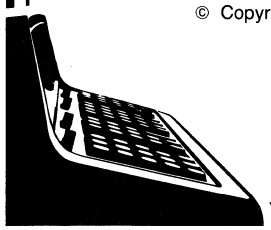
HP 9000 Series 200/500 Computers

HEWLETT PACKARD

HP-UX Concepts and Tutorials
Vol. 1: Text Processing and Formatting

HP-UX Concepts and Tutorials
Vol. 1: Text Processing and Formatting
for the HP 9000 Series 200/500 Computers

Manual Part No. 97089-90004



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.

July 1984...First Edition

Contents

The articles contained in *HP-UX Concepts and Tutorials* 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 and may discuss options or command behavior that does not apply to your system.

HP-UX Concepts and Tutorials has four volumes:

- Volume 1: Text Processing and Formatting
- Volume 2: Program Development and Maintenance
- Volume 3: Shells and Miscellaneous Tools
- Volume 4: Data Communications

This is "Vol. 1: Text Processing and Formatting" and the articles it includes are:

Table of Contents

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 contructs (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.

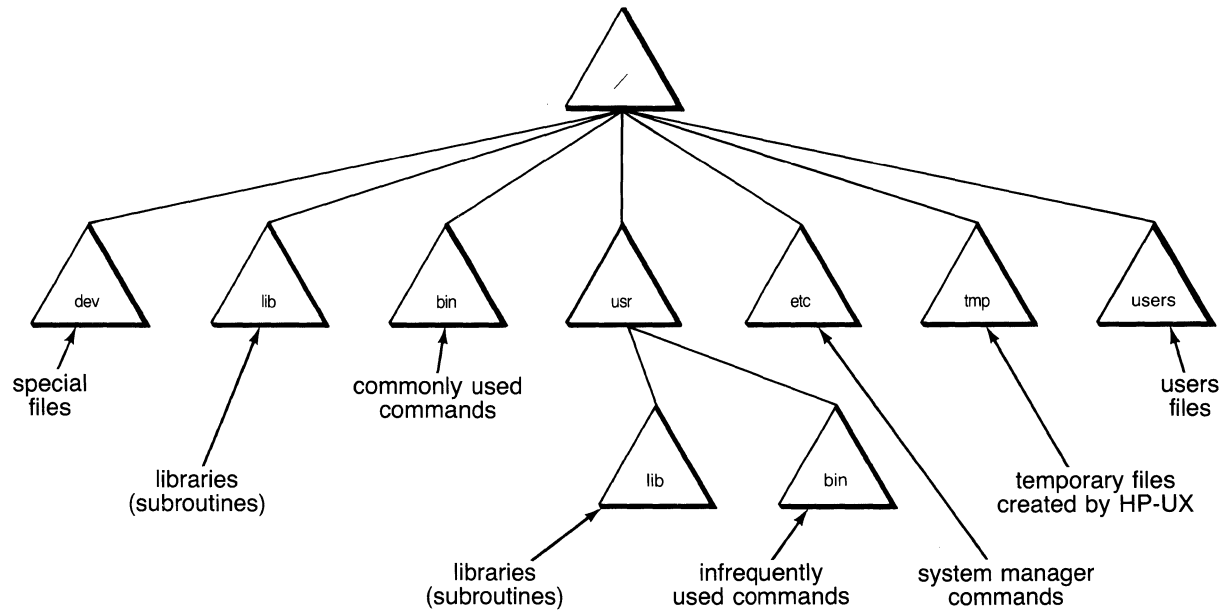
Commands located in the */etc* directory are typically used for system management or system maintenance. Often, these commands can only be accessed by the system manager or super-user. (The super-user is a system user with special capabilities; he is identified by a special user ID.)

