int k;
-?[0-9]+ {
scanf(-1, yytext, "%d", &k);
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[yleng]++;
. |
\n ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
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. To provide a `yywrap` that

never returns true causes 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 upper and lower case 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[0] == '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 `^`. There follow some 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 then adds `'e'-'d'`, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which 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}l{m}{a}{c}{h}  {yytext[0] = + 'r' - 'd';

```

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.

10 Left Context Sensitivity.

Sometimes it is 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:

```

int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; 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;
        }
        }

```

should be adequate.

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 same example as before can be written:

```
%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.

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. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *IHa* while in C the letter *a* is represented as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving 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 next example maps the lower and upper case 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

```
%T
1  Aa
2  Bb
...
26 Zz
27 \n
28 +
29 -
30 0
31 1
...
39 9
%T
```

Sample character table.

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.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input*, *output*, *unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

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

- 1) Definitions, in the form "name space translation".
- 2) Included code, in the form "space code".
- 3) Included code, in the form

```
%{
code
%}
```

- 4) Start conditions, given in the form

```
%S name1 name2 ...
```

- 5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

- 6) A language specifier, which must also precede any rules or included code, in the form “%C” for C or “%R” for Ratfor.

- 7) 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.
xy	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

13 Caveats and Bugs.

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.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15 References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran*, Software — Practice and Experience, 5, pp. 395-496 (1975).
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM 18, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor*, Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library*, Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

Table of Contents

Yacc: Yet Another Compiler-Compiler

Abstract.....	1
Introduction.....	2
Basic Specifications.....	3
Actions.....	5
Lexical Analysis.....	7
How the Parser Works.....	8
Ambiguity and Conflicts.....	12
Precedence.....	15
Error Handling.....	17
The Yacc Environment.....	19
Hints for Preparing Specifications.....	20
Input Style.....	20
Left Recursion.....	20
Lexical Tie-ins.....	21
Reserved Words.....	21
Advanced Topics.....	22
Simulating Error and Accept in Actions.....	22
Accessing Values in Enclosing Rules.....	22
Support for Arbitrary Value Types.....	22
Acknowledgements.....	24
References.....	25
Appendix A: A Simple Example.....	26
Appendix B: Yacc Input Syntax.....	28
Appendix C: An Advanced Example.....	30
Appendix D: Old Features Supported but not Encouraged.....	35

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

0: 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 a portable dialect of C¹ and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow 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 realivly 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 spècifications. 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.^{2,3,4} Yacc has been extensively used in numerous practical applications, including *lint*,⁵ the Portable C Compiler,⁶ and a system for typesetting mathematics.⁷

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, 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 are ignored except that they may not appear in names or multi-character reserved symbols. 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. Upper and lower case 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
```

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 that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to 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. (See Sections 3, 5, and 6 for much more discussion). 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; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: 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 so 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. Then parse tree can be built by supplying actions such as:

```

expr      :      expr '+' expr
           { $$ = node( '+', $1, $3 ); }

```

in the specification.

The user 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”; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: 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 ylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c ) {
        ...
    case '0':
    case '1':
        ...
    case '9':
        ylval = 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 not be used naively (see Section 7).

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.

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.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.⁸ These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these 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.

4: 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 (see the references for more information). 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 *yyllex* 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.

The *shift* action 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 *yyval* 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 recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an 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 a human-readable 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
  . error

  rhyme goto 1
  sound goto 2

state 1
  $accept : rhyme_$end

  $end accept
  . error

state 2
  rhyme : sound_place

  DELL shift 5
  . error

  place goto 4

state 3
  sound : DING_DONG

  DONG shift 6
  . error

state 4
  rhyme : sound place_ (1)

  . reduce 1

state 5
  place : DELL_ (3)

  . reduce 3

state 6
  sound : DING DONG_ (2)

  . reduce 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 while processing 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 “\$end” in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to 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 spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: 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 that 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 the user rather crude 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 previous parser generators have considered conflicts to be fatal errors. Our experience has suggested 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
      reduce 18
```

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 will be at most reduce action possible in the state, and 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. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references^{2,3,4} might be consulted; the services of a local guru might also be appropriate.

6: 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. %prec 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.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. 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.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yparse* returns the value 0.

The user 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 *yparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a *-ly* argument to the loader. To show the triviality of these default programs, the source is given 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.

9: 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.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

- a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
- b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
- c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
- d. 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.
- e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

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!

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;
%}
... other declarations ...

%%

prog  :    decls stats
        ;

decls :    /* empty */
          {      dflag = 1; }
        |    decls declaration
        ;

stats :    /* empty */
          {      dflag = 0; }
        |    stats statement
        ;

... other rules ...

```

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 can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user 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 user can make a stab at it, using the mechanism described in the last 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. There are

powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yterror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent  :    adj noun verb adj noun
        { look at the sentence . . . }
      ;

adj   :    THE      { $$ = THE; }
      |    YOUNG   { $$ = YOUNG; }
      |    ...
      ;

noun  :    DOG      { $$ = DOG; }
      |    CRONE   { if( $0 == YOUNG ){
                    printf( "what?\n" );
                    }
                    $$ = CRONE;
                    }
      ;
      ...
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a \$\$ or \$n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*⁵ will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
    body of union ...
}
```

This declares the Yacc value stack, and the external variables *yyval* and *yyval*, to have type equal to this union. If Yacc was invoked with the `-d` option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable `YYSTYPE` to represent this union. Thus, the header file might also have said:

```
typedef union {
    body of union ...
} YYSTYPE;
```

The header file must be included in the declarations section, by use of `%{` and `%}`.

Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords `%token`, `%left`, `%right`, and `%nonassoc`, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, `%type`, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as `$0` – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between `<` and `>`, immediately after the first `$`. An example of this usage is

```
rule :    aaa { $<intval>$ = 3; } bbb
      {    fun( $<intval>2, $<other>0 ); }
      ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of `$n` or `$$` to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for “one more feature”. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," *Comp. Sci. Tech. Rep. No. 65* (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Appendix A: 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 );
}

```

Appendix B: Yacc Input Syntax

This Appendix 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 decide 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_IDENTIFIER.

```
/* 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 ';'
;

```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is 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 Appendix A; 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 *double*'s. 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 dramatically. 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 vval;
    }

%token <ival> DREG VREG      /* indices into dreg, vreg arrays */
%token <dval> CONST          /* floating point constant */

%type <dval> dexp            /* expression */
%type <vval> 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, $1.hi ); }
      |      DREG '=' dexp '\n'
          { dreg[$1] = $3; }
      |      VREG '=' vexp '\n'

```



```

    error '\n'
    {
        vreg[$1] = $3; }
    {
        yyerrok; }
;

dexp :
CONST
DREG
    {
        $$ = dreg[$1]; }
dexp '+' dexp
    {
        $$ = $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 = $1hi + $3hi;
        $$lo = $1lo + $3lo; }
dexp '+' vexp
    {
        $$hi = $1 + $3hi;
        $$lo = $1 + $3lo; }
vexp '-' vexp
    {
        $$hi = $1hi - $3lo;
        $$lo = $1lo - $3hi; }
dexp '-' vexp
    {
        $$hi = $1 - $3lo;
        $$lo = $1 - $3hi; }
vexp '*' vexp
    {
        $$ = vmul( $1lo, $1hi, $3 ); }
dexp '*' vexp
    {
        $$ = vmul( $1, $1, $3 ); }
vexp '/' vexp
    {
        if( dcheck( $3 ) ) YYERROR;
        $$ = vdiv( $1lo, $1hi, $3 ); }

```

```

|      dexp '/' vexp
|          {      if( dcheck( $3 ) ) YYERROR;
|                  $$ = vdiv( $1, $1, $3 ); }
|      '-' vexp   %prec UMINUS
|          {      $$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 ) ){
        yyval.ival = c - 'A';
        return( VREG );
    }

    if( islower( c ) ){
        yyval.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 ) );
}

```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which 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.

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.

3. Most places where `%` is legal, backslash `"\"` may be used. In particular, `\\` is the same as `%%`, `\left` the same as `%left`, etc.
4. 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`

5. Actions may also have the form

`= { . . . }`

and the curly braces can be dropped if the action is a single C statement.

6. C code between `%{` and `%}` used to be permitted at the head of the rules section, as well as in the declaration section.

Table of Contents

Uucp Implementation Description

Abstract.....	1
Introduction.....	2
Uucp - UNIX to UNIX File Copy.....	2
Type 1.....	3
Type 2.....	3
Type 3.....	4
Type 4 and Type 5.....	4
Uux - UNIX to UNIX Execution.....	4
Uucico - Copy In, Copy Out.....	5
Scan For Work.....	6
Call Remote System.....	6
Line Protocol Selection.....	7
Work Processing.....	7
Conversation Termination.....	8
Uuxqt - Uucp Command Execution.....	8
Command Execution.....	8
Uulog - Uucp Log Inquiry.....	8
Uuclean - Uucp Spool Directory Cleanup.....	8
Security.....	9
Uucp Installation.....	9
Uucp.h Modification.....	10
Makefile Modification.....	10
Compile the System.....	10
Files Required for Execution.....	10
L-devices.....	10
L-dialcodes.....	10
Login/System Names.....	11
Userfile.....	11
L.sys.....	12
Administration.....	13
SQFILE - Sequence Check File.....	13
TM - Temporary Data Files.....	13
LOG - Log Entry Files.....	14
STST - System Status Files.....	14
LCK - Lock Files.....	14
Shell Files.....	14
Login Entry.....	15
File Modes.....	15

Uucp Implementation Description

D. A. Nowitz

ABSTRACT

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current (second) implementation of uucp.

This document is for use by an administrator/installer of the system. It is not meant as a user's guide.

October 31, 1978

Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.¹ This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. 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* are directions for UNIX command executions which involve the resources of one or more systems.

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

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

The secondary programs 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 remainder of this paper will 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. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

```
uucp [ option ] ... source ... destination
```

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

- d Make directories when necessary for copying the file.

†UNIX is a Trademark of Bell Laboratories.

¹ M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

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

The following options are used primarily for debugging:

- r Queue the job but do not start *uucico* program.
- sdir* Use directory *dir* for the spool directory.
- xnum* *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 special shell characters such as “**[/]*”. If a source argument has a *system-name!* prefix for a remote system, the file name 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 “.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. For names with partial path-names, 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 the 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 special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the —r option was specified).

Type 1

A *cp* command is used to do the work. The —*d* and the —*m* options are not honored in this case.

Type 2

A one line *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.)

- [1] R

- [2] The full path-name of the source or a `~user/path-name`. The `~user` part will be expanded on the remote system.
- [3] 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.
- [4] The user's login name.
- [5] A `"-"` followed by an option list. (Only the `-m` and `-d` options will appear in this list.)

Type 3

For each source file, a *work file* is created and 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).

Type 4 and Type 5

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

2. 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 the *uux* command is

```
uux [ - ] [ option ] ... command-string
```

where the command-string is made up of one or more arguments. All special shell characters such as `"<>|"` must be quoted 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. All arguments which do not contain a `"!"` will not be treated as files. (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, essentially for debugging, are:

- `-r` Don't start *uucico* or *uuxqt* after queuing the job;
- `-xnum` 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* which 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 using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). 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.

User Line

U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

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.

Standard Input Line

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.

Standard Output Line

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 - the use of ">>" is not implemented.)

Command Line

C command | arguments | ...

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 is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

3. Uucico - Copy In, Copy Out

The *uucico* program will perform 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,
- b) by one of the *uucp*, *uux*, *uuxqt* or *uucico* programs,
- c) directly by the user (this is usually for testing),
- d) by a remote system. (The *uucico* program should be specified as the “shell” field in the “/etc/passwd” file for the “uucp” logins.)

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 no system name is specified (*-s* option 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.

The *uucico* program is generally started by another program. There are several options used for execution:

- r1* Start the program in *MASTER* mode. This is used when *uucico* is started by a program or “cron” shell.
- ssys* 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.

The following options are used primarily for debugging:

- ddir* Use directory *dir* for the spool directory.
- xnum* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where:

- Type* is an upper case letter, (*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.

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* (files with prefix “C.”). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call 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, *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.

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.

Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "-r1" *uucico* option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

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.

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 "OO" 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.

4. 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 file* is described in the "Uux" section above.

Command Execution

The execution is accomplished by executing 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.

5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

- ssys* Print entries where *sys* is the remote system name;
- uuser* 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.

6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

- ddir* The directory to be scanned is *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.)

- *nhours* Change the aging time from 72 hours to *hours* hours.
- *ppre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
- *xnum* This is the level of debugging output desired.

7. 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.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- 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.)

8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name*.

The four directories are:

- | | |
|----------------|--|
| <i>lib</i> | (<i>/usr/src/cmd/uucp</i>) This directory contains the source files for generating the <i>uucp</i> system. |
| <i>program</i> | (<i>/usr/lib/uucp</i>) This is the directory used for the executable system programs and the system files. |
| <i>spool</i> | (<i>/usr/spool/uucp</i>) This is the spool directory used during <i>uucp</i> execution. |
| <i>xqtdir</i> | (<i>/usr/spool/uucp/.XQTDIR</i>) This directory is used during execution of <i>execute files</i> . |

The names given in parentheses above are the default values for the directories. The italicized named *lib*, *program*, *xqtdir*, and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

makefile modification

There are several *make* variable definitions which may need modification.

- INSDIR This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if “make cp” is used after the programs are compiled.
- IOCTL This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard “C” library; the statement “IOCTL=ioctl.o” is required in this case.
- PKON The statement “PKON=pkcon.o” is required if the packet driver is not in the kernel.

Compile the system The command

make

will compile the entire system. The command

make cp

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in “/usr/bin”. The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where;

- line is the device for the line (e.g. cul0),
- call-unit is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number “0” in this field.),
- speed is the line speed.

The line

cul0 cua0 300

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

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 dial-seq

where;

abb is the abbreviation,
dial-seq is the dial sequence to call that location.

The line

py 165-

would be set up so that entry py7777 would send 165-7777 to the dial-unit.

LOGIN/SYSTEM NAMES

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

USERFILE

This file contains user accessibility information. It specifies four types of constraint;

- [1] which files can be accessed by a normal user of the local machine,
- [2] which files can be accessed from a remote computer,
- [3] which login name is used by a particular remote computer,
- [4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login,sys [c] path-name [path-name] ...

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,
path-name is a path-name 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 path-names 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 path-names 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
```

allows any user to transfer files beginning with “/usr” but the user with login *root* can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name

The name of the remote system.

time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800–1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800–1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

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.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and 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.

9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE — sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

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; the *uuclean* program is useful in this regard. The command

```
uuclean -pTM
```

will remove all *TM* files older than three days.

LOG – log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

```
uulog
```

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new *LOG* entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

STST – system status files

These files are created in the *spool* directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to 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.

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.

Shell Files

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” file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

```
program/uulog
program/uucico -r1
```

Note that the “-r1” option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program/uuclean -pTM -pC. -pD.  
program/uuclean -pST -pLCK -n12
```

can be used. Note the “-n12” option causes the *ST* and *LCK* files older than 12 hours to be deleted. 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 *LOGFILES*. A shell like

```
cp spool/LOGFILE spool/o.LOGFILE  
rm spool/LOGFILE
```

can be used.

Login Entry

One or more logins should be set up for *uucp*. Each of the “/etc/passwd” entries should have the “*program/uucico*” as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of *uucp* (*uucico*) only.

File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the “setuid” bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

Table of Contents

Using the System Console with HP9000 Series 200 Computers

Using the Internal Terminal Emulator	1
Character Entry Group	3
Numeric Pad Group	3
Display Control Group	4
Setting and Clearing Tab Stops	4
Cursor Control	4
Edit Group	8
Function Key Group	9
Defining User Keys Locally	11
Defining User Keys Programmatically	13
Controlling the Function Key Labels Programmatically	13
System Control Group	14
The Display	16
Memory Addressing Scheme	16
Row Addressing	16
Column Addressing	17
Cursor Sensing	17
Absolute Sensing	17
Relative Sensing	17
Cursor Positioning	17
Screen Relative Addressing	18
Absolute Addressing	18
Cursor Relative Addressing	19
Combining Absolute and Relative Addressing	20
Display Enhancements	21
Raster Control	21
Accessing Color (HP 9000 Model 236 with color video)	22
Selecting a Pen (Color Pair)	22
Changing Pen Definitions	22
Configuring the ITE	26
Configuration Function Keys	26
Terminal Configuration Menu	26
Description of Fields	27
Changing the Fields	32
ITE Escape Sequences	33
Sequence Types	34
Keyboard Diagrams for Other Languages	37

Using the System Console with HP9000 Series 200 Computers

Using the Internal Terminal Emulator

The Internal Terminal Emulator consists of “device driver” code contained in the HP-UX kernel and associated with the built-in keyboard and display on the Series 200 Models 226 and 236 with memory management. It also drives the external keyboard and CRT for Series 200 Model 220 computers when the keyboard/HP-IB and composite video interfaces are used for the System Console.

For the remainder of this article, the Series 200 Models 220, 226 and 236 when mentioned have memory management. Memory management means your computer contains a central processing unit designed to run the HP-UX system. To determine if you have memory management, look on the back of your computer for one of the following product numbers:

- 9920U (Model 220)
- 9826U (Model 226)
- 9836U (Model 236)
- 9836CU (Model 236 with color video)

The **system console** is a keyboard and display (or terminal) given a unique status by HP-UX and associated with the special (device) file `/dev/console`. All boot ROM error messages, HP-UX system error messages, and certain system status messages are sent to the system console. Under certain conditions (for example, the single-user state), the system console provides the only mechanism for communicating with HP-UX.

The HP-UX operating system assigns the system console function according to a prioritized search sequence when the HP-UX kernel gains control during the boot-up process. When a given interface board or the Internal Terminal Emulator (ITE) is assigned the system console function, the terminal associated with that interface board (or the keyboard and display associated with the ITE) becomes the physical system console. HP-UX's search for a system console terminates as soon as one of the following conditions is met:

1. An HP 98626A Serial Interface board or HP 98628A Datacomm Interface board that has its “remote bit” set¹ is installed in the computer. If this condition is met and an ITE is present, the ITE is assigned the special (device) file `/dev/tty00` and is considered to be the first non-system console terminal connected to HP-UX. (If no ITE is present, `/dev/tty00` is available for other assignment.)
2. The appropriate hardware (associated with an ITE) is present. This is the general case with the Series 200 Models 226 and 236.

In the case of multiple occurrences, the HP 98626A Serial Interface board or HP 98628A Datacomm Interface board with the lowest select code is chosen to be the system console.

If none of the above conditions are met, no system console exists. While HP-UX tolerates this, you cannot functionally use HP-UX without a system console.

¹ On the HP 98626A Serial Interface board the remote bit is set by cutting a jumper as described in the installation manual supplied with your computer. On the HP 98628A Datacomm Interface board the remote bit is set by setting a switch on the board as described in that board's installation manual.

Note

To install the HP-UX system it is necessary to communicate with the boot ROM. Because the boot ROM will not use a terminal connected to an HP 98628A Datacomm Interface Board as its display, HP-UX must be installed using either the computer's ITE hardware or an HP 98626A Serial Interface board.

For a complete list of Hewlett-Packard terminals supported by HP-UX, see the "Supported Peripherals" appendix located in your *HP-UX System Administrator Manual*.

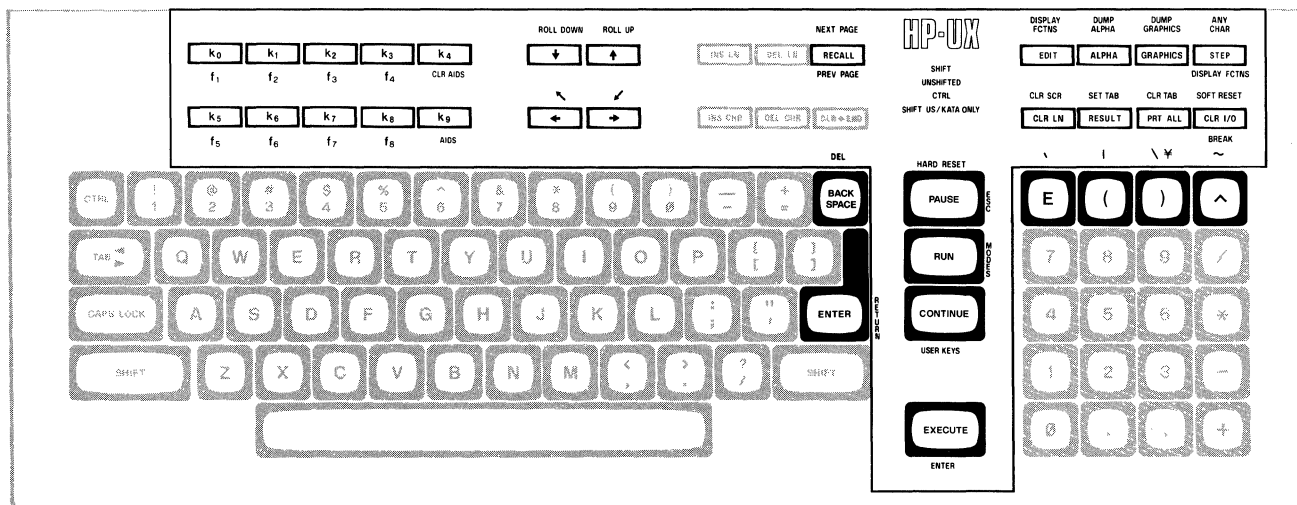
The keyboard overlay supplied with your Series 200 computer allows you to convert it for use as an HP-UX system console keyboard. The key labels on the overlay are color-coded as follows:

- Shifted keys are light blue with the exception of four US/KATA keys. The US/KATA keys and their functions are covered in the "Function Key Group" section. On the overlay, the keys are labeled from left to right: ` , | , \ , and ~ ;
- Unshifted keys are dark brown;
- Control keys are orange;
- Shifted US/KATA keys are dark blue.

The Series 200 computers keyboard is divided into six functional groups:

- Character Entry Group,
- Numeric Pad Group,
- Display Control Group,
- Editing Group,
- Function Key Group,
- Systems Control Group.

These key groups are discussed next.



US ASCII Keyboard and Overlay

Character Entry Group

The character entry keys are arranged like a typewriter, but have some added features.

SHIFT

gives you uppercase letters when you are typing in lowercase (caps lock off). When you are typing in upper case (caps lock on) , this key will give you lower case letters.

CAPS LOCK

sets the unshifted keyboard to either upper case (the power-on default) or lower case (normal typewriter operation) letters.

ENTER

R
E
T
U
R
N

sends the ReturnDef sequence to the computer (the default is to send CR). When a program is running, this key is used to input information requested by the computer.

TAB

sends a tab character (CTRL-I) to the computer.

CTRL

provides access to the standard ASCII control characters.

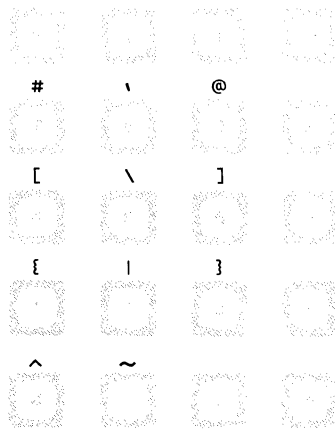
BACK SPACE

sends the back space character (CTRL-H) to the computer in the remote mode, and in the local mode it moves the cursor one space to the left.

Numeric Pad Group

The numeric group of keys is located to the right of the character keys. The layout of the numeric key pad is similar to that of a standard office calculator. These keys are convenient for high-speed entry of numeric data.

The numeric key pad on Series 200 computer also provides the non-US ASCII keyboard user with a few of the character keys not found on their keyboard. These characters are accessed by holding the shift key down and pressing the appropriate numeric key, as shown in the diagram below.



Additional Numeric Key Pad Characters

Display Control Group

The display control group consists of the keys that control the location of the cursor on the display. Each display control key and its function is described in the sections that follow.

Setting and Clearing Tab Stops

You can define and delete a series of tab stops by using the following tab functions.

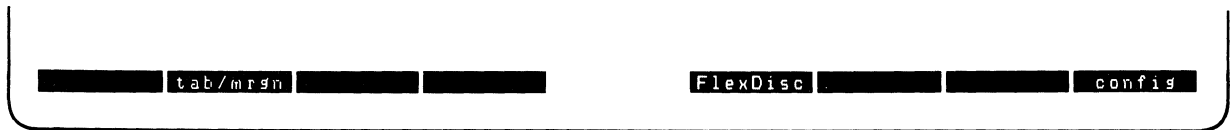
SET TAB
RESULT sets tab stops. To set a tab stop move the cursor to the desired location, hold the **SHIFT** key down, and press **SET TAB**.

CLR TAB
PRT ALL deletes tab stops. To delete tab stops, move the cursor to the tab position which you want to remove, hold the **SHIFT** key down, and press **CLR TAB**.

Cursor Control

To use the knob (cursor control wheel) and arrow keys, either have the **transmit functions mode** disabled or be in the **vi** or **ex** editor. The **transmit functions mode** specifies whether escape code functions are executed locally at the terminal (ITE) or transmitted to the host computer. When the system is shipped the default for this mode is: NO. In the **vi** or **ex** editor, the arrow keys and knob can be used as described in this section. To disable the **transmit functions mode**:

press: **k9** (labeled AIDS on the overlay)



press: **k8** (config)



press: **k5** (terminal)

When a screen display similar to the one shown below appears, press the **TAB** key until the cursor is positioned just after the **XmitFunctn(A)**.