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 progam, 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 RETURN
```

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 D while holding the CTRL 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 aquainted 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**.
2. If you are responsible for installing HP-UX, read the **System Administrator Manual**. 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 Concepts and Tutorials** manual. 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 UNIX[1] System**. 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 `guest` 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**. It describes the system administrator's job and his responsibilies. 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 responsiblity (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 Concepts and Tutorials**.
7. The **HP-UX Reference** 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 "Bourne Shell" in the manual, **HP-UX Concepts and Tutorials**.
9. To learn how to write programs in the C Programming Language, read the softcover text, **C Programming Language**.
   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 "HP-UX Programming" in the **HP-UX Concepts and Tutorials** manual 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**.

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

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

Table of Contents

Edit: An Interactive Editor

Edit
An Interactive Editor

Introduction

Text editors are special computer programs that enable you to easily create, preserve, modify, and print text by use of a computer and terminal. Creating text is very much like typing on an electric typewriter. Modifying text involves telling the text editor what to add, change, insert, or delete. Text is printed by giving a command to print all or part of the text file contents. You can also provide special instructions to control the output format if you desire.

This tutorial is divided into four lessons, and assumes no prior familiarity with computers or with text editing. A series of text editing sessions lead you through the basic steps of creating and revising a text file. After scanning each lesson but before beginning the next, try the examples at a terminal to get a feeling for the actual process of text editing. Allow time for experimentation, and you can quickly learn to use a computer for writing and modifying text.

Other HP-UX features are useful besides the text editor. These features are discussed in the book *Introducing the UNIX System* as well as in other tutorials that provide a general introduction to the system. As soon as you are familiar with your terminal keyboard, its special keys, the system login procedure, and know how to correct typing errors, you are ready to start. Let's first define some terms:

Program A group of computer instructions that defines the sequence of steps to be performed by the computer in order to accomplish a specific task. For example, a series of steps to balance your checkbook is a program.

HP-UX A special type of program called an operating system that supervises the computer, peripheral devices, and all programs that use the HP-UX operating system.

Edit The name of the HP-UX text editor that 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 a more extensive editor named *ex*.

File Each HP-UX account is allotted disc storage space for permanent storage of information such as programs, data, or text. A file is a logical collection of data (such as an essay, a program, or a chapter from a book) that is stored and maintained by a computer system. Once you create a file it is kept until you tell the system to remove it. You can create a file during one HP-UX session, log out, then return to use it at a later time. Files contain anything you choose to write and store in them. File sizes vary, depending on individual needs. One file might contain a single number, while another could contain a very long document or program. The only way to save information from one session to the next is to store it in a file where it is kept for later use.

Filename Filenames are used to distinguish one file from another, serving the same purpose as labels on manila folders in a file cabinet. To write or access information in a file, use the name of that file in an HP-UX command. The system automatically determines where the file is located.

Disc Files are stored on a thin circular disc that is coated with magnetic particles similar to magnetic recording tape. The disc may be permanently installed in a disc drive, or it may be a removable flexible disc that resembes a small phonograph record in a thin, square protective container or envelope. Information from the computer (such as your text) is recorded on the disc surface by the disc drive.

Buffer A temporary work space that is available to the user during a text editing session. The buffer is used to build and modify text files. Buffers are analagous to a piece of scratch paper that is discarded at the end of a session after the information it contained has been copied (written) to a permanent disc file.

Session 1: Creating a Text File

Before you can use the editor, you must first log onto the computer so HP-UX can set up communication between your terminal and the editor program. Here is a review of the standard HP-UX login procedure:

If the terminal you are using is directly linked to the computer, turn it on and press (Return). If your terminal uses an acoustic coupler (or modem) and telephone line instead, turn on the terminal, dial the system-access telephone number, then, when you hear a high-pitched tone, place the telephone receiver in the acoustic coupler. If you are using a modem, consult the modem manual for procedures. Press carriage return (or the (Return) key) once, and await the login message:

```
:login:
```

Type your login name (which identifies you to HP-UX) on the same line as the login message, then press (Return). If the keyboard on your terminal supports both uppercase and lowercase, be sure you enter your login name in lowercase. Otherwise, HP-UX assumes your terminal has only uppercase and will not recognize any lowercase letters you may type. When HP-UX types `:login:`, reply with your login name, for example `susan`:

```
:login: susan (Return)
```

(In this example, input typed by the user appears in bold face to distinguish it from information displayed by HP-UX.)

HP-UX responds 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 (Return))

If any of the information you gave during the login sequence was mistyped or incorrect, HP-UX responds with:

```
Login incorrect.
:login:
```

If this happens, start over and repeat the process. When you successfully log in, HP-UX prints the message of the day and eventually presents you with a % at the beginning of a fresh line. The % is the HP-UX prompt symbol that tells you HP-UX is ready to accept a command.

Asking for *edit*

You are ready to tell HP-UX that you want to use *edit*, the text editor program. Now is a convenient time to choose a name for the text file you are about to create. To begin your editing session type `edit` followed by a space, then the filename you have selected, such as *text*. When you have completed the command, press (Return) and wait for *edit*'s response:

```
% edit text
"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 exists. As we expected, it was unable to find such a file since *text* is the name of the new file to be created. *Edit* confirms this with the line:

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

The colon on the next line is *edit*'s prompt, announcing that *edit* expects a command from you. You are now ready to create the new file.

The "Command not found" Message

Suppose you misspelled *edit* by typing *editor*. Your request would be handled as follows:

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

Your mistake in calling *edit* `editor` was treated by HP-UX as a request for a program named *editor*. Since there is no program named *editor*, HP-UX reported that the program or command could not be found. A new % prompt indicates that HP-UX is ready for another command, so you can now enter the correct command.

Summary

Your exchange with HP-UX 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
:
```

Creating text

You can 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 which, in effect, creates text. Most edit commands have two forms: a word that describes what the command does and a shorter abbreviation of that word. Either form can 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 can be abbreviated *a*. Type `append`, then press (Return).

```
% edit text
:append
```

Messages from *edit*

If you make a mistake while entering a command and type something that *edit* does not recognize, *edit* responds 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 receive this message:

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

When you receive a diagnostic message, examine what you typed to determine what part of your command confused *edit*. The message above means that *edit* could not recognize your mistyped command, so the command was ignored. After displaying a new colon prompt, *edit* is now ready to receive a new command.

Text Input Mode

By giving the command *append* (or using the abbreviation *a*), you activated **text input mode**, also known as **append mode**. When you enter text input mode, *edit* responds by doing nothing. No prompts appear during text input mode, your signal to begin entering lines of text. You can type almost anything you want while inputting text lines. Lines are transmitted one at a time to the buffer and held there during the editing session. You can append as much text as you want. When you are through entering new text lines, type a period by itself at the beginning of a new line, then press (Return). This signals the editor to terminate text input mode and return to **command mode**. *Edit* then prompts you for a new command by displaying a colon (:) prompt.

When you leave *append* mode and return to command mode (necessary in order to do any of the other kinds of editing, such as changing, adding, or printing text), *edit* preserves the text you just typed in the editor buffer, so nothing is lost. If you type any other character besides a period by itself on the last line, *edit* treats the line as text instead of an exit command, and will not let you leave *append*. To exit, type a period by itself on a single line terminated by (Return).

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

Let's say that the lines of text you enter are (try to type exactly what you see, including "thiss"):

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

The last line is the period followed by (Return) that gets you out of append mode. If, while typing the line, you hit an incorrect character, you can change the incorrect character by using the (BACK SPACE) key to back up then retype the line beginning with the incorrect character. If you back-space to the first character in the line then press (Return), a blank line is stored in the buffer. Corrections to a line must be done before the line has been completed by a (Return) (changes in lines already typed are discussed in Session 2).

Writing Text to Disc

Text input is now complete. Before you break for lunch, the text should be put in a disc file for safekeeping until the next editing session. Storing the editor's buffer in a disc file is the only way to save information from one session to the next, since the buffer is temporary and is destroyed after the end of the editing session. Thus, learning how to write a file to disc is second in importance only to entering the text. To write the contents of the buffer to a disc file, use the command, *write* (or its abbreviation *w*):

```
:write
```

Edit now copies the buffer to a disc file. If the file does not exist, a new file is created automatically and the presence of a "New File" will be noted. The newly-created file is given the name specified when you entered the editor, in this case, *text*. To confirm that the disc file has been successfully written, *edit* repeats the filename, then gives 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 that were written to the disc are still in the buffer, should you want to modify or add to them.

This ability to write a file to the disc and still continue editing is useful insurance against loss of data during power failures. It is a good idea to periodically write the edit buffer to a permanent file to minimize the risk of losing an hour's work should the power go off for some reason (the risk is actually not as serious as this sounds, because HP-UX has recovery commands that recover all but very little of the file in the event of a power failure).

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* prints:

```
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 followed by the name of the file.

Logging Off

We have done enough for this first lesson on using the HP-UX text editor, and are ready to terminate (quit) the editing session. To do this, type *quit* (or *q*), then press (Return). The terminal display looks like this:

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

The (%) prompt is from HP-UX, telling you that your session with *edit* is over and you can now interact with HP-UX. To end the entire session at the terminal, you must also exit from HP-UX. In response to the HP-UX prompt of "%" press the (CTRL) and (D) keys simultaneously to terminate the session with HP-UX and make the terminal available to 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, a condition that tempts even the most honest of souls.