```

                                TERMINAL CONFIGURATION
Language      USASCII
ReturnDef    NO
LocalEcho    OFF      CapsLock  OFF      Ascii 8 Bit NO
XmitFunctn(A) NO      InHEolWrP(C) NO

SAVE CFG  NEXT  PREVIOUS  DEFAULT  DSPY FN  config
```

If the word following the label is NO, then the transmit functions mode is not active, so

press: **k8** (config)

to return to the HP-UX environment. If the word following the **XmitFunctn(A)** label is a YES, then

press: **k2** (NEXT)

This turns off the transmit function mode.







press: **k1** (SAVE CFG)

This returns you to the ITE environment.

When the transmit functions mode is off, there are four keys used to change the location of the cursor in the shell environment:

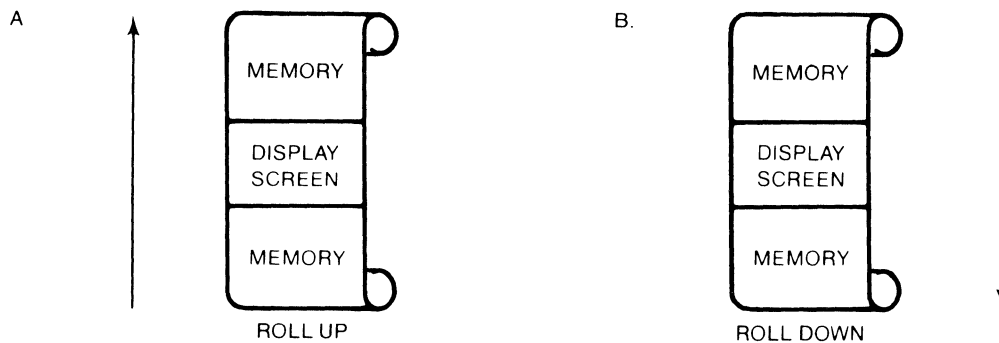
- ↑** moves the cursor up in the output area of the display screen.
- ↓** moves the cursor down in the output area of the display screen.
- ←** moves the cursor to the left in the output area of the display screen.
- moves the cursor to the right in the output area of the display screen.

These same keys when used with the **SHIFT** key enable you to scroll up and down through the screen display, and to move the cursor to its upper or lower home position. These keys are listed as follows:

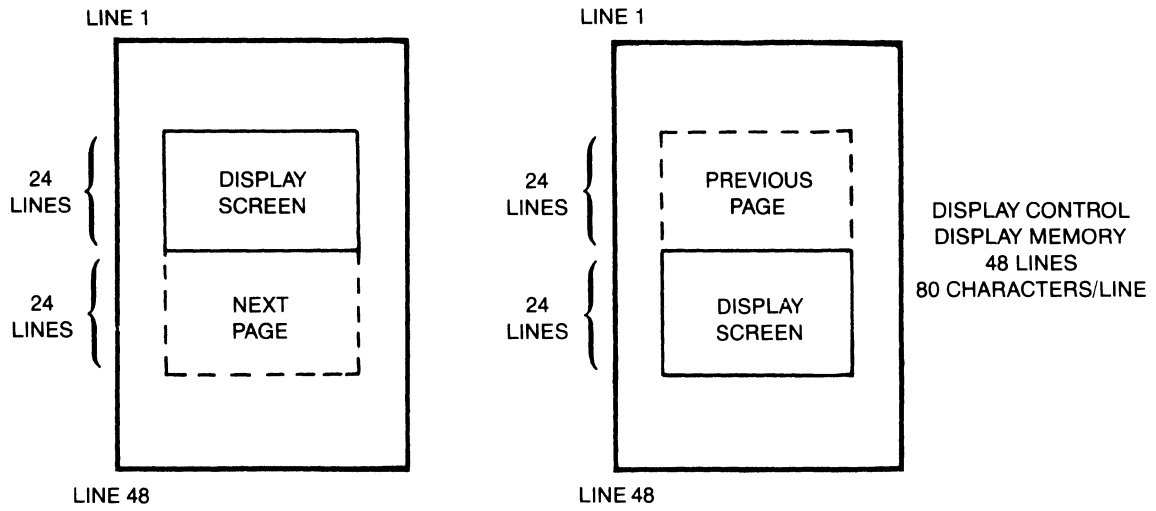
- ROLL DOWN
shift-  scrolls the screen display downward.
- ROLL UP
shift-  scrolls the screen display upward.
- 
shift-  moves the cursor to its upper home position.
- 
shift-  moves the cursor to its lower home position.

Another method used to move the cursor is the **knob**. Rotating the knob either clockwise or counter-clockwise moves the cursor horizontally. Rotating the knob while pushing the **SHIFT** key moves the cursor vertically until the top or bottom of the display screen is reached, at that time the screen display rolls up or down in display to the next page or previous page.

What is the next page or previous page? Data in display memory can be accessed (displayed on the screen) in blocks that are known as “pages”. A page consists of 24 (23 for a Model 226) lines of data. The current page is that sequence of lines which appears on the screen at any given time. The **previous page** is the preceding 24 lines in display memory. The **next page** is the succeeding 24 lines in display memory. This concept, along with the concept of rolling data through the display screen and memory, are shown in the following illustrations.



The “Roll Up and Roll Down” Functions



Previous Page and Next Page Concepts

Edit Group

The edit group consists of the keys that allow you to modify the data presented on the screen. However, the edited data cannot be read back by the system. Typically these features are used to modify text within a program or file, to view data which has scrolled out of the screen window and to clear the screen.

To use these features in the vi editor, you should have an **.exrc** file in your **\$HOME** directory which maps these keys to their function using escape code sequences. Reference to this file is made in the *HP-UX System Administrator Manual* and in "The Vi Editor" selected article.

The **RECALL** key has been redefined as follows:

RECALL moves you back a page within your text.
PREV PAGE

NEXT PAGE moves you forward a page within your text.
shift - **RECALL**

All line edit keys and character edit keys function as stated below:

CLR END clears the characters in a line from the present cursor position to the end of the line.

INS LN causes the line containing the cursor and all text lines below it to move down one line, and a blank line to be inserted in the row containing the cursor. The cursor will move to the left margin of the blank line.

DEL LN deletes the line containing the cursor from display memory. All text lines below this line will roll up one row, and the cursor will move to the left margin.

CLR LN performs the same function as CLR END.

INS CHR sets the insert mode, allowing you to insert characters to the left of the cursor.

DEL CHR deletes the character at the cursor.

DEL sends the DEL character to the computer.
shift - **BACK SPACE**

Function Key Group

In the upper left-hand corner of your keyboard next to the knob are a set of ten function keys (HP-UX recognizes only eight function keys: **f1** through **f8**). The remaining keys are used as "AIDS" keys which will be discussed later on in this section. The functions performed by these keys change dynamically as you use the computer. At any given time the applicable function labels for these keys appear across the bottom of the display screen.

The function key group also includes four US/KATA keys which are accessed using the numeric key pad. These keys are very useful when using HP-UX and they have been given the following names:

shift- E	accent grave.
shift- (vertical bar.
shift-)	backslash.
shift- ^	tilde.

The remainder of this section covers the relationship of the eight function keys, **k1** through **k8** , to the overlay functions: MODES, AIDS and USER KEYS.

RUN	M O D E S	allows access to one of the modes described in the following sections. An example of the function key display is given:
------------	-----------------------	---



You may use these function keys to enable and disable various terminal operating modes. Each defined mode selection key alternately enables, and disables a particular mode. When the mode is enabled, an asterisk (*) appears in the associated key label on the screen, as seen in the REMOTE key label above.

When the **remote mode** is enabled and a key is pressed, the terminal transmits the associated ASCII code to HP-UX. In **local mode** (remote mode is disabled), when a character key is pressed, the associated character is displayed at the current cursor position on the screen (nothing is transmitted to HP-UX).

When the **auto line feed mode** is enabled, an ASCII line feed control code is automatically appended to each ASCII carriage-return control code generated through the keyboard. ASCII carriage-return control codes can be generated through the keyboard in any of the following ways:

- By pressing **RETURN**.
- By holding down CTRL and pressing **M**.
- By pressing any of the user keys **f1** through **f8**, provided that a carriage-return code is included in the particular key definition.

When the **display functions mode** is enabled, the terminal (ITE) displays ASCII control codes, and escape sequences but does not execute them.

kg provides another menu in the display, showing three general control keys:
AIDS



tab/mrgn pressing this softkey provides another menu with: SET TAB, CLR TAB, and CLR TABS. The first two softkey functions were previously defined in the display control group section. The last softkey [CLR TABS] when pressed clears all tab stops currently set.

FlexDisc pressing this softkey provides another menu with these labels: fd.1 and fd.0. Whenever this symbol * appears in the label block the internal disc drive 1 (left drive) or 0 (right drive) is in use.

config pressing this softkey provides another menu with only the softkey [terminal] in it. If you press this softkey, you get the **TERMINAL CONFIGURATION** display on your screen as shown in the display control group section of this article.

CLR AIDS clears the screen display of the function key labels. The user function keys, however, are still enabled.

k4
EXECUTE
ENTER not implemented.

CONTINUE
USER KEYS displays eight function keys which can be defined either locally by the user or remotely by a program executed in a host computer. By “defined” it is meant:

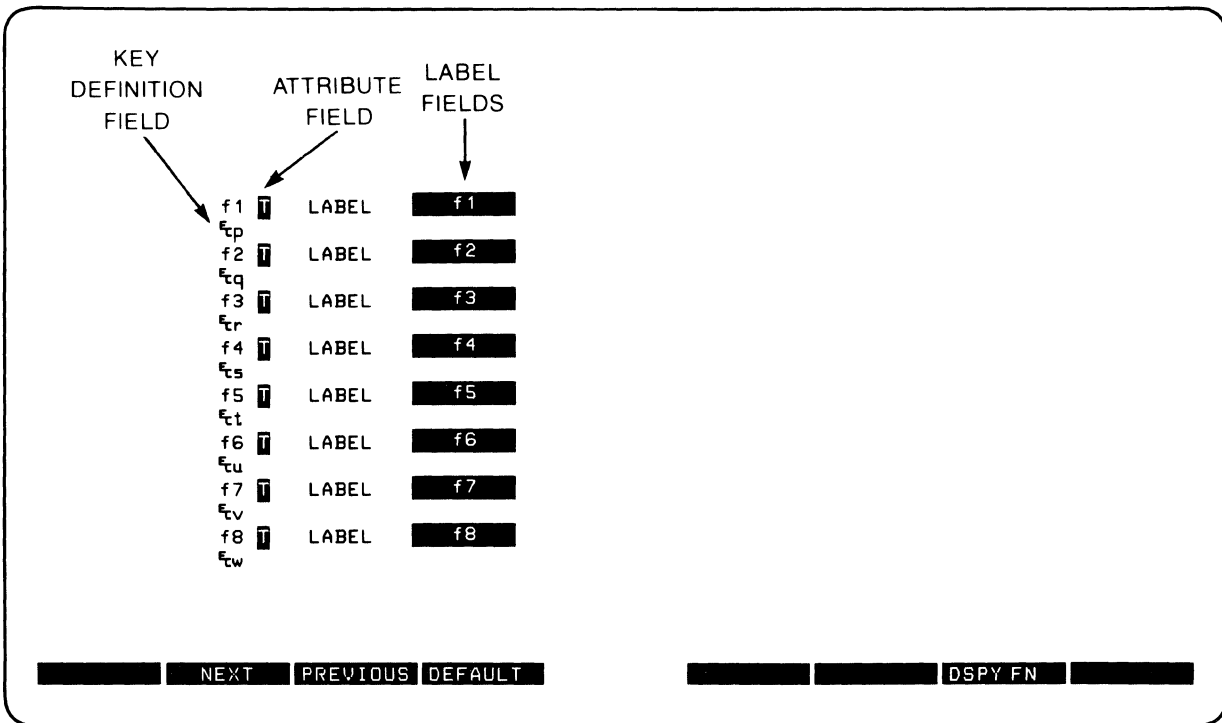
- You can assign to each key a string of ASCII alphanumeric characters and/or control codes (such as carriage return or line feed).
- You can specify each key’s operational attribute: whether its key definition is to be executed locally at the terminal, transmitted to the computer, or both.
- You can assign to each key an alphanumeric label (up to 16 characters) which, in **user keys mode**, is displayed across the bottom of the screen.

Defining User Keys Locally

When defining a key from the keyboard, the key content may include explicit escape sequences (entered using display functions mode) that control or modify the ITE's operation.

The definition of each user key may contain up to 80 characters (alphanumeric characters, ASCII control characters, and explicit escape sequence characters).

To define **USER KEYS** locally (from the keyboard), press the **SHIFT** - **CONTINUE** (USER KEYS) keys simultaneously. The following user keys menu will appear:



This menu contains the default values for all of the fields. If your screen does not contain the default values as shown and you want them set and displayed press **f4** (DEFAULT).

The menu contains a set of fields that you access using the **TAB** key.

For each user key the menu contains three unprotected fields:

ATTRIBUTE FIELD — This one character field always contains an uppercase L, T, or N signifying whether the content of the particular user key is to be:

- L Executed locally only.
- T Transmitted to the host computer only.
- N Treated in the same manner as the alphanumeric keys. If the ITE is in local mode, the content of the key is executed locally. If the ITE is in the remote mode and LocalEcho is disabled (OFF), the content of the key is transmitted to the host computer. If the ITE is in remote mode and local echo is enabled (ON), the content of the key is both transmitted to the host computer and executed locally.

The alphanumeric keys are disabled when the cursor is positioned in this field. You change the content of this field by pressing (NEXT) or (PREVIOUS).

LABEL FIELD — This field eight character field to the right of the word "LABEL" allows you to supply the user key's label. When the ITE is in user keys mode, the key labels are displayed from left to right in ascending order across the bottom of the screen.

KEY DEFINITION FIELD — The entire line (80 characters) immediately below the attribute and label field is available for specifying the character string that is to be displayed, executed, and/or transmitted whenever the particular key is physically pressed.

When entering characters into the key definition field you may use the display functions mode. Note that this implementation of display functions mode is separate from that which is enabled/disabled via the mode selection keys. When entering the label and key definition you may access display functions mode by way of function key (DSPY FN) for the Model 236 and for the Model 226 use function key .

The (RETURN) key can be used to include carriage return (^CR) codes (with display functions mode enabled) in key definitions. If auto line feed mode is also enabled, the (RETURN) key will generate a ^CR ^LF, otherwise it is considered a cursor movement key.

When the user keys menu is displayed on the screen you may use the , and keys for editing the contents of the label and key definition fields.

When you are finished defining all the desired keys, press the (AIDS), (MODES) or (USER KEYS) key (in all three cases the user keys menu disappears from the screen). When you press (USER KEYS), the defined user key labels are displayed across the bottom of the screen and the through user keys, as defined by you, are enabled.

Defining User Keys Programmatically

From a program executing in a host computer, you can define one or more keys using the following escape sequence format:

```
^c&f <attribute><label length><string length><label><string>
```

where:

<attribute> =

0a : normal (0 is the default)

1a : local only

2a : transmit only

<key> = 1-8k : f1-f8, (1 is the default)

respectively

<label length> = 0-16d (0 is the default)

<string length> = 0-80L (1 is the default)

(-1 causes field to be erased)

The <attribute>, <key>, <label length>, and <string length> parameters may appear in any sequence but must precede the label and key definition strings. You must use an uppercase identifier (A, K, D, or L) for the final parameter and a lowercase identifier (a, k, d, or l) for all preceding parameters. Following the parameters, the first 0 through 16 characters, as designated by <label length>, constitute the key's label; however, only the first 8 characters are recognized. The next 0 through 80 characters, as designated by <string length>, constitute the key's definition string. The total number of displayable characters (alphanumeric data, ASCII control codes such as ^R and ^F, and explicit escape sequence characters) in the label string must not exceed 16, and in the definition string must not exceed 80.

Example: Assign login as the label and TOM as the definition for the user key. The key is to have attribute "N".

```
^c&f5k5d4LloginTOM^R^c&jB
```

After issuing the foregoing escape sequence from your program to the terminal, the portion of the user keys menu is as follows:

```
f5 N LABEL login
TOM^R
```

If the transmit only attribute (2) is designated, the particular user key will have no effect unless the terminal is in remote. The ^c&jB sequence turns on the user labels.

Controlling the Function Key Labels Programmatically

From a program executing in a host computer, you can control the function key labels display as follows by using escape sequences:

- You can remove the key labels from the screen entirely (this is the equivalent of pressing (CLR AIDS)).
- You can enable the mode selection keys (this is the equivalent of pressing the (MODES) key).
- You can enable the user keys (this is the equivalent of pressing the (USER KEYS) key).

The escape sequences are as follows:

- Esc & J @ Enables the user keys and remove all key labels from the screen.
- Esc & J A Enables the modes key.
- Esc & J B Enables the user keys.

System Control Group

These keys are located in the upper-right corner of your keyboard. They control system functions related to the display, printer and editing operations.

- PAUSE** ESC generates special ASCII control character number 27 (escape character).
- EDIT** not implemented.
- DISPLAY FCTNS
shift - **EDIT** enables **display functions mode** as defined in the function key group. Pressing the **DISPLAY FCTNS** key again cancels the **display functions mode**.
- ALPHA**
and
GRAPHICS These commands work together to control the Alpha and Graphic displays. The following table shows how these commands work.

If	And you	Result
Alpha ON Graphics OFF	Press Alpha	No change in display
Alpha ON Graphics OFF	Press Graphics	Both displays on screen
Alpha ON Graphics ON	Press Alpha	Graphic display turned off
Alpha ON Graphics ON	Press Graphics	No change in display
Alpha OFF Graphics ON	Press Alpha	Both displays on screen
Alpha OFF Graphics ON	Press Graphics	Alpha display turned off
Alpha ON		

- DUMP GRAPHICS
shift- **GRAPHICS** not implemented.
- DUMP ALPHA
shift- **ALPHA** not implemented.
- ANY CHAR
shift- **STEP** causes the next three characters typed (must be integers) to be interpreted as the decimal specifier of an HP extended ASCII character.
- STEP** not implemented.
- CLR LN** was defined in the Edit Group.
- CLR SCR
shift- **CLR LN** clears the entire alpha portion of the screen display.

<p>RESULT</p>	not implemented.
<p>PRT ALL</p>	not implemented.
<p>SET TAB shift- RESULT</p>	sets a tab at the current cursor position. Tabs are in effect in the alpha display until cleared by CLR TAB .
<p>CLR TAB shift- PRT ALL</p>	clears a tab previously set at the current cursor position.
<p>CLR I/O</p>	not implemented.
<p>SOFT RESET shift- CLR I/O</p>	<p>does the following:</p> <ul style="list-style-type: none"> ● Sounds the computer's beeper. ● Disables display functions mode (if enabled). ● Halts any datacomm transfers currently in progress, clears the datacomm buffers, and reinitializes the datacom port according to the appropriate power-on datacomm configuration parameters. <p>The data on the screen, all terminal operating modes (except display functions mode), and all active configuration parameters are unchanged.</p>
<p>HARD RESET shift- PAUSE</p>	<p>has the same effect as turning the computer's power off and then back on.</p> <p>A hard reset does the following:</p> <ul style="list-style-type: none"> ● Sounds the computer's beeper. ● Clears all of alphanumeric memory. ● Resets the terminal configuration menu parameters to their power on values. ● Resets certain operating modes and parameters as follows: <ul style="list-style-type: none"> - Disables display functions mode, and caps mode. - Turns off the insert character edit function. - Resets the user keys to default values. - Turns on the alphanumeric display. - Resets color pairs to their default values.
<p>BREAK</p>	creates an interrupt signal (SIGINT) which is sent to all processes within your terminal (ITE). For information on this signal read the sections SIGNAL(2) and TTY(4) in your <i>HP-UX Reference</i> . To learn how to use this signal in a shell script read "Shell Programming" in your <i>HP-UX Selected Articles</i> .

The Display

The Internal Terminal Emulator's (ITE's) display has many features of its own, video highlights (such as inverse video and blinking), raster control, cursor sensing and addressing, and color highlight control (for Model 236 computers equipped with a color display). These functions are accessed only through escape sequences and are discussed in the sections that follow. As you read this section, note the last letter in an extended escape sequence is always capitalized. An extended escape sequence consist of the escape-code character followed by at least two subsequent characters.

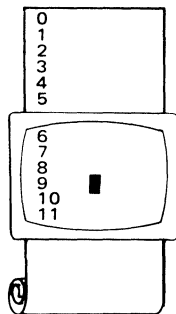
Memory Addressing Scheme

Display memory positions can be addressed using absolute or relative coordinate values. On the Model 220 and 236, display memory is made up of 80 columns (0 thru 79) and two 24 line pages (0 thru 47) with 80 characters per line. A Model 26 upgraded for HP-UX use has a display memory made up of 50 columns (0 thru 49) and two 23 line pages (0 thru 45) with 50 characters per line. The types of addressing available are absolute (memory relative), screen relative, and cursor relative.

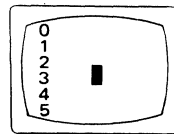
Row Addressing

The figure below illustrates the way that the three types of addressing affect row or line numbers. The cursor is shown positioned in the fourth row on the screen. Screen row 0 is currently at row 6 of display memory. In order to reposition the cursor to the first line of the screen the following three destination rows could be used:

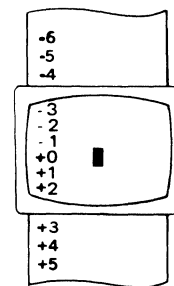
- Absolute: row 6
- Screen Relative: row 0
- Cursor Relative: row -3



a.) Absolute: row 6



b.) Screen Relative: row 0



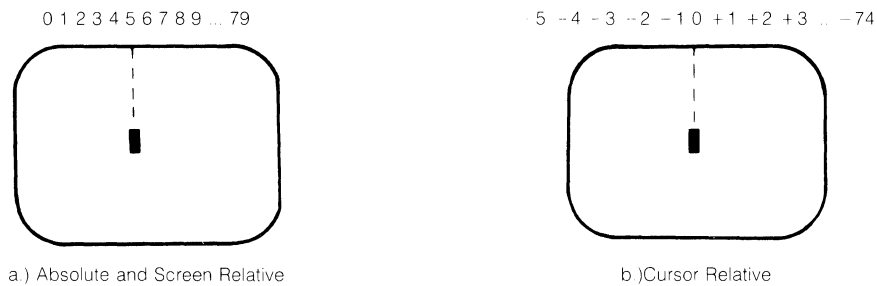
c.) Cursor Relative: row -3

Row Addressing

Column Addressing

Column addressing is accomplished in a manner similar to row addressing. There is no difference between screen relative and absolute addressing. The figure below illustrates the difference between absolute and cursor relative addressing. The cursor is shown in column 5.

Whenever the row or column addresses exceed those available, the largest possible value is substituted. In screen relative addressing, the cursor cannot be moved to a row position that is not currently displayed. For example, in the cursor relative portion of the figure on the previous page (showing row addressing), a relative row address of -10 would cause the cursor to be positioned at the top of the current screen (relative to row -3). Column positions are limited to the available screen positions. For example, in the following illustration, the absolute column addressing example shows limits of 0 and 79, while the relative column addressing example shows limits of -5 and $+74$. The cursor cannot be wrapped around from column 0 to column 79 by specifying large negative values for relative column positions.



Column Addressing

Cursor Sensing

The current position of the screen cursor can be sensed. The position returned can be the absolute position in the display memory or the location relative to the current screen position.

Cursor sensing functions only when the "terminal" is in remote mode.

Absolute Sensing

When a program sends the escape sequence $^E_C a$ to the terminal, the terminal returns to the program an escape sequence of the form $^E_C \& a x x x c y y y Y^C_R$, where xxx is the absolute column number and yyy is the absolute row number of the current cursor position. You will later see that this escape sequence is identical to the escape sequence for an absolute move of the cursor.

Relative Sensing

When a program sends the escape sequence $^E_C \backslash$, the terminal returns to the program an escape sequence of the form $^E_C \& a x x x c y y y Y^C_R$ where xxx is the column number of the cursor and yyy is row position of the cursor relative to screen row 0. This escape sequence is identical to the escape sequence for a relative move of the cursor (discussed later in this article).

Cursor Positioning

The cursor can be positioned directly by giving memory or screen coordinates, or by sending the escape codes for any of the keyboard cursor positioning operations.

Screen Relative Addressing

To move the cursor to any character position on the screen, use any of the following escape sequences:

`^c&.a<column number> c <row number>Y`

`^c&.a<row number> v <column number>C`

`^c&.a<column number>C`

`^c&.a<row number>Y`

where: <column number> is a decimal number specifying the screen column to which you wish to move the cursor. Zero specifies the leftmost column.

<row number> is a decimal number specifying the screen row (0 thru 23) to which you wish to move the cursor. Zero specifies the top row of the screen; 23 specifies the bottom row.

When using the escape sequences for screen relative addressing, the data on the screen is not affected (the cursor may only be moved around in the 24 rows and 80 columns currently displayed, thus data is not scrolled up or down).

If you specify only <column number>, the cursor remains in the current row. Similarly, if you specify only <row number>, the cursor remains in the current column.

Example

The following escape sequence moves the cursor to the 20th column of the 7th row on the screen:

```
^c&.a6v19C
```

Absolute Addressing

You can specify the location of any character within display memory by supplying absolute row and column coordinates. To move the cursor to another character position using absolute addressing, use any of the following escape sequences:

`^c&.a<column number> c <row number>R`

`^c&.a<row number> r <column number>C`

`^c&.a<column number>C`

`^c&.a<row number>R`

where: <column number> is a decimal number (0 thru 79) specifying the column coordinate (within display memory) of the character at which you want the cursor positioned. Zero specifies the first (leftmost) column in display memory, 79 the rightmost column.

<row number> is a decimal number (0 thru max) specifying the row coordinate (within display memory) of the character at which you want the cursor positioned. Zero specifies the first (top) row in display memory, max specifies the last. The value of max is specified as:

`[24 (lines/page) X num_page (pages)] - 1`

where num_page is the number of pages of display memory specified by the system configuration. As shipped to you, the configuration dictates that 2 pages of display memory be allocated. Thus, the last row that can be addressed is 47.

When using the above escape sequences, the data visible on the screen rolls up or down (if necessary) in order to position the cursor at the specified data character. The cursor and data movement occur as follows:

- If a specified character position lies within the boundaries of the screen, the cursor moves to that position; the data on the screen does not move.
- If the absolute row coordinate is less than that of the top line currently visible on the screen, the cursor moves to the specified column in the top row of the screen; the data then rolls down until the specified row appears in the top line of the screen.
- If the absolute row coordinate exceeds that of the bottom line currently visible on the screen, the cursor moves to the specified column in the bottom row of the screen; the data then rolls up until the specified row appears in the bottom line of the screen.

If you specify only a <column number>, the cursor remains in the current row. Similarly, if you specify only a <row number>, the cursor remains in the current column.

Example

To position the cursor (rolling the data if necessary) at the character residing in the 60th column of the 27th row in display memory, the escape sequence is:

```
ESC & a 26 r 59 C
```

Cursor Relative Addressing

You can specify the location of any character within display memory by supplying row and column coordinates that are relative to the current cursor position. To move the cursor to another character position using cursor relative addressing, use any of the following escape sequences:

```
ESC & a ±<column number> c ±<row number> R  
ESC & a ±<row number> r ±<column number> C  
ESC & a ±<column number> C  
ESC & a ±<row number> R
```

where: <column number> is a decimal number specifying the relative column to which you wish to move the cursor. A positive number specifies how many columns to the right you wish to move the cursor; a negative number specifies how many columns to the left.

<row number> is a decimal number specifying the relative row to which you wish to move the cursor. A positive number specifies how many rows down you wish to move the cursor; a negative number specifies how many rows up you wish to move the cursor.

When using the above escape sequences, the data visible on the screen rolls up or down (if necessary) in order to position the cursor at the specified data character. The cursor and data movement occur as follows:

- If a specified character position lies within the boundaries of the screen, the cursor moves to that position; the data on the screen does not move.
- If the specified cursor relative row precedes the top line currently visible on the screen, the cursor moves to the specified column in the top row of the screen; the data then rolls down until the specified row appears in the top line of the screen.
- If the specified cursor relative row exceeds the bottom line currently visible on the screen, the cursor moves to the specified column in the bottom row of the screen; the data then rolls up until the specified row appears in the bottom line of the screen.

If you specify only a <column number>, the cursor remains in the current row. Similarly, if you specify only a <row number>, the cursor remains in the current column.

Example

To position the cursor (rolling the data if necessary) at the character residing 15 columns to the right and 25 rows above the current cursor position (within display memory), use the escape sequence:

```
ESC & a + 15 c - 25 R
```

Combining Absolute and Relative Addressing

You may use a combination of screen relative, absolute and cursor relative addressing within a single escape sequence.

For example, to move the cursor (and roll the text if necessary) so that it is positioned at the character residing in the 70th column of the 18th row below the current cursor position, use the escape sequence:

```
ESC & a 69 c + 18 R
```

Next, to move the cursor so that it is positioned at the character residing 15 columns to the left of the current cursor position in the 4th row currently visible on the screen, use the escape sequence:

```
ESC & a - 15 c 3 Y
```

Similarly, to move the cursor (and roll the text up or down if necessary) so that it is positioned at the character residing in the 10th column of absolute row 48 in display memory, use the escape sequence:

```
ESC & a 9 c 47 R
```

Display Enhancements

The terminal includes as a standard feature the following display enhancement capabilities:

- Inverse Video - black characters are displayed against a white background.
- Underline Video - characters are underscored.
- Blink Video - characters blink on and off.

Note

The Model 226 computer doesn't provide display enhancements. The Model 220 computer provides display enhancements if it has an HP 98204A composite video card. The Model 236 computer with color video doesn't have the half bright enhancement.

The display enhancements are used on a field basis. The field can not span more than one line. The field scrolls with display memory. Overwriting a displayable character in a field preserves the display enhancement. The enhancements may be used separately or in any combination. When used, they cause control bits to be set within display memory.

From a program or from the keyboard, you enable and disable the various video enhancements by embedding escape sequences within the data. The general form of the escape sequence is:

$\text{E}_{c \& d} \langle \text{enhancement code} \rangle$

where enhancement code is one of the uppercase letters A through O specifying the desired enhancement(s) or an @ to specify end of enhancement:

Enhancement Character

	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Half-Bright									x	x	x	x	x	x	x	x
Underline					x	x	x	x					x	x	x	x
Inverse Video			x	x			x	x			x	x			x	x
Blinking		x		x		x		x		x		x		x		x
End Enhancement	x															

Note that the escape sequence for "end enhancement" ($\text{E}_{c \& d} @$) or the escape sequence for another video enhancement, ends the previous enhancement.

Raster Control

The terminal provides the ability to enable and disable the alphanumeric display. The escape sequences for these capabilities are:

$\text{E}_{c * d} E$ Turn on the alphanumeric display; enable writing to the alphanumeric display.

$\text{E}_{c * d} F$ Turn off the alphanumeric display; disable writing to the alphanumeric display.

Accessing Color (Series 200 Model 236 with Color Video only)

To access color on the Series 200 Model 236, you must understand some simple terms.

Color pair - two colors which define the foreground color (color of the characters) and the background color, respectively. At least one of the color pair must be black; displaying color on color is not possible. A total of 64 color pairs are possible, but only eight can be displayed at any one time.

Pen # - one of eight predefined color pairs. Pen 0 through pen 7 are initially defined as follows (re-defining a color pair is discribed later):

Pen #	foreground color	background color
0	white	black
1	red	black
2	green	black
3	yellow	black
4	blue	black
5	magenta	black
6	cyan	black
7	black	yellow

Pen #0 is the default pen selected by the terminal when writing to the display.

Pen #7 is always used for displaying the softkey labels.

Selecting a Pen (Color Pair)

By using an escape sequence, you can select a pen number other than pen #0 when writing to the display. Like other display enhancements, pen selection is used on a field basis. The field cannot span more than one line. That is, the pen selection is only active until a new-line character is encountered; then the default pen is re-selected. The escape sequence for selecting a pen is:

`ESC & V n <parameter>`

where n is the pen number you wish to use, and <parameter> is a single character that specifies the action as described below. To select a pre-defined pen number, the necessary <parameter> is **s**. Thus,

`ESC & V 4s`

selects the pre-defined pen number 4.

Changing Pen Definitions

You may change the pre-defined color pair for any of the eight existing display pens. The three primary colors (red, green and blue) are used in various combinations to achieve the desired color.

The combinations of red, green, and blue that define foreground and background colors can be specified in two notations. The first is RGB (Red-Green-Blue), and the second is HSL (Hue-Saturation-Luminosity). The notation must be selected before you can redefine pens (if no notation type is specified, the "terminal" uses the last notation specified, or RGB notation at power-up). To select a notation type, use the $\text{E}_{c\&v}$ escape sequence used above:

$\text{E}_{c\&v} n<\text{parameter}>$

where n is 0 (for RGB) or 1 (for HSL), and <parameter> is the letter **m**. Thus, the sequence

$\text{E}_{c\&v} 1M$

selects HSL notation. It does nothing more.

To specify the quantity of red (hue), green (saturation), and blue (luminosity) to appear in your background and foreground colors, the a, b, c, x, y, and z parameters are used. These parameters have the following meanings:

- a specifies the amount of red (hue) used in the foreground.
- b specifies the amount of green (saturation) used in the foreground.
- c specifies the amount of blue (luminosity) used in the foreground.
- x specifies the amount of red (hue) used in the background.
- y specifies the amount of green (saturation) used in the background.
- z specifies the amount of blue (luminosity) used in the background.

Each a, b, c, x, y, and z parameter specified is preceded by a number in the range 0 through 1, in increments of 0.01. The following table gives the values needed to define the eight principle colors:

Sample RGB/HSL Color Definition Values

R	G	B	Color	H	S	L
0	0	0	Black	X	X	0
0	0	1	Blue	.66	1	1
0	1	0	Green	.33	1	1
0	1	1	Cyan	.5	1	1
1	0	0	Red	1	1	1
1	0	1	Magenta	.83	1	1
1	1	0	Yellow	.16	1	1
1	1	1	White	X	0	1

X = don't care (may be any value between 0 and 1)

The following tables provide algorithms for explicitly defining the ranges of the parameters mentioned in the previous table for the Model 236 computer with color video.

HSL Definition Algorithm

COLOR SELECTED	parm. 1 H RANGE	parm. 2 S RANGE	parm. 3 L RANGE
BLACK	don't care	don't care	< 0.25
WHITE	don't care	< 0.25	>= 0.25
RED	.00- .08	>= 0.25	>= 0.25
YELLOW	.09- .24	>= 0.25	>= 0.25
GREEN	.25- .41	>= 0.25	>= 0.25
CYAN	.42- .58	>= 0.25	>= 0.25
BLUE	.59- .74	>= 0.25	>= 0.25
MAGENTA	.75- .91	>= 0.25	>= 0.25
RED	.92-1.00	>= 0.25	>= 0.25

In the RGB color method, when N represents the largest-valued (most intense) color of the three color specifications, colors are selected as follows:

RGB Definition Algorithm

COLOR SELECTED	parm.1 RED RANGE	parm. 2 GREEN RANGE	parm. 3 BLUE RANGE
BLACK	< .25 or < N/2	< .25 or < N/2	< .25 or < N/2
WHITE	>= .25 and >= N/2	>= .25 and >= N/2	>= .25 and >= N/2
YELLOW	>= .25 and >= N/2	>= .25 and >= N/2	< .25 or < N/2
GREEN	< .25 or < N/2	>= .25 and >= N/2	< .25 or < N/2
CYAN	< .25 or < N/2	>= .25 and >= N/2	>= .25 and >= N/2
BLUE	< .25 or < N/2	< .25 or < N/2	>= .25 and >= N/2
MAGENTA	>= .25 and >= N/2	< .25 or < N/2	>= .25 and >= N/2
RED	>= .25 and >= N/2	< .25 or < N/2	< .25 or < N/2

One final parameter, **i**, is needed. It is used to assign a pen number to the newly-defined color pair. Thus, the escape sequence for changing a color pair definition is:

```
ESC & v < 0 | 1 > m n a n b n c n x n y n z < pen # > I
```

where either a 0 or a 1 precedes the **m** parameter (selecting either RGB or HSL notation, respectively), and **n** is one of the legal values from the tables. <pen#> is an integer in the range 0 thru 7 which precedes the **i** parameter, and defines that pen number to be the color pair specified by the preceding a, b, c, x, y, and z parameters. Omitting any a, b, c, x, y, or z parameter causes a value of 0 to be assigned to the omitted parameter by default. Also, the background parameters can be specified before the foreground parameters.

Examples

```
^c&.v 0m 1a 0b 0c 0x 1y 0z 5I
```

This example re-defines pen 5 to specify red characters on a green background. (Note that 236 computers with color video ignore the green background specification and assign a black one instead.) This example is equivalent to

```
^c&.v 0m 1a 1y 5I
```

since omitted parameters (a, b, c, x, y, z) are given default values of 0.

```
^c&.v 1m .66a 1b 1c 3i 0m 1c 1x 1y 6I
```

This example re-defines pen 3 to specify blue characters on a black background (HSL notation), and pen 6 to specify blue characters on a yellow background (RGB notation). This example illustrates how multiple pens can be defined on a single line using different notations. (Again, note that the Model 236 with color video will reject the background specification of pen 6, and will use black instead.)

If you should specify color on color when setting up color definitions on the Model 236 computer with color video, you will find that the foreground color will remain as chosen and the background color will default to black.