This is the end of the first session on HP-UX text editing.

Session 2

Login with HP-UX as in the first session:

```
:login:susan (Return)
Password:      (give password then press (Return))
%
```

This time when you type the *edit* command, you can specify the name of the file you worked on last time. Thus, when *edit* starts, it will transfer the original file into its buffer so that you can resume editing the same file. When edit has copied the file into the buffer, it shows the original file name, and lists the number of lines and characters in the file as follows:

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

Your command to edit file *text* caused the editor to copy the 90 characters of text into the buffer. *Edit* now awaits your next command. In this session you learn to append more text to the file, print the contents of the buffer, and change the text in a line.

Adding More Text to the File

To add more to the end of your text, use 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. Why this happens is explained later in this session. This time, 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

Most terminals supported by HP-UX have a DEL (delete) key, sometimes labelled RUN. If you press DEL while working with *edit*, any task the editor is performing is stopped, and the following message is sent to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by **DELETE** or RUN, causing *edit* to prompt you for a new command. If you are appending text at the time the key is pressed, append mode terminates and you are expected to give another command. The line of text that you were typing when the *append* operation was interrupted is lost and is not entered into the buffer.

Making Corrections

If you have read a general introduction to HP-UX, such as *HP-UX User's Guide*, 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 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 ge 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 can 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 can immediately begin to retype the line. This, unfortunately, does not help after you type the line and press Return. To make corrections in completed lines, it is necessary to use the editing commands covered in this and following sessions.

HP-UX and *edit* also support use of BACK SPACE for text corrections. How the backspace key affects the terminal screen display depends on how your terminal or terminal emulator functions. You can look it up in the manual, or just try it out.

Listing Buffer Contents

Having appended text to what you wrote in Session 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 te buffer, the "$"is a special symbol designating the last line of the buffer, and *p* (for *print*) is the command to print from line 1 to the end of the Thus, `1,$p` gives you:

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

You may occasionally place a a character in the buffer that cannot be printed (ASCII control characters are not printed on most output devices). These characters are usually obtained by pressing CTRL and some other key at the same time. When printing lines, *edit* uses a special notation to show the existence of non-printing (control) characters.

Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently holding down the CTRL key while typing `a`. If you asked to have the line printed, EDIT would display:

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

The two-character sequence `'A` indicates that the CTRL key was depressed simultaneously with the "A" key, resulting in a corresponding control character (the apostrophe indicates that CTRL was pressed). The error is easily corrected, as discussed later in this session.

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

Finding Things in the Buffer

You must find something in the buffer before you can change it. To find "thiss" in the text you entered, look at a listing of the lines. *Edit* searches the buffer, looking for the text sequence "thiss", and stops searching when it finds the specified character pattern. You can tell *edit* to search for a pattern by typing the pattern between slash marks:

```
:/thiss/
```

By typing `/thiss/` and pressing Return, *edit* is instructed to search for "thiss" (if *edit* cannot find the pattern of characters in the buffer, it responds "Pattern not found"). When edit finds the characters "thiss", it prints the line where the pattern was found 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

Edit always keeps track of its position in the buffer by identifying the "current line" at the end of each operation. In general, the line that was most recently printed, entered, or changed is considered to be the current position or line in the buffer. The editor assumes the next command is to be applied to the current line, unless you direct it to act in another location (or perform an operation that is not related to the current line). When you bring a file into the editor, the editor is always 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 line. You can refer to your current postion in the buffer by the symbol period (.), usually called "dot". If you type "." then press (Return), you are telling *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 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.
```

Experiment with these commands to ensure 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 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* (not *n* which is used for a different command). You can 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,$nu` lists all lines in the buffer and their corresponding line numbers.

Substitute Command (*s*)

Now that you have found the misspelled word, it is time to change "thiss" to "this". As far as *edit* is concerned, changing text is a matter of substituting one pattern for another. Just as *a* stood for *append*, so *s* stands for *substitute*. Use the abbreviation *s* to reduce the chance of mistyping the *substitute* command. This command instructs *edit* to make the change:

```
2s/thiss/this/
```

First, indicate the line to be changed (2), then type the command (*s*), followed by the characters to be removed (typed between slashes). Finish the line with the characters to be put back in followed by a closing slash mark, then press (Return). Here it is in plain English:

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

If *edit* finds an exact match of the characters to be changed it makes 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 that your instructions could not be carried out. If *edit* finds the characters you want to change, it makes the sutstitution and automatically prints the changed line so you can verify that the correct substitution was made. In the example,

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

line 2 (and line 2 only) is searched for the character pattern "thiss". When the first exact match is found, "thiss" is changed to "this". In reality, since you set the current line number to 2 in an earlier operation, it was unnecessary to specify the number of the line to be changed by this command. In the command:

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

edit assumes that the line where the editor is currently positioned (the **current line**) is to be used. A period can also be used to specify the current line as in the command:

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

although the period is totally unnecessary. In either case, the command without a line number or without a period would have produced the same result as when the line number was specified because the editor was already positioned at the line to be changed. Here is another illustration of substitution.

```
Text editing is strange, but nice.
```

To be a bit more positive, take out the characters "strange, but" so the line reads:

```
Text editing is nice.
```

A command that positions *edit* at that line then makes the substitution is:

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

This command combines the search with a substitution, a perfectly allowable combination. Thus, you do not necessarily have to use line numbers to identify a line to edit. Instead, you can identify the line to be changed by asking *edit* to search for a specified pattern of characters that occurs in the line of interest. The function of each part of the command is as follows:

```
/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"

Note the space after "but" on "/strange, but /". If you do not indicate the space is to be taken out, your line becomes:

```
Text editing is  nice.
```

which looks odd because of the extra space between "is" and "nice". Again, you can see that a blank space is a real character to a computer, and when editing text you need to be aware of spaces within a line just as you 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 conveniet 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 (Return)
```

edit starts with line 1 and continues 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, type the command

```
:z (Return)
```

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

Saving the Modified Text

Now is a good place to pause and end the second session. If you hastily type `q (Return)` to terminate the session, your interaction with *edit* resembles:

```
: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 disc. You are risking the loss of the work you have done during the editing session since the last previous *write* command. Since no previous disc write was performed during this session, everything done during the session would be lost. If you do not want to save the work done during this editing session, you can type *q!* to confirm that you indeed want to end the session immediately, losing the contents of the buffer. However, since you probably prefer to preserve the edited file, use the *write* command as follows:

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

then follow with

```
:q
%logout
```

and hang up the phone or turn off the terminal when HP-UX asks for a login name.

This is the end of the second session on HP-UX text editing.

Session 3

Bringing Text Into the Buffer (*e*)

Login to UNIX and make contact with *edit*. Try to do it without using notes if you can.

Did you remember to give the name of the file you wanted to edit by typing:

```
%edit text
```

or did you type:

```
%edit
```

Both commands activate *edit*, but only the first version can bring a copy of the file named *text* into the buffer. If you forgot to specify the filename, you can recover by typing:

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

The *edit* command which can be abbreviated *e* when you're in the editor, tells *edit* that you want to destroy anything already in the editor's buffer and copy the file *text* into the buffer for editing. You can 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 disc before editing the next file.

Moving Text in the Buffer (*m*)

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

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

This example directs *edit* to move lines 2, 3, and 4 to the end of the buffer following the last line, indicated by ($). When constructing the *move* command, specify the first line to be moved, the last line to be moved, the move command *m*, then the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

commands *edit* to move lines l through 6 (inclusive) to a position immediately following 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 that 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.
```

You can restore the original order by typing:

```
:4,$m1
```

or you can combine context searching and the *move* command for the same result:

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

The danger in combining context searching with the move command lies in the higher probability of making a typing error in such a long command. Typing line numbers is usually much safer.

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 last line in the buffer ($). 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 resulting new current line (.).

If you do not happen to know the line, number you can search for the line 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 containing the indicated text. Once you are sure that you have correctly specified the line 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 the search operation. 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.
```

type

```
:2,3d
```

to specify the range of lines (2 thru 3) and the operation on those lines (*d* for delete).

Again, this assumes that you know the line numbers for the lines to be deleted. If you do not, you can combine the *search* and *delete* commands as follows:

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

This tells the editor to find the first line (following the current line) that contains the characters "And this is some", then delete it and all subsequent lines until it has deleted the line containing "Text editing is nice".

Be Careful

In using the search function to locate lines to be deleted, make absolutely sure that the characters you give as the basis for the search will take edit to the line you want deleted. Edit searches for the first occurrence of the characters starting from where you last edited; that is, from the line you see printed if you type a period (.) then press Return.

A search based on too few characters may result in the wrong line being deleted (if an identical pattern appears elsewhere in the text). For this reason, it is usually safer to specify the search, then delete in a second separate step, at least until you become familiar enough with the editor that you understand how best to specify searches. For beginners, be safe and double-check each command before pressing Return to send the command on its way.

Oops! I goofed. Now what? (*undo*)

The *undo* (*u*) command has the ability to reverse the effects of the last (and only 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. *Undo* affects only those commands that can change the buffer, such as 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 important: the only command that can be reversed by undo is the last "undo-able" command preceding the *undo*.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command deleted the lines that were formerly numbered 2 and 3. Executing *undo* at this time reverses the effects of the deletion, causing those two lines to be restored to their original position in the buffer.

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

Again, as before, edit informs you when the command affects more than one line, and prints the text of the resulting new current line.

More About Dot (.) and Buffer End ($)

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

- Exit from append mode by typing a period (by itself) followed immediately by Return
- Refer to the current line in the editor's buffer.

A period can also be combined with an equal sign to get the number of the line currently being edited (current line):

```
:.=
```

Thus, type `.=` to ask for the number of the current line, or use a colon instead of the equal sign (`.:`) to ask for the text in the current line.

In this editing session, as in the last, the dollar sign was used to to indicate the last line in the buffer for commands such as *print*, *copy*, and *move*. As a command, the dollar sign asks *edit* to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=), *edit* prints 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 (.) through the last line in 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. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few (say, 3) lines ago, you can type:

```
-3P
```

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

```
+2P
```

tells *edit* to print the line which is 2 ahead of our current position. You can 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$
```

copies 4 lines: the line preceding the current line, the current line, and the two lines following the current line, placing them after the last line in the buffer ($).

Try typing a single minus (-). 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 moves you one line ahead in the buffer and prints that line.

If you are at the last line in the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, *edit* reminds 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 then switch to text input mode in order to accept the text that will replace them. Let's assume that you 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 HP-UX text editor.
```

To do so, you can type:

```
:.,.2c
2 lines changed
This text was created with the HP-UX text editor.
.
:
```

The command *1,2c*, specifies that you 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 u86 enters the change command, *edit* notifies you if more than one line is being changed, then places you in text input mode. Any text typed on the following lines is inserted into the position where lines were deleted by the change command. You remain in text input mode until you exit 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 HP-UX.

Session 4

This lesson covers several topics, starting with commands that affect the entire buffer, characters with special meanings, and how to issue HP-UX commands while using the editor. The next topics deal with files, discussing more about reading and writing, and explaining how to recover files lost in a crash. The final section provides leads to other sources of information and other editors that expand beyond *edit*.

Making Commands Global (*g*)

One disadvantage of using the commands in the manner illustrated when searching or substituting is that if you have a number of instances of a word to change, it would appear 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 can give a command such as:

```
:14x/text/material/g
```

to change every instance of "text" in line 14 alone. Note further that 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. Be sure you know what you are telling *edit* to do to the entire buffer. For example:

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

deletes every line containing a blank anywhere in it. This could demolish your document, because most lines contain spaces between words, and thus would be deleted. After executing the global command, *edit* prints a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. Try experimenting with the global command on a small buffer of text to see what it can do for you.

More about Searching and Substituting

Previous examples of using slashes to identify a character string that you want to search for or change have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" you can type either

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

as before, or use 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 that 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 can 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 is 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/
```

you 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: the dollar sign ($), and circumflex (^). ($) 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 circumflex (^) 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.

These 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 HP-UX 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 editor functions, so they require use of HP-UX system commands (also referred to as "shell" commands, because the HP-UX program that processes HP-UX commands is called a "shell"). You do not need to quit the editor to execute an HP-UX command as long as you indicate that it is to be sent to the shell for execution. To use the HP-UX command *rm* to remove the file named *junk*, type:

```
:!rm junk
!
:
```

The exclamation point (!) indicates that the rest of the line is to be processed as an HP-UX command. If the buffer contents have not been written since the last change, a warning is printed before the command is executed. The editor replies with an exclamation point when the command is completed. The *Getting Started with HP-UX* manual describes useful features of the system, and is helpful background when you need to access HP-UX from *edit*.

Filenames and File Manipulation

Throughout each editing session, *edit* keeps track of the name of the file being edited as the current filename (the current filename is 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 you have seen, supplies the current filename. You can have the editor write all or part of its buffer contents to a different file by including the new file name in the *write* command:

```
:wchapter3
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor does not change as a result of the *write* command unless it is the first filename given in the editing session. Thus, using the previous example, the next *write* command that does not specify a file name will write onto the current file, not onto the file *chapter3*.