```
^c&.v 5I
```

This example re-defines pen 5 to specify a black foreground and a black background, using the previous notation type.

Note

Supplying neither a foreground nor a background color when defining a color pair causes both the foreground and background to be black.

Configuring the ITE

The Internal Terminal Emulator is designed so that the various ITE characteristics can be configured quickly by displaying configure “menus” on the screen and then using system function keys to change the content of these menus.

Configuration Function Keys

To gain access to the configuration menus through the keyboard, press the function key **k9** (labeled AIDS on the system console overlay). This causes the following softkey display to appear at the bottom of the screen:



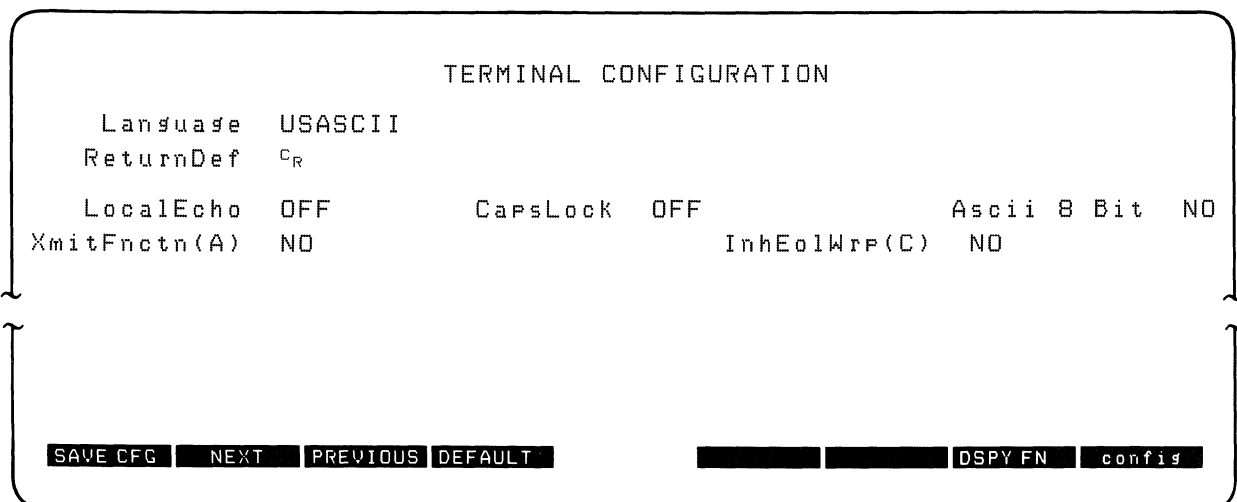
The function keys **f2** (tab/mrgn) and **f5** (FlexDisc) were covered in the section “Using the Internal Terminal Emulator” along with a brief discussion on the function key **k8** (config). When you press **f8** (config) a new softkey display appears at the bottom of your screen.



Pressing **f5** (terminal) fills the display with the **Terminal Configuration Menu** which is covered next.

Terminal Configuration Menu

After pressing **f5** (terminal) your display should look like this:



Description of Fields

The TERMINAL CONFIGURATION menu contains a set of unprotected fields that you access using the **TAB** key. Note that all fields can be changed using function keys **f2** (NEXT) and **f3** (PREVIOUS) with the exception of **ReturnDef** which is changed using the function key **f7** (DSPY FN).

There are seven fields which can be changed using the function keys as defined in the next section. These fields are described as follows:

- ReturnDef** specifies the definition of the **ENTER** key (labeled RETURN key on the overlay). The default definition is an ASCII c_R . The definition may consist of up to two characters. If the second character is a space, it is ignored and only the first character is used.
- Default: c_R space
- LocalEcho** specifies whether characters entered through the keyboard are both displayed on the screen and transmitted to the host computer.
- ON ($E_C \& K$ 1L)
- Characters entered through the keyboard are both displayed on the screen and transmitted to the host computer.
- CapSLock** determines whether the ITE generates the full 128-character ASCII set or only Teletype-compatible codes.
- ON ($E_C \& K$ 1C)
- The ITE generates only Teletype-compatible codes: uppercase ASCII (00-5F, hex) and DEL (7F, hex). Unshifted alphabetic keys (a-z) generate the codes for their uppercase equivalents. The {, | and } keys generate the codes for [, \, and] respectively. The keys for generating ~ and ` are disabled.
- OFF ($E_C \& K$ 0C)
- The ITE generates the full 128-character ASCII set of codes.
- Default: OFF
- XmitFncn(A)** determines whether the escape code functions are executed at the ITE and transmitted to the host computer.
- YES ($E_C \& S$ 1A)
- The escape code sequences generated by control keys such as **↑** and **↓** are transmitted to the host computer. If LocalEcho is ON, the function is also performed locally.
- NO ($E_C \& S$ 0A)
- The escape code sequences for the major function keys are executed locally but NOT transmitted to the host computer.
- Note that display functions will emit E_C Z and E_C Y to a host computer.
- Default: NO

InhEolWrP(C) designates whether or not the end-of-line wrap is inhibited.

NO (E_c&s 0C)

When the cursor reaches the right margin it automatically moves to the left margin in the next lower line (a local carriage return and line feed are generated).

YES (E_c&s 1C)

When the cursor reaches the right margin it remains in that screen column until an explicit carriage return or other cursor movement function is performed (succeeding characters overwrite the existing character in that screen column).

Language changes the character set on your keyboard to one of the following when you press function key or :

USASCII (United States)

SVENSK/SUOMI (Swedish/Finnish)

FRANCAIS azM (French AZERTY¹ layout with mutes)

FRANCAIS qwM (French QWERTY layout with mutes)

FRANCAIS az (French AZERTY¹ layout)

FRANCAIS qw (French QWERTY layout)

DEUTSCH (German)

ESPANOL M (Spanish with mutes)

ESPANOL (Spanish)

KATAKANA (Japanese)

The system console overlay works the same on all the keyboards listed. Diagrams of the previously mentioned keyboards can be found at the end of this article.

Series 200 computers support two character sets which contains the special characters associated with all of the international languages. These character sets are: Extended Roman and KATAKANA. The following charts show these character sets. Note that blank spaces in the chart are given the character **hp**; however, they have not been shown on the charts so that the left half of the chart or standard 7-bit ASCII code could be shown as separate from the right half of the chart or 8-bit code. The 8-bit code can be accessed by configuring the field **ASCII 8 Bit** to **YES**.

¹ The AZERTY characters can be obtained only through a software configuration of the keyboard. Physical AZERTY keyboards or hardware are not available on Series 200 computers.

To use the Extended Roman Character Set, configure your TERMINAL CONFIGURATION menu to the language you want, ASCII 8 Bit should be set to YES and your terminal (ITE) should be in the remote mode. All characters for the various languages mentioned in the menu are now available to you except KATAKANA.

COL BIT		8	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1		
ROW BIT		7	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
		6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1		
		5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
4	3	2	1																	
0	0	0	0	0	N	D		0	e	P	`	p				-	ヲ	ヱ		
0	0	0	1	1	S	D	!	1	A	Q	a	q			o	ア	チ	ㇿ		
0	0	1	0	2	S	D	"	2	B	R	b	r			r	イ	ツ	ㇾ		
0	0	1	1	3	E	D	#	3	C	S	c	s			l	ウ	テ	ㇽ		
0	1	0	0	4	E	D	\$	4	D	T	d	t			\	エ	ト	ㇼ		
0	1	0	1	5	E	k	%	5	E	U	e	u			.	オ	ト	ㇼ		
0	1	1	0	6	k	S	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
0	1	1	1	7	Q	E	'	7	G	W	g	w			ア	キ	ㇾ	ㇼ		
1	0	0	0	8	E	N	{	8	H	X	h	x			イ	ク	ㇾ	ㇼ		
1	0	0	1	9	H	E	}	9	I	Y	i	y			ウ	ケ	ㇼ	ㇼ		
1	0	1	0	10	L	S	*	:	J	Z	j	z			エ	コ	ㇼ	ㇼ		
1	0	1	1	11	Y	E	+	;	K	[k	{			オ	ㇾ	ㇼ	ㇼ		
1	1	0	0	12	F	S	,	<	L	≠	l				ㇼ	ㇼ	ㇼ	ㇼ		
1	1	0	1	13	S	S	-	=	M	J	m	}			ㇼ	ㇾ	ㇼ	ㇼ		
1	1	1	0	14	S	S	.	>	N	^	n	~			ㇼ	ㇼ	ㇼ	ㇼ		
1	1	1	1	15	S	S	/	?	O	_	o	*			ㇼ	ㇼ	ㇼ	ㇼ		ㇼ

KATAKANA Character Set

To use the KATAKANA Character Set, configure your TERMINAL CONFIGURATION menu to the language KATAKANA, ASCII 8 Bit should be set to YES and your terminal (ITE) should be in the remote mode. To type Japanese ASCII characters, press **CTRL** - **'**. If you want KATAKANA characters, press **CTRL** - **.**

For the French keyboard layouts, the AZERTY² and QWERTY designations refer to the location of the A, Z, Q, and W keys as follows:

AZERTY: Row 3 = A Z E R T Y
Row 2 = Q S D (etc.)
Row 1 = W X C (etc.)

QWERTY Row 3 = Q W E R T Y
Row 2 = A S D
Row 1 = Z X C (etc.)

For the French and Spanish keyboard layouts, the mutes designation refers to the manner in which certain accent character keystrokes are handled (^ and ` on the French layout and ` on the Spanish). If the mutes are enabled, those keystrokes will generate the particular accent character but will NOT move the cursor. If you then type an applicable vowel, the vowel will appear in the same character position as the accent and the cursor then moves to the next column (if you type any character other than an applicable vowel, however, the character will replace the accent character).

ASCII 8 Bit

transmits from full set of 8-bit codes when enabled (YES) and transmits only codes less than 128 when disabled (NO).

Values: YES (E_{C&K} 1I) = 8-bit codes.

NO (E_{C&K} 0I) = Standard 7-bit codes.

Changing the Fields

To change the fields in the display, press the **TAB** key until the cursor is located under the field you wish to change. Next, use the following function keys to change the state of the field.

- SAVE CFG** saves the fields on the configuration menu which you have altered.
- NEXT** changes the setting of the field you are presently in to the next setting in that field. For example, if you press **TAB** until you are located at the field **LocalEcho OFF** and then press **f2** the **LocalEcho** field changes to **ON**.
- PREVIOUS** changes the setting of the field you are presently in to the previous setting in that field. For example, if you press **TAB** until you are located at the field **LocalEcho ON** and then press **f3** the **LocalEcho** field changes to **OFF**.
- DEFAULT** causes the fields in the menu to be filled with their default value. The default values are as shown in the **TERMINAL CONFIGURATION** display at the beginning of this section. The only exception is the "language" field it defaults to the language option shipped with your system.
- DSPY FN** enables and disables the display functions mode. Pressing the key once enables the display functions mode and pressing it a second time disables it. When enabled * appears in the function key label box. You use the display function mode for entering ASCII control characters in the **ReturnDef** field. Note that this implementation of display function is separate from that which is enabled/disabled via the mode selection keys. Enabling or disabling display functions mode using this function key does **NOT** alter the effect of the **DISPLAY FCTNS** mode selection key (and vice versa).
- confis** removes the menu from the screen and changes the function key labels to the following:



ITE Escape Sequences

Several Internal Terminal Emulator (ITE) keyboard functions can be activated or controlled by a remote computer by use of escape-code sequences. The effect is identical to using non-ASCII keys on the ITE keyboard. Escape sequences consist of the escape-code character followed by one or more visible (non-control) ASCII characters.

The sequences listed in this section are recognized and executed by the ITE whether they are received from the data communication link or from the keyboard, although the keyboard is seldom used for escape sequences. If an illegal or unrecognized sequence is received, the ITE ignores the message and all subsequent data until one of the following characters is received: @, A thru Z, [, \, ^, _, carriage-return, escape-code or any ASCII 7-bit code less than the character space.

Sequence Types

There are two general categories of escape sequences. The two-character ITE control sequences are used primarily for ITE, screen, and cursor control. Most of these sequences are equivalent to keyboard operations that involve a single keystroke or the simultaneous pressing of two keys.

The extended escape sequences consist of the escape-code character followed by at least two subsequent characters. They are used, either for functions that are not included in the two-character sequences, or for sequences whose inherent complexity requires two or more characters in addition to the escape-code in order to define the operation. Absolute and relative cursor addressing are examples of operations that require longer control sequences.

Escape Sequences for ITE Control

Escape Code	Function
E _c 1	Set tab
E _c 2	Clear tab
E _c 3	Clear all tabs
E _c 4	NOT IMPLEMENTED (Set left margin)
E _c 5	NOT IMPLEMENTED (Set right margin)
E _c 9	NOT IMPLEMENTED (Clear all margins)
E _c @	NOT IMPLEMENTED (Makes the terminal program wait approximately one second.)
E _c A	Cursor up
E _c B	Cursor down
E _c C	Cursor right
E _c D	Cursor left
E _c E	Hard reset (power on reset of ITE)
E _c F	Cursor home down
E _c G	Move cursor to the left margin
E _c H	Cursor home up
E _c I	Horizontal tab
E _c J	Clear screen from cursor to the end of memory.

Escape Sequences for ITE Control (continued)

Escape Code	Function
E _c K	Clear line from cursor to end of line
E _c L	Insert line
E _c M	Delete line
E _c P	Delete character
E _c Q	Start insert character mode
E _c R	End insert character mode
E _c S	Roll up
E _c T	Roll down
E _c U	Next page
E _c V	Previous page
E _c W	NOT IMPLEMENTED (Format mode on)
E _c X	NOT IMPLEMENTED (Format mode off)
E _c Y	Enables display functions mode
E _c Z	Disables display functions mode
E _c [NOT IMPLEMENTED (Start unprotected field)
E _c]	NOT IMPLEMENTED (End unprotected/transmit-only field)
E _c ^	NOT IMPLEMENTED (Primary terminal status request)
E _c `	Sense cursor position(relative)
E _c a	Sense cursor position (absolute)
E _c b	NOT IMPLEMENTED (Unlock keyboard)
E _c c	NOT IMPLEMENTED (Lock keyboard)
E _c d	NOT IMPLEMENTED (Transmit a block of text to computer)
E _c f	NOT IMPLEMENTED (Modem disconnect)
E _c g	Soft reset (of ITE)
E _c h	Cursor home up.
E _c i	Backtab
E _c j	NOT IMPLEMENTED (Begin User Key Definition mode)
E _c k	NOT IMPLEMENTED (End User Keys Definition mode)
E _c l	NOT IMPLEMENTED (Begin Memory Lock mode)
E _c m	NOT IMPLEMENTED (End Memory Lock Mode)
E _c P	Default value for softkey <input type="button" value="f1"/>
E _c Q	Default value for softkey <input type="button" value="f2"/>
E _c R	Default value for softkey <input type="button" value="f3"/>
E _c S	Default value for softkey <input type="button" value="f4"/>
E _c t	Default value for softkey <input type="button" value="f5"/>
E _c U	Default value for softkey <input type="button" value="f6"/>
E _c V	Default value for softkey <input type="button" value="f7"/>
E _c W	Default value for softkey <input type="button" value="f8"/>
E _c Z	NOT IMPLEMENTED (Initiate terminal self test)
E _c ~	NOT IMPLEMENTED (Secondary terminal status request)

Extended Escape Sequences

Escape Code Sequence	Function
$\text{E}_{c\&.a} \langle \text{col} \rangle c \langle \text{row} \rangle Y$	Moves the cursor to column “col” and screen row “row” or the screen (screen relative addressing).
$\text{E}_{c\&.a} \langle \text{col} \rangle c \langle \text{row} \rangle R$	Moves the cursor to column “col” and row “row” in memory (absolute addressing).
$\text{E}_{c\&.a} \pm \langle \text{col} \rangle c \pm \langle \text{row} \rangle Y$	Moves the cursor to column “col” and row “row” (on the screen) relative to its present position (“col” and “row” are signed integers). A positive number indicates right or downward movement and a negative number indicates left or upward movement.
$\text{E}_{c\&.a} \pm \langle \text{col} \rangle c \pm \langle \text{row} \rangle R$	Moves the cursor to column “col” and row “row” (on the screen) relative to its present position (“col” and “row” are signed integers). A positive number indicates right or downward movement and a negative number indicates left or upward movement.
$\text{E}_{c\&.q} 0L$	NOT IMPLEMENTED (Unlock configuration)
$\text{E}_{c\&.q} 1L$	NOT IMPLEMENTED (Lock configuration)
$\text{E}_{c\&.d} \langle \text{char} \rangle$	Selects the display enhancement indicated by <char> to begin at the present cursor position. <char> can be E or A thru Q .
$\text{E}_{c\&.k} \langle x \rangle A$	AUTO LF enable: $x = 1$; disable: $x = 0$
$\text{E}_{c\&.k} \langle x \rangle B$	NOT IMPLEMENTED (BLOCK enable: $x = 1$; disable: $x = 0$)
$\text{E}_{c\&.k} \langle x \rangle C$	Caps Lock enable: $x = 1$; disable: $x = 0$
$\text{E}_{c\&.k} \langle x \rangle I$	ASCII8Bits enable: $x = 1$; disable: $x = 0$
$\text{E}_{c\&.k} \langle x \rangle J$	NOT IMPLEMENTED (FrameRate 50 Hz : $x = 1$; 60 Hz : $x = 0$)
$\text{E}_{c\&.k} \langle x \rangle L$	LocalEcho enable: $x = 1$; disable: $x = 0$
$\text{E}_{c\&.k} \langle x \rangle M$	NOT IMPLEMENTED (MODIFY ALL enable: $x = 1$; disable: $x = 0$)
$\text{E}_{c\&.k} \langle x \rangle N$	NOT IMPLEMENTED (SPOWLatch enable: $x = 1$; disable: $x = 0$)
$\text{E}_{c\&.k} \langle x \rangle P$	Caps Mode is primarily used as a typing convenience and affects only the 26 alphabetic keys. When it is enabled, all unshifted alphabetic keys generate uppercase letters and all shifted alphabetic keys generate lowercase letters. enable: $x = 1$; disable: $x = 0$
$\text{E}_{c\&.k} \langle x \rangle R$	REMOTE enable: $x = 1$; disable: $x = 0$

Extended Escape Sequences (continued)

Escape Code Sequence	Function
$\text{E}_c\&.s <x>A$	xmitFnctn(A) enable: x = 1; disable: x = 0
$\text{E}_c\&.s <x>B$	NOT IMPLEMENTED (SPOW(B) enable: x = 1; disable: x = 0)
$\text{E}_c\&.s <x>C$	InhEolWrp(C) enable: x = 1; disable: x = 0
$\text{E}_c\&.s <x>D$	NOT IMPLEMENTED (Line/Page(D) enable: x = 1; disable: x = 0)
$\text{E}_c\&.s <x>G$	NOT IMPLEMENTED (InfHndShk(G) enable: x = 1; disable: x = 0)
$\text{E}_c\&.s <x>H$	NOT IMPLEMENTED (Inh DC2(H) enable: x = 1; disable: x = 0)
$\text{E}_c\&.w 12F$	Turns on the display window (top 24 rows)
$\text{E}_c\&.w 13F$	Turns off the display window
$\text{E}_c * d <parameters>$	List of <parameters> for display control: E Turns on alphanumeric display; F Turns off alphanumeric display. When terminating a string of escape sequences with these parameters use a capital letter.
$\text{E}_c * s <parameter> ^$	Read device I.D. Status is parameter number 1.
$\text{E}_c\&.j <x>$	Enables and disables the function keys (f1 thru f8). If x equals: A Display the Modes set of function key labels, B Enable the User function keys. (The user key labels are displayed.) @ Remove the function key labels from the screen. The User function keys, however, are still active.
$\text{E}_c\&.f <attribute>a$ <key>k <label length>d <string length>L	Defines the function keys. Information on how this is done can be found in the function key group section of this manual under the definition of user keys.

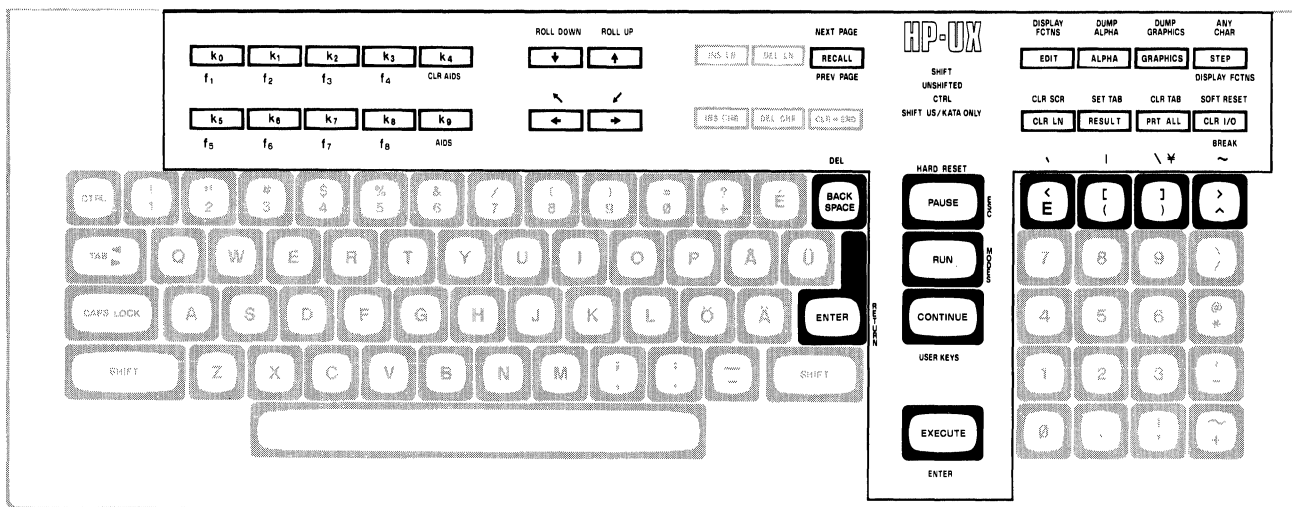
If an escape sequence is not recognized, the terminal ignores subsequent characters until ASCII decimal characters 0 thru 31 or 64 thru 95 is received, terminating the sequence. Note that E_c will terminate the old sequence and start a new one.

Keyboard Diagrams for Other Languages

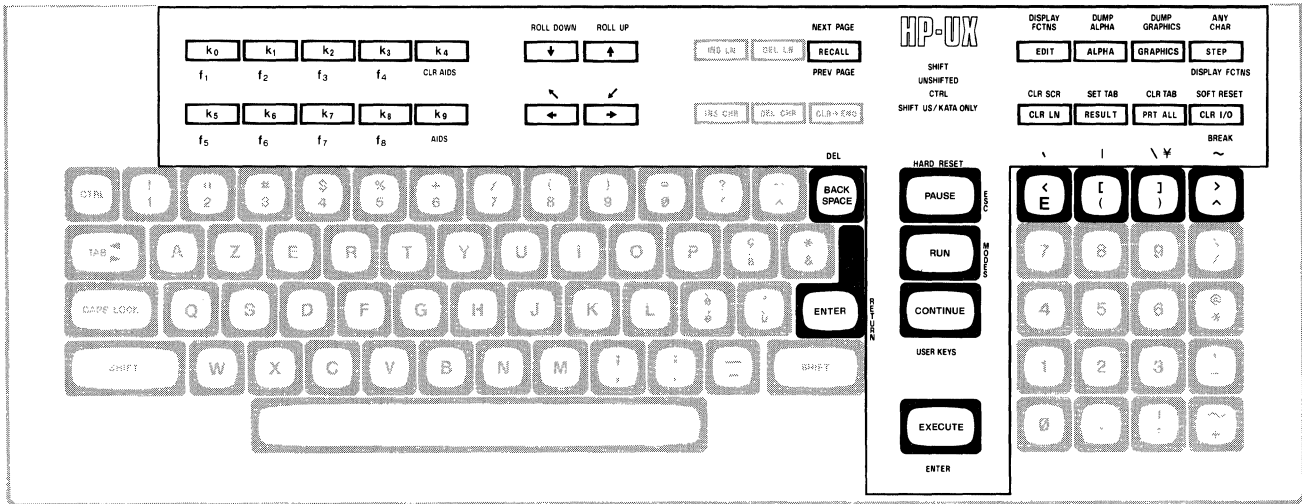
This section shows diagrams of the types of keyboards which are available to the Series 200 user. The keyboard option you now have can be configured to any one of the languages by use of the TERMINAL CONFIGURATION menu previously mentioned in this article.

To change to another keyboard language use the terminal configuration menu and select the **Language** field you want. Then change the **ASCII 8 Bit** field to YES. Next, save the changed configuration menu. Note that the languages accessed through the terminal configuration menu are not available in the vi editor. The vi editor only uses the first 128 ASCII characters of your systems particular character set. For more information on your keyboard, read the appropriate manual sent with your system.

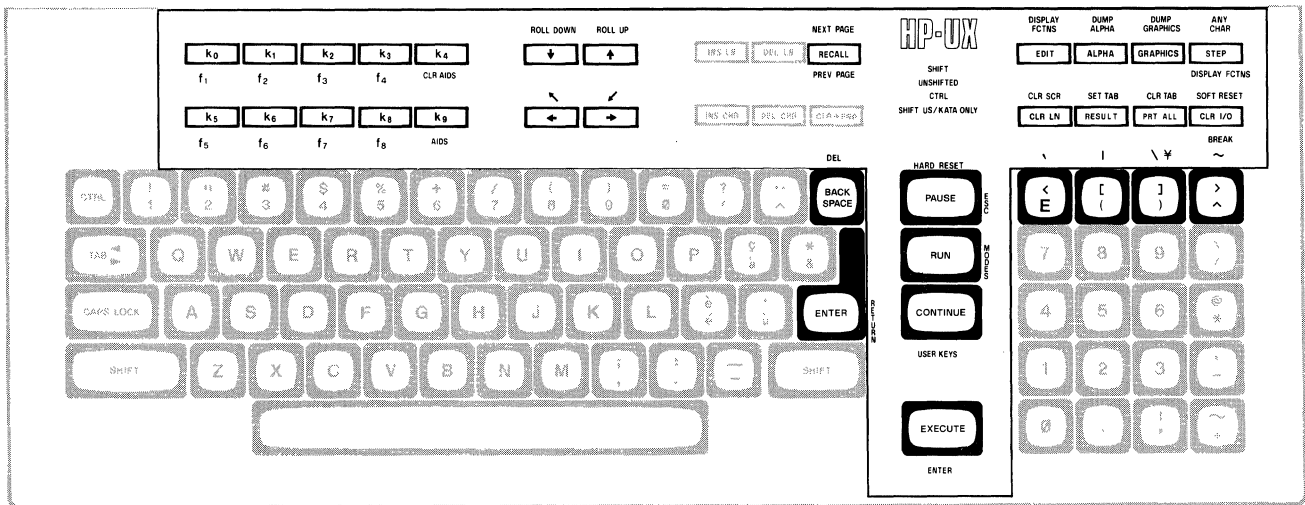
You will notice as you look at the keyboards that a majority of the character key and numeric key labels have changed. To use your keyboard effectively in any one of these languages, you will have to re-label the key caps.



SVENSK/SUOMI (Swedish/Finnish)

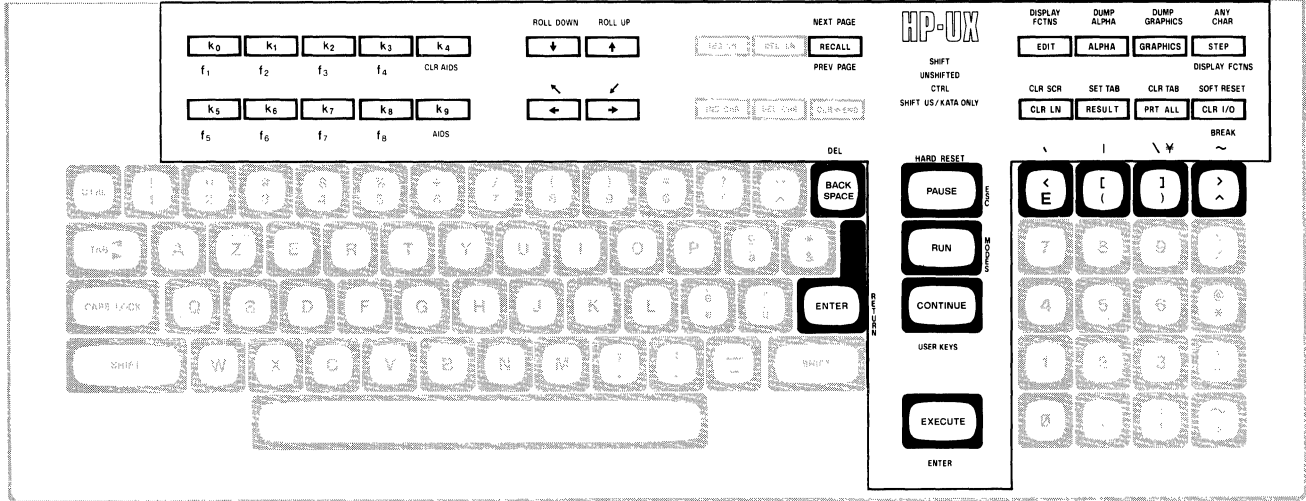


FRANCAIS azM (French AZERTY³ layout with mutes is not available)

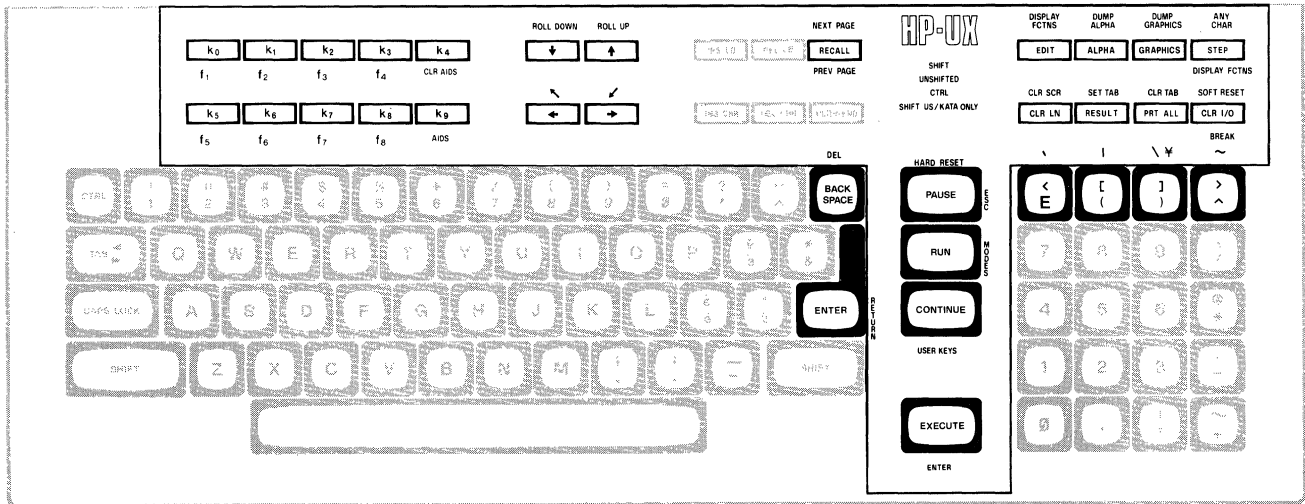


FRANCAIS qwM (French QWERTY layout with mutes)

³ The AZERTY characters can be obtained only through a software configuration of the keyboard. Physical AZERTY keyboards or hardware are not available on Series 200 computers.

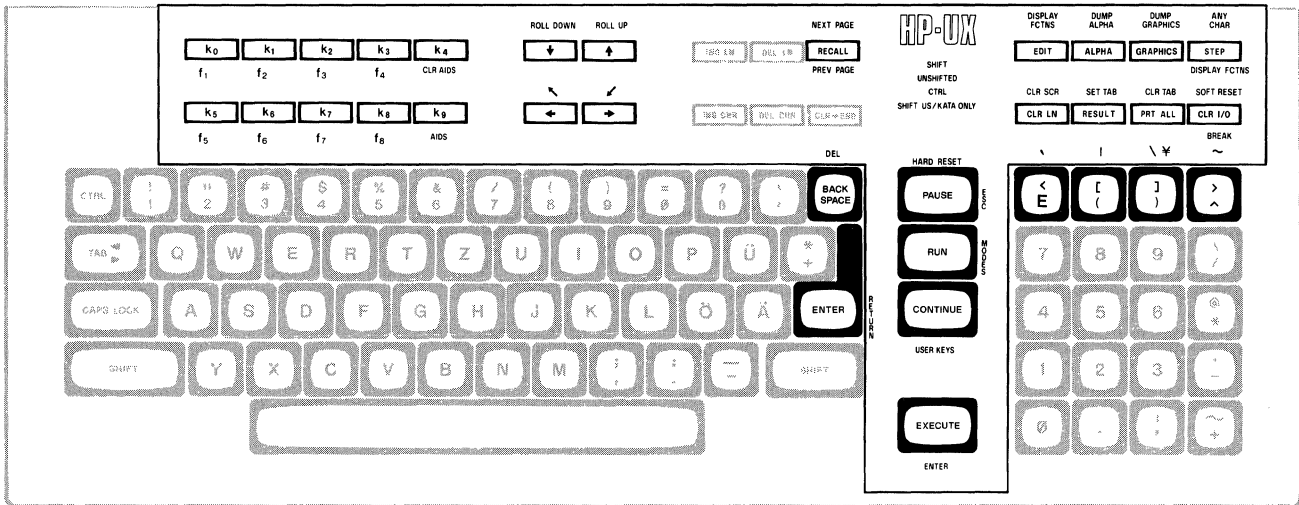


FRANCAIS az (French AZERTY⁴ layout is not available)

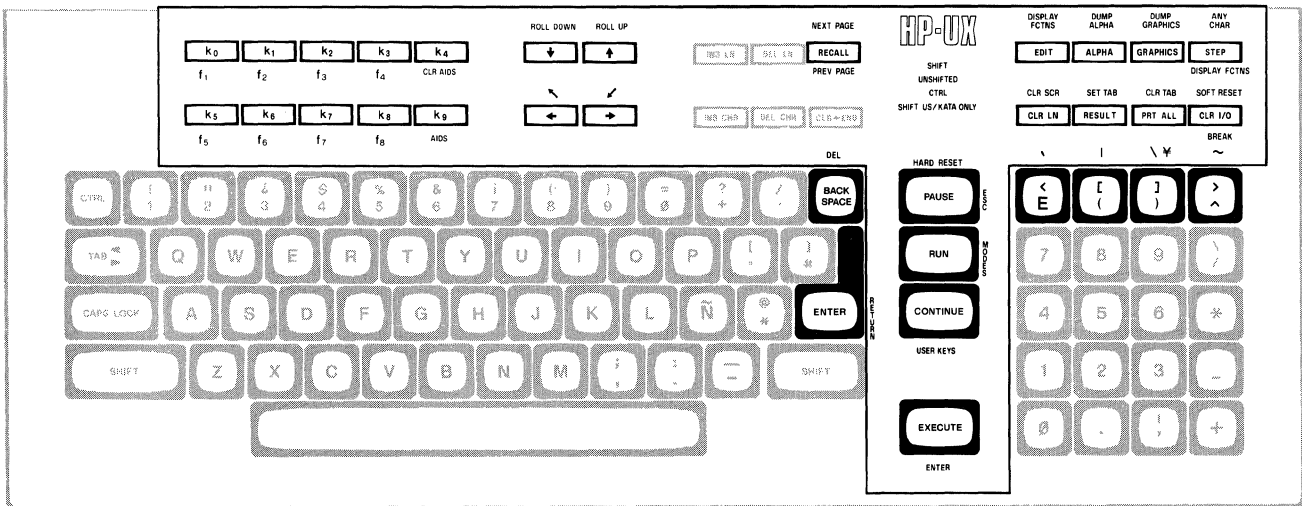


FRANCAIS qw (French QWERTY layout)

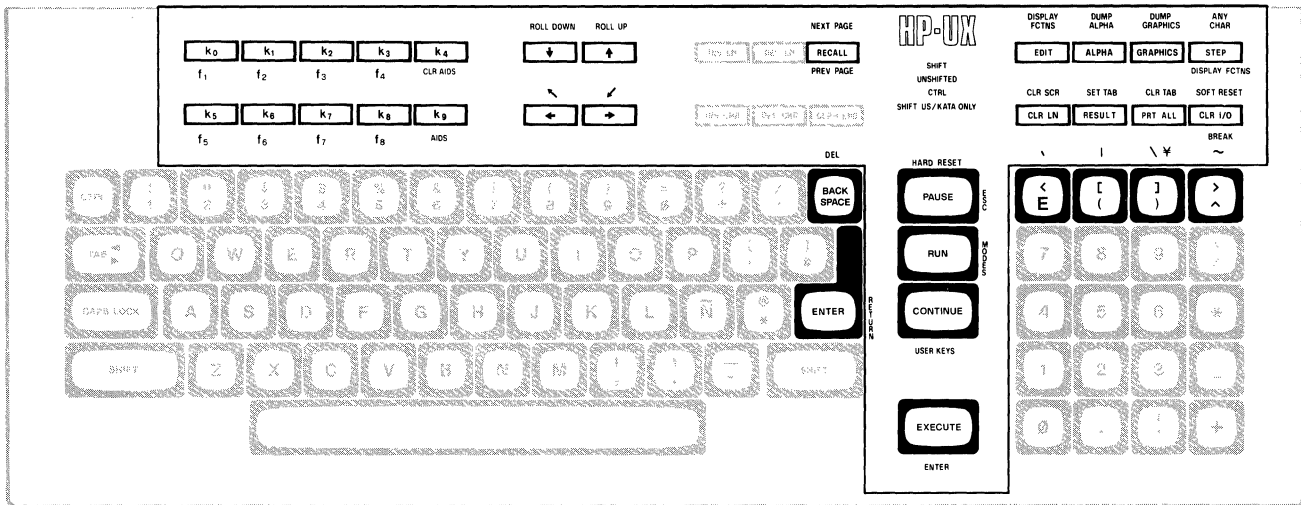
⁴ The AZERTY characters can be obtained only through a software configuration of the keyboard. Physical AZERTY keyboards or hardware are not available on Series 200 computers.



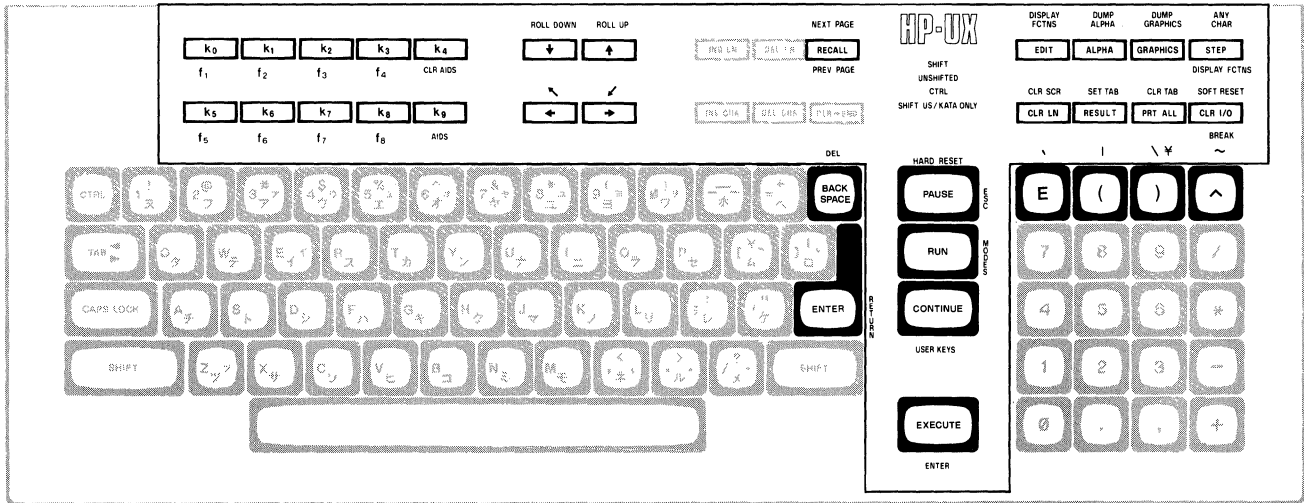
DEUTSCH (German)



ESPAÑOL M (Spanish with mutes)



ESPAÑOL (Spanish)



KATAKANA (Japanese)

Notes

Table of Contents

HP-UX and the HP 9000 Model 520 as System Console

The Keyboard.....	2
Alphanumeric Group.....	2
Numeric Pad Group	2
Display Control Group.....	2
Setting and Clearing Margins	2
Setting and Clearing Tab Stops.....	3
Cursor Control.....	3
Edit Group	6
Insert Character Mode.....	7
Function Key Group.....	7
Modes.....	7
AIDS.....	8
User Keys.....	9
User-definable Keys	9
The Display	11
Memory Addressing Scheme	11
Row Addressing	11
Column Addressing.....	11
Cursor Sensing.....	12
Absolute Sensing.....	12
Relative Sensing.....	12
Cursor Positioning.....	12
Screen Relative Addressing.....	13
Example	13
Absolute Addressing.....	13
Example	14
Cursor Relative Addressing.....	14
Example	15
Combining Absolute and Relative Addressing.....	15
Display Enhancements	15
Raster Control.....	16
Accessing Color	16
Selecting a Pen (Color Pair).....	17
Changing Pen Definitions	17
Examples.....	19
Controlling Configuration and Status.....	20
Re-configuring the Terminal.....	20
Sending Terminal Status.....	21
Primary Terminal Status	21
Secondary Terminal Status.....	23

HP-UX and the HP 9000 Model 520 As System Console

The **system console** is the terminal to which HP-UX sends system loader messages and soft system error messages. Like other terminals on an HP-UX system, it is also used for general system access (such as logging in, running programs, and entering data). The system console:

- must be connected to the computer via select code 0.
- must not be connected via a modem.
- must have a device file named */dev/console*.

Each system must have a system console. When HP-UX is run on the HP 9000 Model 520, the computer's keyboard and display act as the system console. This article describes the HP 9000 Model 520 as a terminal and as system console. It also discusses the methods of accessing the "terminal's" features: from the keyboard and from a program or command (via escape sequences).

HP-UX treats your HP 9000 Model 520 as six independent devices. The first device is a 32-bit mini-computer composed of a central processing unit, an I/O processor and memory. The second, third, fourth and fifth devices are: the built-in thermal printer, the built-in flexible disc drive, the built-in Winchester disc drive, and the graphics display. The sixth device is the computer's keyboard and display. This last device is the terminal and system console discussed in this article.

The display portion of the "terminal" consists of a display screen and display memory. The display cursor (a blinking underscore on the screen) indicates where the next character entered appears. As you enter characters, each is displayed at the cursor position, the ASCII code for the character is recorded at the associated position in display memory, and the cursor moves to the next character position on the screen. As the screen becomes full, newly entered data causes existing lines to be forced off the screen. Data lines forced off the screen are still maintained in display memory and can subsequently be moved back onto the screen. The size of display memory is determined by the HP-UX configuration. Once the display memory is full, additional data entered causes the older data in display memory to be lost.

Throughout this article, the sequence `\E` represents the escape character. Supplying an invalid escape sequence causes that sequence to be ignored. Escape sequences with optional or required parameters (referred to as "parameterized escape sequences") must be terminated by an upper case character before the sequence is implemented.

The Keyboard

The Model 520's keyboard is divided into major functional groups: the alphanumeric group, the numeric pad group, the display control group, the edit group, and the function group. Each function group is discussed in the sections below, with an emphasis on features and their access.

Alphanumeric Group

This group of keys is similar to a standard typewriter keyboard and consists of the alphabetic, numeric, and symbol keys. Included are lower and uppercase alphabetic characters, ASCII control codes, punctuation characters, and some commercial symbols.

Numeric Pad Group

The numeric group of keys is located to the right of the alphanumeric keys. The layout of the numeric key pad is similar to that of a standard office calculator. These keys are convenient for high-speed entry of large quantities of numeric data.

Display Control Group

The display control group consists of the keys that control the location of the cursor on the display. Each display control key and its function is described in the sections that follow. The escape code for accessing each display control feature is provided with each display control key. Some display control features can only be accessed via an escape sequence; no key is associated with the feature. The escape code for such features is also provided in the sections that follow.

Setting and Clearing Margins

You can redefine the left and/or right margin. These margins affect the cursor positioning for certain functions (such as carriage-return, home up, home down, etc.) and establish operational bounds for the insert character and delete character functions. In addition, the left margin is always an implicit tab stop. Data to the left of the left margin or to the right of the right margin is still accessible.

When you are entering data through the keyboard and the cursor reaches the right margin, it automatically moves to the left margin in the next lower line. When you press **RETURN** the cursor moves to the left margin in the current line if auto line feed mode is disabled or to the left margin in the next lower line if auto line feed mode is enabled.

Margins can be set with the AIDS keys (discussed in a later section) or with escape sequences:

- \E4 - set the left margin at the current cursor location.
- \E5 - set the right margin at the current cursor location.
- \E9 - clear both margins; by default the left margin becomes 1, the right margin becomes 80.

Attempting to set the left margin to the right of the right margin (or the right margin to the left of the left margin) causes the new margin to be rejected; the system beeps to notify you that the new margin was not accepted.

Setting and Clearing Tab Stops







You can define a series of tab stops to which you can move the cursor using the tab and back tab functions shown below. From the keyboard you set and clear tab stops using the **TAB SET** and **TAB CLEAR** keys. To set a tab stop, move the cursor to the desired location and press **TAB SET**. To clear a tab stop, move the cursor to the tab stop position and press **TAB CLEAR**. Additionally, you may use the functions provided with the AIDS keys (discussed in a later section) to set and clear tab stops.







Note that the left margin is always an implicit tab stop and cannot be cleared. The escape sequences to set and clear tab stops are:

- \E1 - set a tab stop at the current cursor position.
- \E2 - clear a tab stop previously set at the current cursor position.
- \E3 - clear all tab stops currently set. Note that this feature is available only from softkeys (as are the margin functions described above).

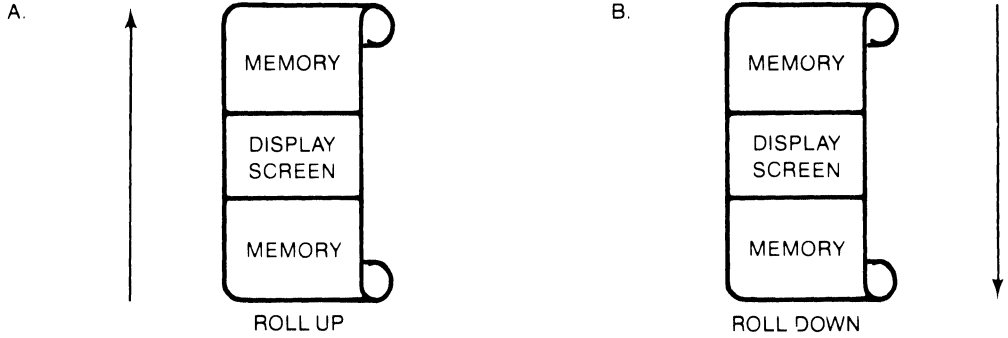
Cursor Control

Several keys exist on keyboard for changing the location of the cursor:

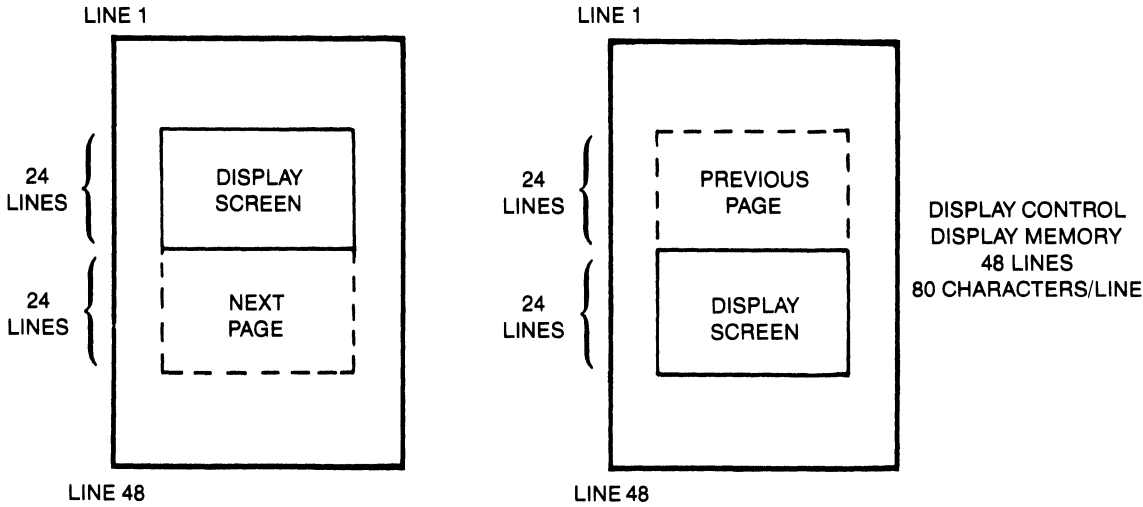
Key	Escape Sequence	Feature
	\EA	Move the cursor up one row in the current column position. Holding the key down causes the cursor to move continuously, row by row, until the key is released. When the cursor is in the top row of the screen, moving the cursor up actually moves the cursor to the same column position in the bottom row of the screen.
	\EB	Move the cursor down one row in the current column position. Holding the key down causes the cursor to move continuously, row by row, until the key is released. When the cursor is in the bottom row of the screen, moving the cursor down actually moves the cursor to the same column position in the top row of the screen.
	\EC	Move the cursor right one position in the current line; if the current position is the right margin, the cursor is moved to the left margin of the next line. Holding the key down causes the cursor to move continuously, column by column, until the key is released.
	\ED	Move the cursor left one position in the current line; if the current position is the left margin, the cursor is moved to the right margin of the previous line. Holding the key down causes the cursor to move continuously, column by column, until the key is released.
SHIFT 	\EH or \Eh	Home up: moves the cursor to the left margin in top row of the screen and rolls the text in display memory down as far as possible so that the first line in display memory appears in the top row of the screen.
SHIFT 	\EF	Home down: moves the cursor to the left margin in the bottom line of the screen and rolls the text in display memory up as far as necessary so that the last line in display memory appears immediately above the cursor position.

Key	Escape Sequence	Feature
none	\EG	Move the cursor to the left margin.
	\EI	Move the cursor forward to the next tab stop.
	\Ei	Move the cursor backwards to the previous tab stop.
	\ES	Roll the text in display memory up one row on the screen. The top row rolls off the screen, the remaining data rolls up one line on the screen, and a new line of data rolls from display memory into the bottom line of the screen. When the key is held down, the text continues to roll upward until the key is released or until the final line of data in display memory appears in the top row of the screen. In the latter case, pressing or continuing to press down the key has no further effect. The roll up and roll down functions are shown in the illustrations at the end of this section.
	\ET	Roll the text in display memory down one row on the screen. The bottom row rolls off the screen, the remaining data rolls down one line on the screen, and a new line of data rolls from the display memory into the top line of the screen. When the key is held down, the text continues to roll down until the key is released or until the first line of data in display memory appears in the top row of the screen. In the latter case, pressing or continuing to press down the key has no further effect. The roll up and roll down functions are shown in the illustrations at the end of this section.
	\EU	Roll the text in display memory up so that the next page (see the explanation below) of data replaces the current page on the screen. If the key is held down, the operation is repeated until the key is released or until the final line in display memory appears in the top line of the screen. In the latter case, pressing or continuing to hold down the key has no further effect.
	\EV	<p>The cursor is placed at the left margin, at the top row of the display.</p> <p>Roll the text in display memory down so that the previous page (see the explanation below) of data replaces the current page on the screen. If the key is held down, the operation is repeated until the key is released or until the first line in display memory appears in the top line of the screen. In the latter case, pressing or continuing to hold down the key has no further effect.</p> <p>The cursor is placed at the left margin, at the top row of the display.</p>

The data in display memory can be accessed (displayed on the screen) in blocks that are known as "pages". A page consists of 24 lines of data. The current page is that sequence of lines which appears on the screen at any given time. The **previous page** is the preceding 24 lines in display memory. The **next page** is the succeeding 24 lines in display memory. This concept, along with the concept of rolling data through the display screen and memory, are shown in the following illustrations.



The "Roll" Data Functions



Previous Page and Next Page Concepts

Edit Group

The edit group consists of the keys that allow you to modify the data presented on the screen. Currently, however, the edited data cannot be read back by the system. Typically, these features are used to modify data presented by programs. For example, the *vi* text editor program uses these features.

You can edit data on the screen by simply overstriking the old data. In addition, the following edit keys and escape sequences may be used:

Key	Escape Sequence	Function
CLEAR SCN	\EJ	Removes from display memory, all characters from the current location of the cursor to the end of display memory.
CLEAR LINE	\EK	Removes from display memory, all characters from the current location of the cursor to the end of the current line.
INS LN	\EL	The text line containing the cursor and all text lines below it roll downward one line, a blank line is inserted in the screen row containing the cursor, and the cursor moves to the left margin of the blank line. Holding the key down causes the operation to be repeated until the key is released.
DEL LN	\EM	The text line containing the cursor is deleted from display memory, all text lines below it roll upward one row, and the cursor moves to the left margin. Holding the key down causes the operation to be repeated until the key is released or until there are no subsequent lines of text remaining in display memory. In the latter case, pressing or continuing to hold down this key has no further effect.
DEL CHR	\EP	The cursor remains stationary while the character at the current cursor location is deleted. All characters between the cursor and the right margin move left one column and a blank moves into the line at the right margin. This function is meant to be used within that portion of the screen delineated by the left and right margins. If the cursor is positioned to the left of the left margin, the delete character function works as previously described. If the cursor is positioned beyond the right margin, the delete character function affects those characters from the current cursor position through the right boundary of the screen. If the key is held down, the terminal continues to delete characters until either the key is released or no characters remain between the cursor position and the right margin. In the latter case, pressing or continuing to hold down this key has no further effect.
INS CHR	\EQ	Turn on the insert character mode (see the description that follows).
INS CHR	\ER	Turn off the insert character mode (see the description that follows).

Insert Character Mode

When the “insert character” editing mode is enabled, characters entered through the keyboard or received from the computer are inserted into display memory at the cursor position. Each time a character is inserted, the cursor and all characters from the current cursor position through the right margin move one column to the right. Characters that are forced past the right margin are lost. When the cursor reaches the right margin, it moves to the left margin in the next lower line and the insert character function continues from that point.

The edit function is meant to be used within that portion of the screen delineated by the left and right margins. If the cursor is positioned to the left of the left margin, the insert character function works as previously described. If the cursor is positioned beyond the right margin, however, the insert character function affects those characters between the current cursor position and the right boundary of the screen. In such a case, when the cursor reaches the right boundary of the screen, it moves to the left margin in the next lower line and the insert character function continues from that point as described in the previous paragraph.

When the insert character mode is enabled (and softkey labels are displayed), the characters IC are displayed between the fourth and fifth function key labels. These characters are displayed to remind you that you are in the insert character mode.

Function Key Group

Across the top right of the keyboard are 16 keys labeled (0 16) through (15 31). HP-UX recognizes only the first 8 keys, (0 16) through (7 23), as function keys. The functions performed by these keys change dynamically as you use the terminal. At any given time the applicable function labels for these keys appear across the bottom of the display screen. However, softkeys are not supported by HP-UX (softkeys are those keys physically located on the display).

Modes

When you press the “MODES” key (12 28), the eight function keys are redefined. Pressing a redefined key allows access to one of the “modes” described in the following sections. The labels for the redefined keys are shown below (keys without labels are undefined):

(0 16)	(1 17)	(2 18)	(3 19)	(4 20)	(5 21)	(6 22)	(7 23)
			REMOTE			DISPLAY AUTO	
			MODE			FUNCT LF*	

You may use these function keys to enable and disable various terminal operating modes. Each defined mode selection key alternately enables and disables a particular mode. When the mode is enabled, an asterisk (*) appears in the associated key label on the screen (for example, auto line feed mode is enabled in the key menu above).

When the **remote mode** is enabled and a key is pressed, the terminal transmits the associated ASCII code to HP-UX. In **local mode** (remote mode is disabled), when an alphanumeric key is pressed the associated character is displayed at the current cursor position on the screen (nothing is transmitted to HP-UX).

When the **auto line feed mode** is enabled, an ASCII line feed control code is automatically appended to each ASCII carriage return control code generated through the keyboard. ASCII carriage return control codes can be generated through the keyboard in any of the following ways:

- By pressing either **EXECUTE** or **RETURN** (HP-UX treats these keys identically).
- By simultaneously pressing the keys **CTRL** and **M**.
- By pressing any of the user keys (**0 16** through **7 23**), provided that a carriage-return code is included in the particular key definition.

When the **display functions mode** is enabled, the terminal operates as follows:

- In local mode, it displays ASCII control codes and escape sequences but does not execute them. For example, if you press **←**, the terminal displays `\ED` on the screen but does not move the cursor one character to the left.
- In remote mode, it transmits ASCII control codes and escape sequences but does not execute them locally. For example, if you press **ROLL ↑**, the terminal transmits `\ES` but does not perform the “roll up” function. If local echo is enabled (ON) then the `\ES` is also displayed on the screen. **Local echo** specifies that the character is not only transmitted, but displayed on the terminal as well.

These same mode selection functions can be accessed via the escape sequences:

- `\E&k <x>R` – when x is 0, the remote mode is off; when x is 1, the remote mode is on.
- `\E&k <x>A` – when x is 0, auto line feed mode is off; when x is 1, auto line feed mode is on.
- `\EY` – enables display functions; when enabled, all printing and non-printing characters are displayed.
- `\EZ` – disables display functions; when disabled, only printing characters are displayed.

AIDS

When you press the AIDS key **12 28**, another menu is displayed, showing a single, defined key (the MARGINS/TABS key). When this key is pressed, the eight function keys become general control keys that you use for setting and clearing margins and tabs from the keyboard. Pressing one of the defined keys causes the terminal to issue the appropriate escape sequence for the function selected. These escape sequences and their function are discussed with the Display Control Group, earlier in this section.

Note that the MARGINS and TABS keys only send their associated escape sequences to HP-UX when display functions are enabled and when the A Strap is set (discussed later in this article). If these conditions are not met, the escape sequence is executed locally but is not sent to HP-UX.

User Keys

When you press the USER KEYS key (14 30), the eight function keys display the user defined key labels. In the following section ("User-definable Keys"), the function of the user keys and the procedure for defining them is described. To remove the user key labels from the screen (while still retaining their defined functions), press (12 28) (the AIDS key) while holding the (SHIFT) key depressed.

The USER KEYS key always toggles between displaying the current key labels and the user key labels.

User-definable Keys

The eight function keys (0 16) through (7 23), besides performing the terminal control functions described above, can be defined by a program. In this context, "defined" means:

- You can assign to each key a string of ASCII alphanumeric characters and/or control codes (such as carriage return or line feed).
- You can specify each key's operation attribute: whether its key definition is to be executed locally at the terminal, transmitted to the computer, or both.
- You can assign to each key an alphanumeric label (up to 16 characters) which, in user keys mode (i.e. when the USER KEYS key (14 30) is pressed), is displayed across the bottom of the screen.

The definition of each user key may contain up to 80 characters (alphanumeric characters, ASCII control characters, and explicit escape sequence characters).

To define a user-definable key, enter the escape sequence:

```
\E&f <attribute><key><label length><string length><label><string>
```

where <attribute> is a two character combination from the list 0a, 1a, or 2a. The default value for <attribute> is 0a. The attribute character specifies whether the definition of the particular user key is to be:

- a. Treated in the same manner as the alphanumeric keys (0a).

If the terminal is in local mode, the definition of the key is executed locally. If the terminal is in remote mode and local echo is disabled (OFF), the definition of the key is transmitted to the computer. If the terminal is in remote mode and local echo is enabled (ON), the definition of the key is both transmitted to the computer and executed locally.

- b. Executed locally only (1a).
- c. Transmitted to the computer only (2a).

When the transmit-only attribute (2a) is designated, the particular user key has no effect unless the terminal is in remote mode. A transmit-only user key appends the appropriate terminator to the string (either carriage-return or carriage-return/line feed, depending on the state of Auto Line Feed).

<key> is a two character identifier specifying the key to be defined. The key is specified by a value in the range 1k through 8k (1k is the default). For example, to specify the fifth user key, enter 5k for <key>. Note that this differs from the physical key labels on the HP 9000 Model 520's keyboard (they are labeled 0 through 7).

<label length> is the number of characters in the key label. Acceptable values are in the range 0d through 16d. Specifying a zero length causes the key label to remain unchanged. 0d is the default value for the label length.

<string length> is the length of the string forming the key definition. Acceptable values are in the range -1L through 80L; 1L is the default. Entering a string length value of zero causes the key definition to remain unchanged. A string length value of -1 causes the key definition to be erased.

<label> is the character sequence for the label.

<string> is the character sequence for the key definition.

The <attribute>, <key>, <label length>, and <string length> parameters may appear in any sequence but must precede the label and key definition strings. You must use an uppercase identifier (A, K, D, or L) for the final parameter and a lowercase identifier (a, k, d, or l) for all preceding parameters. If any of the four fields are omitted, their default values are used. At least one of the parameters must be specified because its uppercase identifier is needed to terminate the sequence.

Following the parameters, the first 0 through 16 characters, as designated by <label length>, constitute the key's label and the next 0 through 80 characters, as designated by <string length>, constitute the key's definition string. The total number of characters (alphanumeric data, ASCII control codes such as carriage-return and line feed, and explicit escape sequence characters) in the label string can exceed 16, but only the first 16 characters are used. The same is true for the destination string; only the first 80 characters are used.

The initial (power-on) definition of the user keys is:

- all keys are transmit-only (attribute is Z a).
- the user key labels are f 1 through f 8.
- definitions are \Ep, \Eq, \Er, \Es, \Et, \Eu, \Ev, and \Ew for keys

0	16
---	----

 through

7	23
---	----

, respectively.

The Display

The “terminal’s” display has many features of its own, such as video highlights (inverse video and blinking), raster control, cursor sensing and addressing, and color highlight control (for Model 520 Computers equipped with a color display). These functions are accessed only through escape sequences and are discussed in the sections that follow.

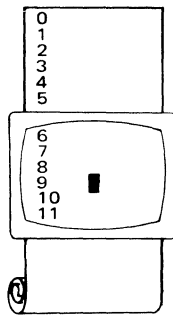
Memory Addressing Scheme

Display memory positions can be addressed using absolute or relative coordinate values. Display memory is made up of 80 columns (0 - 79) and any number of 24 line pages (specified by the HP-UX configuration). As shipped to you, the display memory has 48 lines (0 - 47) of 80 characters (2 screens). The amount of display memory can be determined from byte 0 of the primary terminal status (discussed in the section entitled “The Terminal”, later in this article). The types of addressing available are absolute (memory relative), screen relative, and cursor relative.

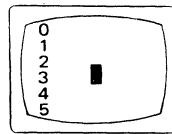
Row Addressing

The figure below illustrates the way that the three types of addressing affect row or line numbers. The cursor is shown positioned in the fourth row on the screen. Screen row 0 is currently at row 6 of display memory. In order to reposition the cursor to the first line of the screen the following three destination rows could be used:

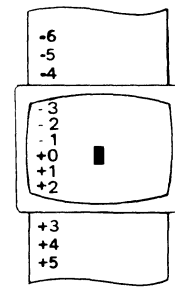
- Absolute: row 6
- Screen Relative: row 0
- Cursor Relative: row -3



a.) Absolute: row 6



b.) Screen Relative: row 0



c.) Cursor Relative: row -3

Row Addressing

Column Addressing

Column addressing is accomplished in a manner similar to row addressing. There is no difference between screen and cursor relative column addressing. The figure below illustrates the difference between absolute and relative addressing. The cursor is shown in column 5.

Screen Relative Addressing

To move the cursor to any character position on the screen, use any of the following escape sequences:

```
\E&a<column number> c <row number>Y  
\E&a<row number> y <column number>C  
\E&a<column number>C  
\E&a<row number>Y
```

where <column number> is a decimal number specifying the screen column to which you wish to move the cursor. Zero specifies the leftmost column.

<row number> is a decimal number specifying the screen row (0 - 23) to which you wish to move the cursor. Zero specifies the top row of the screen; 23 specifies the bottom row.

When using the escape sequences for screen relative addressing, the data on the screen is not affected (the cursor may only be moved around in the 24 rows and 80 columns currently displayed, thus data is not scrolled up or down).

If you specify only <column number>, the cursor remains in the current row. Similarly, if you specify only <row number>, the cursor remains in the current column.

Example

The following escape sequence moves the cursor to the 20th column of the 7th row on the screen:

```
\E&a6719C
```

Absolute Addressing

You can specify the location of any character within display memory by supplying absolute row and column coordinates. To move the cursor to another character position using absolute addressing, use any of the following escape sequences:

```
\E&a<column number> c <row number>R  
\E&a<row number> r <column number>C  
\E&a<column number>C  
\E&a<row number>R
```

where <column number> is a decimal number (0 - 79) specifying the column coordinate (within display memory) of the character at which you want the cursor positioned. Zero specifies the first (leftmost) column in display memory, 79 the rightmost column.

<row number> is a decimal number (0-max) specifying the row coordinate (within display memory) of the character at which you want the cursor positioned. Zero specifies the first (top) row in display memory, max specifies the last. The value of max is specified as:

$$[24 (\text{lines/page}) \times \text{num_page} (\text{pages})] - 1$$

where num_page is the number of pages of display memory specified by the system configuration. As shipped to you, the configuration dictates that 2 pages of display memory be allocated. Thus, the last row that can be addressed is 47.

When using the above escape sequences, the data visible on the screen rolls up or down (if necessary) in order to position the cursor at the specified data character. The cursor and data movements occur as follows:

- If a specified character position lies within the boundaries of the screen, the cursor moves to that position; the data on the screen does not move.
- If the absolute row coordinate is less than that of the top line currently visible on the screen, the cursor moves to the specified column in the top row of the screen; the data then rolls down until the specified row appears in the top line of the screen.
- If the absolute row coordinate exceeds that of the bottom line currently visible on the screen, the cursor moves to the specified column in the bottom row of the screen; the data then rolls up until the specified row appears in the bottom line of the screen.

If you specify only a <column number>, the cursor remains in the current row. Similarly, if you specify only a <row number>, the cursor remains in the current column.

Example

To position the cursor (rolling the data if necessary) at the character residing in the 60th column of the 27th row in display memory, the escape sequence is:

```
\E&a26r59C
```

Cursor Relative Addressing

You can specify the location of any character within display memory by supplying row and column coordinates that are relative to the current cursor position. To move the cursor to another character position using cursor relative addressing, use any of the following escape sequences:

```
\E&a +/- <column number> c +/- <row number>R  
\E&a +/- <row number> r +/- <column number>C  
\E&a +/- <column number>C  
\E&a +/- <row number>R
```

where <column number> is a decimal number specifying the relative column to which you wish to move the cursor. A positive number specifies how many columns to the right you wish to move the cursor; a negative number specifies how many columns to the left.

<row number> is a decimal number specifying the relative row to which you wish to move the cursor. A positive number specifies how many rows to the right you wish to move the cursor; a negative number specifies how many rows to the left.

When using the above escape sequences, the data visible on the screen rolls up or down (if necessary) in order to position the cursor at the specified data character. The cursor and data movements occur as follows:

- If a specified character position lies within the boundaries of the screen, the cursor moves to that position; the data on the screen does not move.
- If the specified cursor relative row precedes the top line currently visible on the screen, the cursor moves to the specified column in the top row of the screen; the data then rolls down until the specified row appears in the top line of the screen.
- If the specified cursor relative row precedes the bottom line currently visible on the screen, the cursor moves to the specified column in the bottom row of the screen; the data then rolls up until the specified row appears in the bottom line of the screen.

If you specify only a <column number>, the cursor remains in the current row. Similarly, if you specify only a <row number>, the cursor remains in the current column.

Example

To position the cursor (rolling the data if necessary) at the character residing 15 columns to the right and 25 rows above the current cursor position (within display memory), use the escape sequence:

```
\E&a+15c-25R
```

Combining Absolute and Relative Addressing

You may use a combination of screen relative, absolute and cursor relative addressing within a single escape sequence.

For example, to move the cursor (and roll the text if necessary) so that it is positioned at the character residing in the 70th column of the 18th row below the current cursor position, use the escape sequence:

```
\E&a69c+18R
```

Similarly, to move the cursor (and roll the text up or down if necessary) so that it is positioned at the character residing in the 10th column of absolute row 48 in display memory, use the escape sequence:

```
\E&a9c47R
```

Display Enhancements

The terminal includes as a standard feature the following display enhancement capabilities:

- Inverse Video - black characters are displayed against a white background.
- Underline Video - characters are underscored.
- Blink Video - characters blink on and off.

Note

The half bright display enhancement is not implemented on this terminal. When the half bright enhancement is selected on the HP 9000 Model 520 with a black-and-white display, it is ignored. Selecting the half bright enhancement on the HP 9000 Model 520 with a color display causes the terminal to select pen 3. (See the section "Accessing Color" later in this article).

The display enhancements are used on a field basis. The field cannot span more than one line. The field scrolls with display memory. Overwriting a displayable character in a field preserves the display enhancement. The enhancements may be used separately or in any combination. When used, they cause control bits to be set within display memory.

From a program or from the keyboard, you enable and disable the various video enhancements by embedding escape sequences within the data. The general form of the escape sequence is:

```
\E&d<enhancement code>
```

where enhancement code is one of the uppercase letters A through O specifying the desired enhancement(s) or an @ to specify end of enhancement:

Enhancement Character

	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Half-Bright									x	x	x	x	x	x	x	x
Underline					x	x	x	x					x	x	x	x
Inverse Video			x	x			x	x			x	x			x	x
Blinking		x		x		x		x		x		x		x		x
End Enhancement	x															

Note that the escape sequence for “end enhancement” (`\E&d@`) or the escape sequence for another video enhancement, ends the previous enhancement.

Raster Control

The terminal provides the ability to enable and disable both its alphanumeric display and its graphic display. The escape sequences for these capabilities are:

- `\E*dc` - Turn on graphics display; enable writing to the graphics display.
- `\E*dd` - Turn off graphics display; disable writing to the graphics display.
- `\E*de` - Turn on the alphanumeric display; enable writing to the alphanumeric display.
- `\E*df` - Turn off the alphanumeric display; disable writing to the alphanumeric display.

Whether used individually or in combination, the last character of the escape sequence must be uppercase. For example, to turn off the graphics display, use the escape sequence `\E*dD`. When these sequences are combined, an uppercase specifier must terminate the sequence. To turn on the graphics display and turns off the alphanumeric display, use the escape sequence `\E*dcF`.

Accessing Color

If your Model 520 computer is equipped with a color display, you may access its color capabilities from HP-UX. First, you need to understand some simple terms.

Color pair - two colors which define the foreground color (color of the characters) and the background color, respectively. At least one color of the color pair must be black; displaying color on color is not possible. A total of 15 color pairs are possible, but only eight can be displayed at any one time.

Pen # - one of eight predefined color pairs. Pen 0 through pen 7 are initially defined as follows (re-defining a color pair is described later):

Pen #	foreground color	background color
0	white	black
1	red	black
2	green	black
3	yellow	black
4	blue	black
5	magenta	black
6	cyan	black
7	black	yellow

Pen #0 is the default pen selected by the terminal when writing to the display.

Pen #7 is always used for displaying the softkey labels.

Selecting a Pen (Color Pair)

By using an escape sequence, you can select a pen number other than pen #0 when writing to the display. Like other display enhancements, pen selection is used on a field basis. The field cannot span more than one line. That is, the pen selection is only active until a new-line character is encountered; then the default pen is re-selected. The escape sequence for selecting a pen is:

```
\E&v n<parameter>
```

where n is the pen number you wish to use, and <parameter> is a single character that specifies what action you want to take. To select a pre-defined pen number, the necessary <parameter> is **s**. If n > 7, HP-UX performs the calculation (n Modulo 8) on the supplied value to determine the actual pen number. Thus,

```
\E&v 4S
```

selects the pre-defined pen number 4. Note that **s** is capitalized in the preceding escape sequence. This is because escape sequences are terminated by a capital letter. Thus, the last character of any escape sequence *must* be uppercase. However, if a parameter is *not* the last character of the escape sequence, it may appear in lower-case.

Changing Pen Definitions

You may change the pre-defined color pair for any of the eight existing display pens. The three primary colors (red, green and blue) are used in various combinations to achieve the desired color.

The combinations of red, green, and blue that define foreground and background colors can be specified in two notations. The first is RGB (Red-Green-Blue), and the second is HSL (Hue-Saturation-Luminosity). The notation must be selected before you can redefine pens (if no notation type is specified, the “terminal” uses the last notation specified, or RGB notation at power-up). To select a notation type, use the \E&v escape sequence used above:

```
\E&v n<parameter>
```

where n is 0 (for RGB) or 1 (for HSL), and <parameter> is the letter **m**. Thus, the sequence

```
\E&v 1M
```

selects HSL notation. It does nothing more.

To specify the quantity of red (hue), green (saturation), and blue (luminosity) to appear in your background and foreground colors, the a, b, c, x, y, and z parameters are used. These parameters have the following meanings:

- a specifies the amount of red (hue) used in the foreground.
- b specifies the amount of green (saturation) used in the foreground.
- c specifies the amount of blue (luminosity) used in the foreground.
- x specifies the amount of red (hue) used in the background.
- y specifies the amount of green (saturation) used in the background.
- z specifies the amount of blue (luminosity) used in the background.

Each a, b, c, x, y, and z parameter specified is preceded by a number in the range 0 through 1, in increments of 0.01. The following table gives the values needed to define the eight principle colors:

R	G	B	Color	H	S	L
0	0	0	Black	X	X	0
0	0	1	Blue	0.66	1	1
0	1	0	Green	0.33	1	1
0	1	1	Cyan	0.50	1	1
1	0	0	Red	1	1	1
1	0	1	Magenta	0.83	1	1
1	1	0	Yellow	0.16	1	1
1	1	1	White	X	0	1

(Note that X's in the above table represent "don't care" situations.)

One final parameter, **i**, is needed. It is used to assign a pen number to the newly-defined color pair. Thus, the escape sequence for changing a color pair definition is:

`\E&v <0|1>m na nb nc nx ny nz <pen#>I`

where either a 0 or a 1 precedes the **m** parameter (selecting either RGB or HSL notation, respectively), and n is one of the legal values from the table above. `<pen#>` is an integer in the range 0 - 7 which, when combined with the **i** parameter, defines that pen number to be the color pair specified by the preceding a, b, c, x, y, and z parameters. Omitting any a, b, c, x, y, or z parameter causes a value of 0 to be assigned to the omitted parameter by default.

Examples

```
\E&v 0m 1a 0b 0c 0x 1y 0z 5I
```

This example re-defines pen 5 to specify red characters on a green background. (Note that the Model 520 will ignore the green background specification and assign a black one instead.) This example is equivalent to

```
\E&v 0m 1a 1y 5I
```

since omitted parameters (a, b, c, x, y, z) are given default values of 0.

```
\E&v 1m .66a 1b 1c 3i 0m 1c 1x 1y 6I
```

This example re-defines pen 3 to specify blue characters on a black background (HSL notation), and pen 6 to specify blue characters on a yellow background (RGB notation). This example illustrates how multiple pens can be defined on a single line using different notations. (Again, note that the Model 520 will reject the background specification of pen 6, and will use black instead.)

```
\E&v 0m 1y 1z 1a 1c 4I
```

This example re-defines pen 4 to specify a cyan background with magenta characters. This example shows how background and foreground specifications can be reversed. The Model 520 will accept the magenta foreground, but will reject the cyan background; black will be used instead.

If the foreground and background colors are both non-black, the foreground color will be used, and the background color will be black, regardless of the order in which the parameters are specified.

```
\E&v 5I
```

This example re-defines pen 5 to specify a black foreground and a black background, using the previous notation type.

Note

Supplying neither a foreground nor a background color when defining a color pair causes both the foreground and background to be black. This is like typing on a typewriter without paper or ribbon; you can't see what is written.

Controlling Configuration and Status

The terminal provides additional escape sequences for managing its configuration and its status.

Re-configuring the Terminal

The terminal allows you to reset a few of its configuration parameters via escape sequences. These parameters and their escape sequences are:

Function	Escape Sequence	Description
Auto Line Feed Mode	\E&k nA	When n is 0, auto line feed mode is off. When n is 1, auto line feed mode is on. Default = OFF.
Local Echo	\E&k nL	Characters entered through the keyboard are displayed on the screen and transmitted to the computer when n = 1. When n = 0, characters entered through the keyboard are transmitted to the computer only; if they are to appear on the screen, the computer must "echo" them back to the terminal. Default = OFF.
Remote Mode	\E&k nR	When n is 0, the remote mode is off. When n is 1, the remote mode is on. Default = ON.
Caps Mode	\E&k nP	When caps mode is enabled, all unshifted alphabetic keys generate uppercase letters and all shifted alphabetic keys generate lowercase letters. This mode is used primarily as a typing convenience and affects only the 26 alphabetic keys. When n = 1, the caps mode is enabled. When n = 0, the caps mode is disabled. Default = OFF. From the keyboard, you enable and disable caps mode using the CAPS key. This key alternately enables and disables caps mode.
Transmit Function (STRAP A)	\E&s <x>A	This escape sequence specifies whether or not escape code sequences are both executed at the terminal and transmitted to HP-UX. When x = 1, the escape code sequences generated by control keys such as ROLL ↑ and ROLL ↓ are transmitted to HP-UX. If local echo is ON, the function is also performed locally. When x = 0, the escape sequences for the major function keys are executed locally, but are not transmitted to HP-UX. The default is x = 0.
Enable End Of Line Wrap (STRAP C)	\E&s <x>C	This field specifies whether or not the end-of-line wrap is inhibited. When x = 0 and the cursor reaches the right margin, it automatically moves to the left margin in the next lower line (a local carriage return and line feed are generated). When x = 1 and the cursor reaches the right margin, it remains in that screen column until an explicit carriage return or other cursor movement function is performed (succeeding characters overwrite the existing character in that screen column). Default = OFF.

Sending Terminal Status

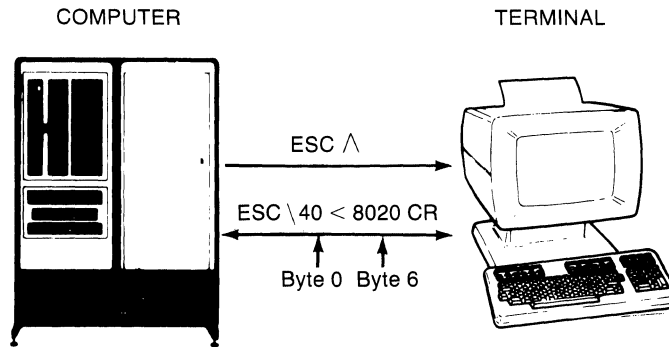
Terminal status is made up of 14 status bytes (bytes 0 through 13) containing information such as display memory size, switch settings, configuration menu settings, and terminal errors. There are two terminal status requests: primary and secondary. Each returns a set of seven status bytes.

Primary Terminal Status

You can request the first set of terminal status bytes (bytes 0 through 6) by issuing the following escape sequence:

`\E^`

The terminal responds with an `\E\`, and seven status bytes followed by a terminator (a carriage-return character). A typical primary terminal status request and response is shown in the following illustration.

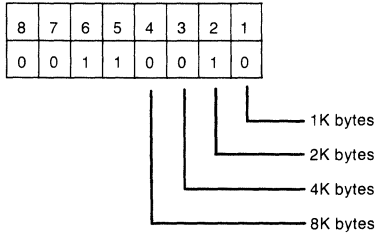


BYTE	ASCII	BINARY	STATUS
0	4	0011 0100	4K bytes of display memory (2 pages of 24 lines)
1	0	0011 0000	Function key transmission disabled
			Cursor wraparound disabled
2	<	0011 1100	Configuration Straps A-H
3	8	0011 1000	
			Auto line feed disabled
			Terminal sends secondary status
4	0	0011 0000	
5	2	0011 0010	Last Self-Test ok
6	0	0011 0000	

Primary Terminal Status Example

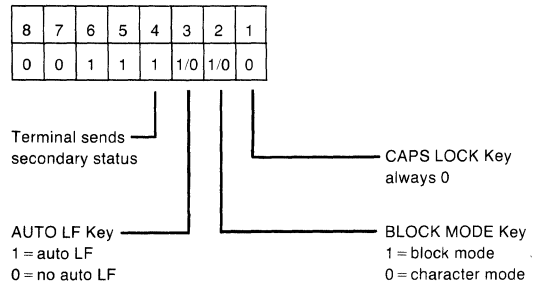
PRIMARY STATUS BYTES

BYTE 0 DISPLAY MEMORY SIZE

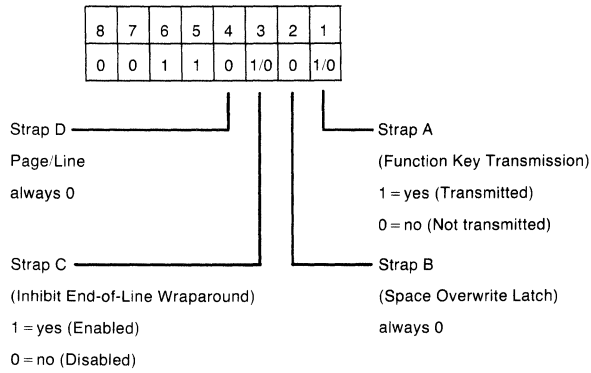


This byte specifies the amount of display memory available in the terminal.
(roughly 2K per page)

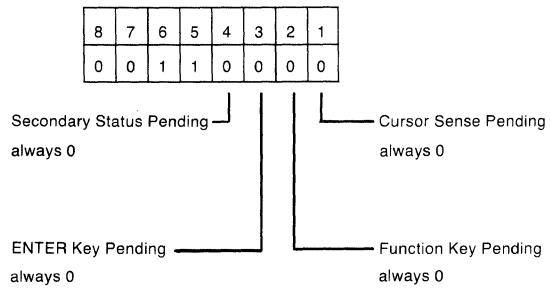
BYTE 3 LATCHING KEYS



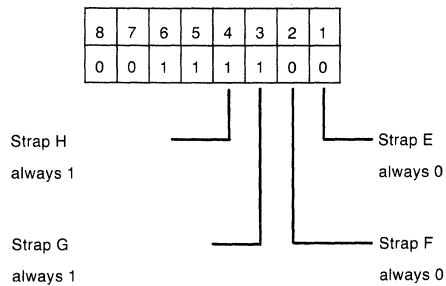
BYTE 1 CONFIGURATION STRAPS A-D



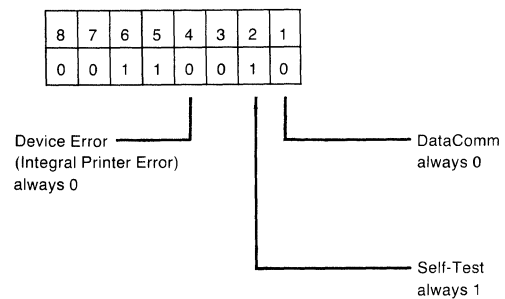
BYTE 4 TRANSFER PENDING FLAGS



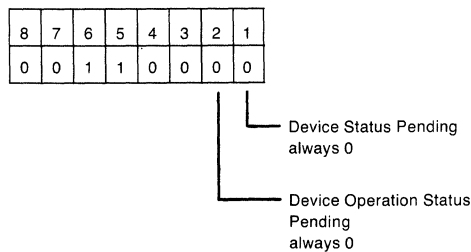
BYTE 2 CONFIGURATION STRAPS E-H



BYTE 5 ERROR FLAGS



BYTE 6 DEVICE TRANSFER PENDING FLAGS



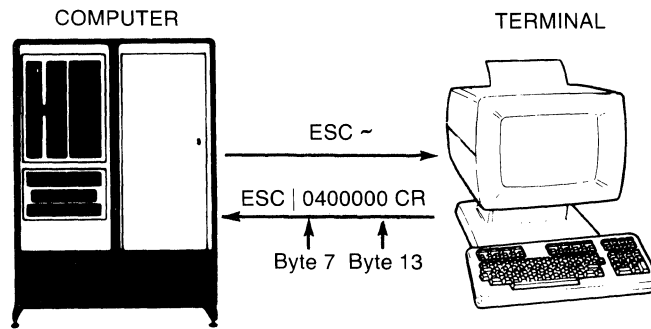
(tracks "S", "F", or "U" completion codes associated with E_c &p device control sequences.)

Secondary Terminal Status

You can request the second set of terminal status bytes (bytes 7 through 13) by issuing the following escape sequence:

`\E~`

The terminal responds with an `\E!`, and seven status bytes followed by a terminator (a carriage-return character). A typical secondary terminal status request and response is shown in the following illustration.



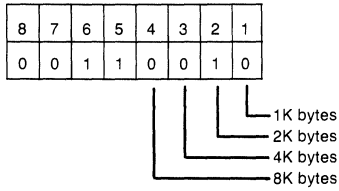
BYTE	ASCII	BINARY	STATUS
7	0	0011 0000	
8	4	0011 0101	
9	0	0011 0000	
10	0	0011 0000	
11	0	0011 0000	
12	0	0011 0000	
13	0	0011 0000	

└── Terminal identifies self

Secondary Terminal Status Example

Secondary Status Bytes

BYTE 7 Buffer Memory (always zero)



Memory installed in addition to display memory that is available for use as data buffers. Note that the HP 9000 Model 20 terminals always return a 0 value.

BYTE 8 TERMINAL FIRMWARE CONFIGURATION

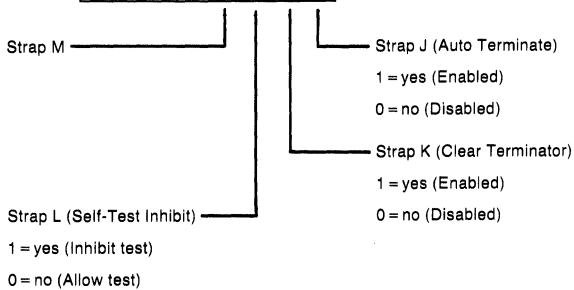
8	7	6	5	4	3	2	1
0	0	1	1	0	1	0	0



APL Firmware does not apply.

BYTE 9 CONFIGURATION STRAPS J-M (always zero)

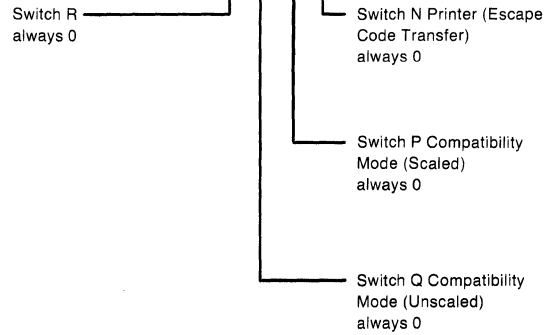
8	7	6	5	4	3	2	1
0	0	1	1	0	0	0	0



Straps J-M do not apply to the terminal.

BYTE 10 KEYBOARD INTERFACE KEYS (N-R)

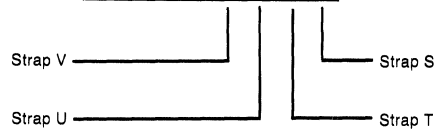
8	7	6	5	4	3	2	1
0	0	1	1	0	0	0	0



Straps N-R do not apply

BYTE 11 CONFIGURATION STRAPS S-V (always zero)

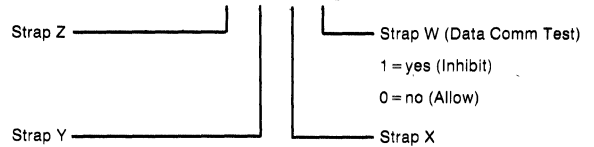
8	7	6	5	4	3	2	1
0	0	1	1	0	0	0	0



Straps S-V do not apply to the terminal.

BYTE 12 CONFIGURATION STRAPS W-Z (always zero)

8	7	6	5	4	3	2	1
0	0	1	1	0	0	1	0



Straps X, Y, and Z do not apply to the terminal.

BYTE 13 MEMORY LOCK MODE (always zero)

8	7	6	5	4	3	2	1
0	0	1	1	0	0	0	0

Table of Contents

MC68000 Assembler on HP-UX	1
Instruction Format	1
In General	1
Symbols	1
Local Labels	1
Opcodes	1
Size Suffixes	2
Expressions	2
Pseudo-Op Syntax and Semantics	3
Interfacing Assembly Routines	4
Linking	4
Calling Conventions	4
Language Dependencies	7
C	7
Fortran	7
Pascal	7
Conversion from the Pascal Language System (PLS)	9
The ADB Debugger	11
Introduction	11
Invocation	11
Command Format	12
Displaying Information	13
Debugging C Programs	15
Debugging a Core Image	15
Setting Breakpoints	17
Advanced Breakpoint Usage	21
Other Breakpoint Facilities	23
Maps	24
Variables and Registers	25
Formatted Dumps	26
Patching	29
Anomalies	30
Command Summary	31
Formatted Printing	31
Breakpoint and Program Control	31
Miscellaneous Printing	31
Calling the Shell	31
Assignment to Variables	31
Format Summary	32
Expression Summary	32
Expression Components	32
Dyadic Operators	32
Monadic Operators	32

MC68000 Assembler on HP-UX

Instruction Format

In General

Assembly instructions are written one per line. Mnemonic operation codes (opcodes) and register symbols must be written in lower case. Upper and lower case characters may not be used interchangeably, that is, it is a case sensitive assembler. Instructions are free format with respect to spaces.

If a label is present, it must start in column one of the line. The opcode must start in column two or later. Blanks are not permitted within the operand field. The first blank encountered after the start of the operand field begins the comment field.

```
Label      move a1,a2          comment field
```

A "*" in column one indicates a comment.

```
*
*      These are comments.
*
```

Symbols

Symbols must begin with an alphabetic character, but may contain letters, numbers, @, # and ... Symbols may contain any number of characters. The restriction is that each instruction must be contained on one line.

* is a symbol having the value of the program counter.

Register symbols are those used to refer to the predefined registers. They are a0...a7, d0...d7, sp, pc, ccr, and sr.

Local Labels

A local label has the form <digit>#. A local label may be used to label any machine instruction. Any number of occurrences of the same local label may occur within an assembly source file. When a local label is referenced, the reference will refer to the nearest declaration of the local label.

Opcodes

Most opcodes and their syntaxes are defined in the *MC68000 User's Manual*. Size suffixes are only allowed for those operations which include a size field in the instruction and for the conditional branch bcc. In addition to the opcodes listed in the manual, the Series 200 will recognize some variants. For the bcc instruction the form jcc may be used. Also, jbsr may be used in place of bsr. In these cases, the assembler will decide the appropriate size for the instruction. No size suffix can be used.

Size Suffixes

Size suffixes are used in the language to specify the size of the operand in the instruction, including addressable locations and registers. All instructions which can operate on more than one data size will assume the default size of word (16 bits) unless a size suffix is used. Size suffixes can also be appended to address register specifications when used in indexed addressing. Operand sizes are defined as follows:

Suffix	Data Unit	Bits
b	byte	8
w	word	16
l	long	32

Expressions

Expressions are evaluated in left to right order, and parentheses are permitted. Symbols which refer to defined labels are permitted in expressions. The value of these symbols is their relative value within the assembled code. The only operations which can be done on these symbols are addition and subtraction. One label can be subtracted from another; the result is an absolute value. A label can be added to an absolute value but not to another symbol. The allowed operators are:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
!	Bitwise <i>or</i>
&	Bitwise <i>and</i>
^	Bitwise <i>exclusive or</i>
<	Shift left
>	Shift right

Pseudo-Op Syntax And Semantics

The following is a list of the commands which direct the assembler to take the described actions. For a list of the machine commands, see the *MC68000 User's Manual*.

align <name>,<modulus>

Create a global symbol of type **align**. When the loader sees this symbol it will create a hole beginning at symbol <name> whose size will be such that the next symbol will be aligned on a <modulus> boundary.

asciz '<string>'

Put a null terminated <string> into the code at this point.

bss

Put the following assembly into the uninitialized data segment.

comm <name>,<size>

Create a global symbol <name>, put it in the bss segment with size <size>.

data

Place the following assembly in the initialized data segment.

dc[.b|.w|.l] <expr>|'<string>'[,<expr>|'<string>']

Place the list of expressions <expr> or strings <string> into the code at this point. Size suffixes may be used to specify the units of storage into which the values will be placed. Default is word. In the case of string literals, the amount of storage needed will be determined by the assembler and each character will be assigned into a unit.

ds[.b|.w|.l] <expr>

The units of space are specified by the size suffix. The number of units is determined by the expression.

equ <expr>

Assigns the value and attributes of the expression to the label.

even

Forces even word alignment.

globl <name>[,<name>]...

Declares the list of names to be global symbols.

include "<name>"|<<name>>

Specifies a file to be merged into the assembly at the point where the instruction is located. The file will be searched for according to the conventions of C (see manual page for cc).

text

Place the following assembly in the code segment.

Interfacing Assembly Routines

In order to know how to use the assembler effectively, it will be necessary to know how to interface to the various higher level languages that the HP-UX Series 200 supports.

Linking

In order for a symbol to be known externally it must be declared in a `global` statement. It is not necessary for a symbol defined externally to be declared in a module. If a symbol is not defined, it is assumed to be externally defined. It is, however, recommended that all external symbols be declared in a `global` statement, since this will avoid possible name confusion with local symbols.

Calling Conventions

All languages currently supported on the Series 200 follow certain conventions regarding the calling of subroutines. These conventions must be followed in order to call or be called by a higher level language.

The calling conventions can be summarized as follows:

- Parameters are pushed in reverse order and taken off in the same order as the procedure call;
- The calling routine pops the parameters from the stack upon return;
- The called routine saves and restores the registers it uses (except `d0`, `d1`, `a0`, `a1`);
- Function results are generally returned in `d0`, `d1`;
- `test.b` required for all stack space used plus that required for the link of any routine called; and
- `link/unlk` instructions are used to allocate local data space and to reference parameters.

These conventions can be more easily understood by means of an example. The best would be to examine the code output by the compiler to do this. This can be easily done using C since it outputs assembly language instructions. Consider the following C program.

```
main()
{
    test(1,2);
}
test(i,j)
register int i, j;
{
    int k;
    k = i + j;
    return k;
}
```

It will produce the following assembly language instructions.

```
1          data
2          text
3          globl  _main
4  _main
5          link   a6,#--F1
6          tst.b  --M1-8(a7)
7          movem.l #__S1,--F1(a6)
8          move.l #2,--(sp)
9          move.l #1,--(sp)
10         jbsr   _test
11         addq   #8,sp
12         jra    L12
13  L12     unlk   a6
14         rts
15  __F1   equ    0
16  __S1   equ    0
17  __M1   equ    0
18         data
19         text
20         globl  _test
21  _test
22         link   a6,#--F2
23         tst.b  --M2-8(a7)
24         movem.l #__S2,--F2(a6)
25         move.l 8(a6),d7
26         move.l 12(a6),d6
27         move.l d7,d0
28         add.l  d6,d0
29         move.l d0,-4(a6)
30         move.l -4(a6),d0
31         jra    L14
32         jra    L14
33  L14     movem.l --F2(a6),#192
34         unlk   a6
35         rts
36  __F2   equ    12
37  __S2   equ    192
38  __M2   equ    0
39         data
```

Things to note are that when the parameters are pushed by the calling routine (`_main`), the second parameter is pushed first and the first parameter is pushed second (lines 8 and 9). When the called routine (`_test`) goes to access the parameters (lines 25 and 26), it finds the first parameter first on the stack and the second parameter second. Line 25 accesses the first parameter and line 26 accesses the second parameter.

Also note that the stack is popped upon return from the subroutine (line 11) and not by the subroutine itself. Since the called routine makes use of `d6` and `d7`, it pushes those registers on the stack (line 24) and then pops them (line 33) before it returns.

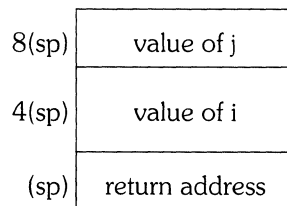
The function result is placed in `d0` before returning (line 30). If the function returned a double precision floating point number, that number would have been placed in `d0` and `d1`.

A `testb` instruction (line 23) is needed before any use is made of stack space in any assembly language routine. The `testb` makes sure that there is enough stack space for this routine. If the test fails, the operating system can detect this and get more stack space for the process. If the test is not done, the program may die unnecessarily with a segmentation violation. The amount of space that must be tested for is the sum of:

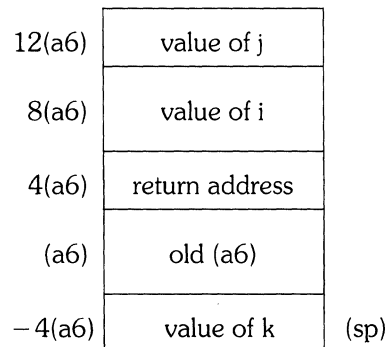
- The amount of space taken by the link instruction;
- The greatest amount of space used for any parameters that may be pushed;
- The constant 8 to account for subroutine jumps and the link which that routine may do.

C and other higher level languages use the `link` and `unlk` instructions (lines 22, 34) in all routines. The `link` instruction is used to allocate local data space and to allow a constant reference point for accessing parameters. The following illustration shows what happens when the `link` instruction on line 22 is executed.

Before the link:



After the link:



Note how the parameter `i` is accessed on line 25. On line 29 the local variable `k` is set. The `link` instruction is not necessary in assembly language code. If it is not there, however, the routine will not show up in a stack backtrace from `adb`. If a `link` instruction is done, an `unlk` must be done before returning.

Language Dependencies

C

In C, all variables and functions declared by the user are prefixed with an underbar. Thus, a variable named `test` in C would be known as `_test` at the assembly language level. All global variables can be accessed through this name using a long absolute mode of addressing. C will always push a four-byte quantity on the stack for pointers and any form of integer (char, short, long). C will always push eight bytes for a floating point number (floats are converted to double).

Fortran

Fortran uses the same naming convention as C, and externals can be accessed in the same fashion. Fortran will always push the address of its parameter for user-defined functions.

Pascal

In Pascal, any exported user-defined function is prefixed by the module name surrounded by underbars. For Pascal, then, a function named `funk` in module `test` would be known as `_test_funk` to an assembly language programmer. If a procedure is declared external as in:

```
procedure Proc; external;
```

all calls to `Proc` will emit a reference to `_Proc`.

Global variables are accessed as a 32-bit absolute relative to the global base. In the example below, the global variable `i1` would be accessed as:

```
move.l test+0x4,d0
```

Following is the example:

```
Pascal [Rev 2.1Ma 4/19/83] test.p                               Page 1

1:D      0 $list 'test.1',tables$
2:D      0 program test;
3:D      1 var
4:D      8 1 i1,i2: integer;
5:D      1 procedure P;
6:D      2 var
7:D     -4 2 j: integer;
8:C      2 begin

Dump of P
j          var lev= 0d2 addr=-00000004 local
P dump complete

9:C      2 end;
10:C     1 begin

Dump of TEST
i1          var lev= 0d1 addr=00000004 longabs globalbase = test
i2          var lev= 0d1 addr=00000000 longabs globalbase = test
P           proc lev= 0d1 entry: 00000000

test       proc lev= 0d0 entry: 00000012

TEST dump complete

11:C      1 end.
```

Pascal will always push a four-byte quantity on the stack for pointers and integers. For a user-defined function, any parameter greater than four bytes will be passed as an address.

The manual pages for these compilers should be consulted for further information. Assembly listings can be generated by C and Fortran. These can be consulted to get valuable information. The only current means for looking at the code generated by Pascal is through the debugger **adb**.

Conversion from the Pascal Language System (PLS)

A translator (**atrans**) is provided to assist in converting from PLS assembly language to HP-UX assembly language syntax. All code to be ported should be run through the translator first. Lines that will require human intervention will be noted by the translator. To see exactly what the tasks are that it performs, check the manual page.

atrans will not detect or alter parameter passing conventions which are pushed in the opposite order on PLS.

as assumes `round 0` for all assemblies. **as** does not generate relative references to external symbols; all external references are absolute. As such, code size can increase when being ported from the PLS to HP-UX.

as does not have support for Pascal modules.

as will accept the same syntax as the PLS assembler for all machine instructions with these exceptions:

Additions:

- **as** will accept `jump cc` where `cc` is a condition code accepted by `bcc`. In this case, **as** will decide the length of the instruction required.
- **as** will accept a greater number of operators for expressions. Parentheses are permitted within expressions.
- **as** will accept an immediate operand for the register list in a `movem` instruction. Needed for compiler.
- **as** will allow numeric value for displacement as in `lz(pc, dB)`. Needed for compiler.
- **as** will accept `<digit> $` to specify a local label.

Differences:

- **as** is a case-sensitive assembler. All opcodes and register names must be listed in lower case.
- **as** accepts `(pc)` to specify pc-relative references. This is the only way to specify pc-relative.
- The PLS assembler will assume pc with index in some cases for a parameter of the form `B(a0)`. **as** will not.

The greatest differences occur in the pseudo-ops that are supported. The only PLS pseudo-ops that are supported are `dc`, `ds`, `equ`, and `include`. The translator will handle some of the other pseudo-ops, but others will have to be handled by hand.

The ADB Debugger

Introduction

ADB is a debugging program that is available on HP-UX. It provides capabilities to look at “core” files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB.

Invocation

ADB is invoked as:

```
adb objfile corefile
```

where `objfile` is an executable HP-UX file and `corefile` is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are `a.out` and `core` respectively. The filename minus (-) means “ignore this argument,” as in:

```
adb - core
```

The `objfile` can be written to if **adb** is invoked with the `-w` flag as in:

```
adb -w a.out -
```

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request `^C` or `^D` (or **CTRL**-**D**) must be used to exit from ADB.

Command Format

The general form of a request is:

[address] [,count] [command] [modifier]

ADB maintains a current address, called *dot*, similar in function to the current pointer in the HP-UX editor. When **address** is entered, dot is set to that location. The command is then executed count times.

Address and count are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type `name` or `_name`; ADB will recognize both forms. The default base for integer input is initialized to hexadecimal, but can be changed.

The following table illustrates some general ADB commands and meanings:

?	Print contents from <code>a.out</code> file
/	Print contents from <code>core</code> file
=	Print value of "dot"
:	Breakpoint control
\$	Miscellaneous requests
;	Request separator
!	Escape to shell

A **CTRL**-**C** will terminate the execution of any command in ADB.

Displaying Information

ADB has requests for examining locations in either **objfile** or **corefile**. The `?` request examines the contents of **objfile**, the `/` request examines the **corefile**.

Following the `?` or `/` command the user specifies a format.

The following are some commonly used format letters:

<code>c</code>	one byte as a character
<code>x</code>	two bytes in hexadecimal
<code>X</code>	four bytes in hexadecimal
<code>d</code>	two bytes in decimal
<code>F</code>	eight bytes in double floating point
<code>i</code>	MC68000 instruction
<code>s</code>	a null terminated character string
<code>a</code>	print in symbolic form
<code>n</code>	print a newline
<code>r</code>	print a blank space
<code>^</code>	backup dot

A command to print the first hexadecimal element of an array of long integers named `ints` in C would look like:

```
ints/X
```

This instruction would set the value of **dot** to the symbol table value of `_ints`. It would also set the value of the dot increment to four. The dot increment is the number of bytes printed by the format.

Let us say that we wanted to print the first four bytes as a hexadecimal number and the next four as a decimal one. We could do this by:

```
ints/XD
```

In this case, **dot** would still be set to `_ints` and the dot increment would be eight bytes. The dot increment is the value which is used by the `newline` command. `newline` is a special command which repeats the previous command. It does not always have meaning. In this context, it means to repeat the previous command using a count of one and an address of dot plus dot increment. In this case, `newline` would set dot to `ints+0x8` and type the two long integers it found there, the first in hex and the second in decimal. The `newline` command can be repeated as often as desired and this can be used to scroll through sections of memory.

Using the above example to illustrate another point, let us say that we wanted to print the first four bytes in long hex format and the next four bytes in byte hex format. We could do this by:

```
ints/X4b
```

Any format character can be preceded by a decimal repeat character.

The count field can be used to repeat the entire format as many times as desired. In order to print three lines using the above format we would type:

```
ints,3/X4bn
```

The `n` on the end of the format is used to output a carriage return and make the output much easier to read.

In this case the value of dot will not be `_ints`. It will rather be `_ints+0x10`. Each time the format was re-executed dot would have been set to dot plus dot increment. Thus the value of dot would be the value that dot had at the beginning of the last execution of the format. Dot increment would be the size of the format: eight bytes. A `newline` command at this time would set dot to `ints+0x18` and print only one repetition of the format, since the count would have been reset to one.

In order to see what the value of dot is at this point the command:

```
,=a
```

could be typed. `=` is a command which can be used to print the value of **address** in any format. It is also possible to use this command to convert from one base to another:

```
0x32=oxd
```

This will print the value `0x32` in octal, hexadecimal and decimal.

Complicated formats are remembered by ADB. One format is remembered for each of the `?`, `/` and `=` commands. This means that it is possible to type:

```
0x64=
```

and have the value `0x64` printed out in octal, hex and decimal. And after that, type:

```
ints/
```

and have ADB print out four bytes in long hex format and four bytes in byte hex format.

To an observant individual it might seem that the two commands:

```
main,10?i
```

and

```
main?10i
```

would be the same.

There are two differences. The first is that the numbers are in a different base. The repeat factor can only be a decimal constant, while the count can be an expression and is therefore, by default, in a hex base.

The second difference is that a `newline` after the first command would print one line, while a `newline` after the second command would print another ten lines.

Debugging C Programs

Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate some of the useful information that can be gotten from a core file. The object of the program is to calculate the square of the variable `ival` by calling the function `sqr` with the address of the integer. The error is that the value of the integer is being passed rather than the address of the integer. Executing the program produces a core file because of a bus error.

Figure 1: C program with pointer bug

```
int ints[]= {1,2,3,4,5,6,7,8,9,0,
             1,2,3,4,5,6,7,8,9,0,
             1,2,3,4,5,6,7,8,9,0,
             1,2,3,4,5,6,7,8,9,0};

int ival;
main()
{
    register int i;
    for(i=0;i<10;i++)
    {
        ival = ints[i];
        sqr(ival);
        printf("sqr of %d is %d\n",ints[i],ival);
    }

    sqr(x)
    int *x;
    {
        *x *= *x;
    }
}
```

ADB is invoked by:

```
adb
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. This request can be used to check the validity of the parameters passed. As shown in Figure 2 we can see that the value passed on the stack to the routine `sqr` is a 1, which is not what we are expecting.

Figure 2: ADB output for program of Figure 1

```

$c
_main+0x30:  _sqr      (0x1)
--start+0x38:  _main     (0x1, 0xFFFFDD0)
$r
Ps         0x4
Pc         0x20CA  _sqr+0x14:  move.l    0xB(a6),--(a7)

d0         0x4900          a0         0x1
d1         0x800           a1         0xFFFFDD0
d2         0x0             a2         0x0
d3         0x0             a3         0x0
d4         0x0             a4         0x0
d5         0x0             a5         0x0
d6         0x0             a6         0xFFFFDAC
d7         0x0             sp         0xFFFFDAC
"sqr+e,5?ia"
_sqr+0xE:      move.l    0xB(a6),a0
_sqr+0x12:     move.l    (a0),--(a7)
_sqr+0x14:     move.l    0xB(a6),--(a7)
_sqr+0x18:     jsr      _almul
_sqr+0x1E:     addq.w   #0x8,a7
_sqr+0x20:
$e
_argc_value:   0x1
_errno:        0x0
_environ:      0xFFFFDDB
_argv_value:   0xFFFFDD0
_ints:         0x1
_ival:         0x1
__pfile:       0x0
__iob:         0x0
__ctype:       0x202020
__sobuf:       0x0
__lastbuf:     0x4E08
__sibuf:       0x0
tb_pwt4:       0x31D1411E
tb_pwt8:       0x30FC4501
tb_bcd:        0x10203
tb_pwt:        0x2F52FBAC
tb_auxpt:      0xAC8062B
tb_pwt:        0x32A50FFD
tb_bin:        0x10203
_end:          0x0
_edata:        0x

```

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location. The instruction printed for the pc does not always make sense. This is because the pc has been advanced and is either pointing at the next instruction, or is left at a point part way through the instruction that failed. In this case the pc points to the next instruction. In order to find the instruction that failed we could list the instructions and their offsets by the following command.

```
sqr+e,5?ia
```

This would show us that the instruction that failed was:

```
_sqr+0x12:move.l(a0),--(a7)
```

This is the first instruction before the value of the pc. The value printed out for register a0 also indicates that a dereference of its value would fail.

The request:

```
$e
```

prints out the values of all external variables at the time the program crashed.

Setting Breakpoints

Consider the C program in Figure 3. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

Figure 3: C program to decode tabs

```
#include <stdio.h>
#define MAXLINE 80
#define YES      1
#define NO      0
#define TABSP    8

char  input[] = "data";
FILE  *stream;
int   tabs[MAXLINE];
char  ibuf[BUFSIZ];

main()
{
    int col, *ptab;
    char c;

    setbuf(stdout,ibuf);
    ptab = tabs;
    settab(ptab); /*Set initial tab stops */
    col = 1;
    if((stream = fopen(input,"r")) == NULL) {
        printf("%s : not found\n",input);
        exit(8);
    }
    while((c = getc(stream)) != EOF) {
        switch(c) {
            case '\t': /* TAB */
                while(tabPos(col) != YES) {
                    putchar(' '); /* put BLANK */
                    col++;
                }
                break;
            case '\n': /*NEWLINE */
                putchar('\n');
                col = 1;
                break;
            default:
                putchar(c);
                col++;
        }
    }
}
```



```

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
    if(col > MAXLINE)
        return(YES);
    else
        return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
    int i;

    for(i = 0; i<= MAXLINE; i++)
        (i%TABSP ? (tabs[i] = NO) : (tabs[i] = YES));
}

```

We will run this program under the control of ADB (see Figure 4) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```

settab+e:b
fopen+e:b
tabpos+e:b

```

set breakpoints at the starts of these functions. The above addresses are entered as `symbol+e` so that they will appear in any C backtrace since the first three instructions of each function is a standard sequence that links in the new function. Note that one of the functions is from the C library.

Figure 4: ADB output for C program of Figure 3

```

adb a.out -
executable file = a.out
ready
settab+e:b
fopen+e:b
tabpos+e:b
$b
breakpoints
count  bkpt          command
0x1    _tabpos+0xE
0x1    _fopen+0xE
0x1    _settab+0xE
:r
Process 11640 created
a.out: running
breakpoint  _settab+0xE:  clr.l  -0x4(a6)
settab+e:d
:c

```

```

a.out: running
breakPoint      _fopen+0xE:      jsr      __findiop
$c
_main+0x48:      _fopen   (0x4E3B, 0x4E3E)
__start+0x2C:   _main    (0x1, 0xFFFFDE0)
_tabs/24X
_tabs:          0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0
                0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0
                0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0

:c
a.out: running
breakPoint      _tabPos+0xE:    cmp.l    #0x50,0x8(a6)
:s
a.out: running
stopped at      _tabPos+0x16:   ble.s   _tabPos+0x1C
<newline>
a.out: running
stopped at      _tabPos+0x1C:   move.l   0x8(a6),d0
<newline>
a.out: running
stopped at      _tabPos+0x20:   asl.l    #0x2,d0
<newline>
a.out: running
stopped at      _tabPos+0x22:   add.l    #0x58A4,d0
<newline>
a.out: running
stopped at      _tabPos+0x28:   move.l   d0,a0
<newline>
a.out: running
stopped at      _tabPos+0x2A:   move.l   (a0),d0
:d*
:c
a.out: running
Process terminated
settab+e;b settab,5?ia
tabPos+e,3:b ibuf/20c
:r
Process 3255 created
a.out: running
settab,5?ia
_settab:        link     a6,#0xFFFFFFFF
_settab+0x4:     tst.b   -0x10(a7)
_settab+0x8:     movem.l #<>,-0x4(a6)
_settab+0xE:     clr.l   -0x4(a6)
_settab+0x12:    cmp.l   #0x50,-0x4(a6)
_settab+0x1A:    breakPoint
breakPoint      _settab+0xE:    clr.l   -0x4(a6)
:c
a.out: running
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
breakPoint      _tabPos+0xE:    cmp.l    #0x50,0x8(a6)
$c
Process 3255 killed

```

To print the location of breakpoints one types:

```
#b
```

The display indicates a *count* field. A breakpoint is bypassed *count-1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function `settab` we see that the breakpoint is set after the instruction to save the registers on the stack. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at `settab` with the addresses of each location displayed.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function `settab`, one types:

```
settab+e:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for `fofen`), ADB requests can be used to display the contents of memory. For example:

```
#c
```

to display a stack trace, or:

```
tabs,3/8X
```

to print three lines of 8 locations each from the array called `tabs`. The format `X` is used since integers are four bytes on the MC68000. By this time (at location `fofen`) in the C program, `settab` has been called and should have set a one in every eighth location of `tabs`.

Advanced Breakpoint Usage

When we continue the program with:

```
:c
```

we hit our first breakpoint at `tabpos` since there is a tab following the “This” word of the data. We can execute one instruction by:

```
:s
```

and can single step again by hitting “carriage return”. Doing this we can quickly single step through `tabpos` and get some confidence that it is working. We can look at twenty characters of the buffer of characters by typing:

```
>buf/20c
```

Several breakpoints of `tabpos` will occur until the program has changed the tab into equivalent blanks. Since we feel that `tabpos` is working, we can remove all the breakpoints by:

```
:d*
```

If the program is continued with:

```
:c
```

it resumes normal execution and continues to completion after ADB prints the message:

```
a.out: running
```

It is possible to add a list of commands we wish to execute as part of a breakpoint. By way of example let us reset the breakpoint at `settab` and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+e:b settab,5?ia
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
tabpos+e,3:b ibuf/20c
```

This request will print twenty character from the buffer of characters at each occurrence of the breakpoint.

If we wished to print the buffer every time we passed the breakpoint without actually stopping there we could type:

```
tabpos+e,-1:b ibuf/20c
```

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+e:b settab,5?ia;rtab/o
```

could be entered after typing the above requests. The semicolon is used to separate multiple ADB requests on a single line.

Now the display of breakpoints:

```
$b
```

shows the above request for the `settab` breakpoint. When the breakpoint at `settab` is encountered the ADB requests are executed.

Note

Setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+e:b .,5?ia  
fopen+e:b
```

will print the last thing dot was set to (in the example `fopen`) not the current location (`settab`) at which the program is executing.

The HP-UX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile> outfile
```

This request kills any existing program under test and starts the `a.out` afresh. The process will run until a breakpoint is reached or until the program completes or crashes.

If it is desired to start the program without running it the command:

```
:e arg1 arg2 ... <infile> outfile
```

can be executed. This will start the process, and leave it stopped without executing the first instruction.

If the program is stopped at a subroutine call it is possible to step around the subroutine by:

```
:S
```

This sets a temporary breakpoint at the next instruction and continues. This may cause unexpected results if `:S` is executed at a branch instruction.

ADB allows a program to be entered at a specific address by typing:

```
address:r
```

The count field can be used to skip the first n breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first n breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by:

```
address:c
```

The program being debugged runs as a separate process and can be killed by:

```
:k
```

All of the breakpoints set so far can be deleted by:

```
:d*
```

A subroutine may be called by:

```
:x address [parameters]
```

Maps

HP-UX supports several executable file formats. These are used to tell the loader how to load the program file. A nonshared text program file is the most common and is generated by a C compiler invocation such as `cc P#M,c`. A shared text file is produced by a C compiler command of the form `cc -n P#M,c`, ADB interprets these different file formats and provides access to the different segments through the maps. To print the maps type:

```
$m
```

In nonshared files, both text (instructions) and data are intermixed. In shared files the instructions are separated from data and `?*` accesses the data part of the `a.out` file. The `?*` request tells ADB to use the second part of the map in the `a.out` file. Accessing data in the `core` file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. Figure 5 shows the display of two maps for the same program linked as a nonshared and shared respectively. The `b`, `e`, and `f` fields are used by ADB to map addresses into file addresses. The `f1` field is the length of the header at the beginning of the file (0x40 bytes for an `a.out` file and 0x800 bytes for a `core` file). The `f2` field is the displacement from the beginning of the file to the data. For a nonshared file with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

Figure 5: ADB output for maps

```
adb a.out.unshared core.unshared
$m
executable file = a.out.unshared
core file = core.unshared
ready
? map `a.out.unshared'
b1 = 0x2000    e1 = 0x20FC    f1 = 0x40
b2 = 0x2000    e2 = 0x20FC    f2 = 0x40
/ map `core.unshared'
b1 = 0x2000    e1 = 0x2400    f1 = 0x800
b2 = 0xFFF400 e2 = 0x1000000 f2 = 0xC00
$u
variables
b = 0x2000
d = 0x400
e = 0x2000
m = 0x107
s = 0xC00
$q
```

```
adb a.out.shared core.shared
$m
executable file = a.out.shared
core file = core.shared
ready
? map `a.out.shared'
b1 = 0x2000    e1 = 0x20FC    f1 = 0x40
b2 = 0xB0000    e2 = 0xB0000    f2 = 0x13C
/ map `core.shared'
b1 = 0x2400    e1 = 0x2B00    f1 = 0x800
b2 = 0xFFF400 e2 = 0x1000000 f2 = 0xC00
$u
```

```

variables
b = 0x2400
d = 0x400
e = 0x2000
m = 0x108
s = 0xC00
t = 0x400
$#

```

The `b` and `e` fields are the starting and ending locations for a segment. Given an address, `A`, the location in the file (either `a.out` or `core`) is calculated as:

$$b1 \leq A \leq e1 \rightarrow \text{file address} = (A - b1) + f1$$

$$b2 \leq A \leq e2 \rightarrow \text{file address} = (A - b2) + f2$$

Variables and Registers

ADB provides a set of variables which are available to the user. A variable is composed of a single letter or digit. It can be set by a command such as:

```
0x32>5
```

which sets the variable 5 to hex 32. It can be used by a command such as:

```
<5=X
```

which will print the value of the variable 5 in hex format.

Some of these variables are set by ADB itself. These variables are:

```

O    last value printed
b    base address of data segment
d    length of the data segment
e    The entry point
m    execution type (0x107 (nonshared),0x108 (shared))
s    length of the stack
t    length of the text

```

These variables are useful to know if the file under examination is an executable or `core` image file. ADB reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, then the header of the executable file is used instead.

Variables can be used for such purposes as counting the number of times a routine is called. Using the example of Figure 3, if we wished to count the number of times the routine `tabPos` is called we could do that by typing the sequence:

```
0>5 tabPos+e,-1:b <5+1>5 :r <5=d
```

The first command will set the variable 5 to zero. The second command will set a breakpoint at `tabPos+e`. Since the count is -1 the process will never stop there but ADB will execute the breakpoint command every time the breakpoint is reached. This command will increment the value of the variable 5 by 1. The `:r` command will cause the process to run to termination. And the final command will print the value of the variable.

`$v` can be used to print the values of all non-zero variables.

The values of individual registers can be set and used in the same way as variables. The command:

```
0x32>d0
```

will set the value of the register `d0` to hex 32. The command:

```
<d0=X
```

will print the value of the register `d0` in hex format. The command `$r` will print the value of all the registers.

Formatted dumps

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are some examples.

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

- `<b` The base address of the data segment.
- `<b,-1` Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format `4o4^8Cn` is broken down as follows:

- `4o` Print 4 octal locations.
- `4^` Backup the current address 4 locations (to the original start of the field).
- `8C` Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as `␣` followed by the corresponding character in the range 0140 to 0177. An `␣` is printed as `␣␣`.
- `n` Print a new line.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (`<d` provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, `dump`, of requests. An example of such a script is:

```
120#w
4095#s
#v
=3n
#m
=3n"C Stack Backtrace"
#C
=3n"C External Variables"
#e
=3n"Registers"
#r
0#s
=3n"Data Segment"
<b,-1/8ona
```

The request `120#w` sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request `4095#s` increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request `=` can be used to print literal strings. Thus, headings are provided in this `dump` program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request `#v` prints all non-zero ADB variables. The request `0#s` sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 7 shows the results of some formatting requests on the C program of Figure 6.

Figure 6: Simple C program for Illustrating Formatting and Patching

```

char   str1[] "This is a character strings";
int    one    1;
int    number 456;
long   lnum   1234;
float  fpt    1.25;
char   str2[] "This is the second character strings";
main()
{
    one = 2;
}

```

Figure 7: ADB output illustrating fancy formats

```

adb a.out.shared -
executable file = a.out.shared
ready
<b,-1?Bona
_str1:          052150  064563  020151  071440  060440  061550  060562  060543

_str1+0x10:     072145  071040  071564  071151  067147  0        0        01

_number:
_number:       0        0710    0        02322  037640  0        052150  064563

_str2+0x4:      020151  071440  072150  062440  071545  061557  067144  020143

_str2+0x14:     064141  071141  061564  062562  020163  072162  064556  063400
<b,20?4o4^8Cn
_str1:          052150  064563  020151  071440  This is
060440  061550  060562  060543  a charac
072145  071040  071564  071151  ter stri
067147  0        0        01      ns@`@`@`@`@`@a

_number:       0        0710    0        02322  @`@`@aH@`@`@dR

_fpt:         037640  0        052150  064563  ? @`@`This
020151  071440  072150  062440  is the
071545  061557  067144  020143  second c
064141  071141  061564  062562  haracter
020163  072162  064556  063400

address not found in a.out file
<b,20?4o4^8t8Cna
_str1:          052150  064563  020151  071440  This is
_str1+0x8:      060440  061550  060562  060543  a charac
_str1+0x10:     072145  071040  071564  071151  ter stri
_str1+0x18:     067147  0        0        01      ns@`@`@`@`@`@a

_number:       0        0710    0        02322  @`@`@aH@`@`@dR

_fpt:         037640  0        052150  064563  ? @`@`This
_fpt:         020151  071440  072150  062440  is the
_str2+0x4:      071545  061557  067144  020143  second c
_str2+0xC:      064141  071141  061564  062562  haracter
_str2+0x14:     020163  072162  064556  063400

address not found in a.out file
<b,a?2b8t^2cn
_str1:          0x54    0x68    Th
0x69    0x73    is
0x20    0x69    i
0x73    0x20    s
0x61    0x20    a
0x63    0x68    ch
0x61    0x72    ar
0x61    0x63    ac
0x74    0x65    te
0x72    0x20    r

$9

```

Patching

Patching files with ADB is accomplished with the **write**, **w** or **W**, request (which is not like the **ed** editor write command). This is often used in conjunction with the **locate**, **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

```
?l value
```

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either **locate** or **write** requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, `file1` is created if necessary and opened for both reading and writing. `file2` can be opened for reading but not for writing.

For example, consider the C program shown in Figure 6. We can change the word “This” to “The “ in the executable file for this program, `ex7`, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request `?l` starts at dot and stops at the first match of “Th” having set dot to the address of the location found. Note the use of `?` to write to the `a.out` file. The form `?*` would have been used for a shared text file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of “Th” and print the entire string. Execution of this ADB request will set dot to the address of the “Th” characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:e arg1 arg2
flag/w 1
:c
```

The `:e` request is used to start `a.out` as a subprocess with arguments `arg1` and `arg2`. If there is a subprocess running ADB writes to it rather than to the file so the `w` request causes `flag` to be changed in the memory of the subprocess.

Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the `link` instruction. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. If a `:s` command is executed at a branch instruction, and the branch is taken, the command will act as a `:c` command. This is because a breakpoint is set at the next instruction and if it is not reached, the process will not stop.

Command Summary

Formatted Printing

`? format` print from `a.out` file according to *format*
`/format` print from `core` file according to *format*
`= format` print the value of `dot`
`?w expression` write expression into `a.out` file
`/w expression` write expression into `core` file
`?l expression` locate expression in `a.out` file

Breakpoint and Program Control

`:b` set breakpoint at `dot`
`:c` continue running program
`:d` delete breakpoint
`:k` kill the program being debugged
`:r` run `a.out` file under ADB control
`:s` single step

Miscellaneous Printing

`$b` print current breakpoints
`$c` C stack trace
`$e` external variables
`$f` floating registers
`$m` print ADB segment maps
`$q` exit from ADB
`$r` general registers
`$s` set offset for symbol match
`$v` print ADB variables
`$w` set output line width

Calling the Shell

`!` call *shell* to read rest of line

Assignment to Variables

`>name` assign `dot` to variable or register *name*

Format Summary

a	the value of dot
b	one byte in hexadecimal
c	one byte as a character
d	two bytes in decimal
f	four bytes in floating point
i	MC68000 instruction
o	two bytes in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next <i>n</i> space tab
u	two bytes as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

Expression Components

decimal integer	e.g. 0d256
octal integer	e.g. 0277
hexadecimal	e.g. 0xff
symbols	e.g. flag _main
variables	e.g. <b
registers	e.g. <pc <d0
(expression)	expression grouping

Dyadic Operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise <i>and</i>
 	bitwise <i>or</i>
#	round up to the next multiple

Monadic Operators

~	not
*	contents of location
-	integer negate

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.