The File (*f*) Command

To ask for the current filename, type *file* (or *f*). In response, the editor provides updated 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 to a disc file, the buffer is no longer considered modified:

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

Reading Additional Files (*r*)

The read (r) command enables 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 is to be placed, the command *r*, then the name of the file.

```
:$r 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 any file you specify. You are already familiar with writing the entire contents of the buffer to a disc 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 to the file named *ending*. The lines remain in the buffer as part of the document you are editing, so you can 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 your terminal is accidentally disconnected. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login, listing 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 while 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 receive the message "Quota exceeded", you have tried to use more disc 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
```

then seek help. Do not simply leave the editor. If you do, the buffer will be released (and possibly destroyed), 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 unwanted 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 disc.

Further Reading and 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 most 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* tutorial, which is applicable to both *ex* and *edit*. One way to become familiar with *ex* 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 *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* tutorial. 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 accidently 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 Vi Editor* provides a full discussion of visual mode.

Table of Contents

Ex Extended Editor

Ex
Extended Editor

The *ex* editor has many options that can be set to meet individual needs. It is much more comprehensive and more versatile than the *edit* version that uses predefined defaults for some options to better fit the needs of beginning and casual users. In this tutorial, default settings are assumed for all command options unless stated otherwise.

Starting ex

When invoked, *ex* uses the environment variable `TERM` to determine the terminal type. If a `TERMCAP` variable in the environment matches the terminal described by the TERM variable, that description is used. Also if the `TERMCAP` variable contains a pathname (beginning with `/`), the editor seeks the description of the terminal in that file (rather than the default */etc/termcap*). If there is a variable `EXINIT` in the environment, the editor executes the commands contained in that variable. Otherwise, if there is a file *.exrc* in your HOME directory, *ex* reads commands from that file, thus simulating a source command. Option-setting commands placed in `EXINIT` or *.exrc* are executed before each editor session.

The *ex* start-up command has the following prototype:

`ex [-] [-v] [-t <tag>] [-r][-l] [-w<n>] [-x] [-R] [+ <command>] <name> . . .`

where brackets ([ ]) surround optional command parameters. The most common case edits a single file with no options, i.e.:

`ex` *name*

Command-line options function as follows:

-	Suppresses all interactive-user feedback; useful when processing editor scripts in command files.
`-v`	Equivalent to using *vi* rather than *ex*.
`-t`	Equivalent to an initial tag command. Edits the file containing the tag and positions the editor at its definition.
`-r`	Used in recovering after an editor or system crash. retrieves the last saved version of the named file or, if no file is specified, types a list of saved files.
`-l`	Sets up for editing LISP, by setting the showmatch and lisp options.
`-w`	Sets the default window size to n, and is useful on dial-ups to start in small windows.
`-x`	Causes *ex* to prompt for a key that is then used to encrypt and decrypt the contents of the file. The file should have been previously encrypted using the same key, see *crypt*(1).
`-R`	Sets the read-only option at the start.
name	Indicates which file(s) to edit.

An argument of the form + <command> indicates that the editor should begin by executing the specified command. If <command> is omitted, the argument defaults to "S", initially positioning the editor at the last line of the first file. Other useful commands here are scanning patterns of the form */pattern*, or line numbers such as `+100` (which starts at line 100).

File Manipulation

Current File

In normal use, *ex* is used to edit the contents of a single file whose name is specified by the **current** filename. In a typical editing sequence, the name of the file to be edited becomes the current filename, and the original file contents are copied into a buffer which is actually a temporary buffer file. *Ex* performs all editing actions on the buffer file. Changes made to the buffer have no effect on the file being edited unless and until the original file is replaced by the edited buffer contents (by use of a write command). The write operation destroys the original file and replaces it with the edited version.

The current file is almost always treated as having been edited. This means that the buffer file contents 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 original file has not been edited, then *ex* will not normally write on it if it already exists (a "not edited" message is returned when the write operation is attempted).

Alternate File

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

Filename Expansion

Filenames within the editor can be specified using normal shell-expansion conventions. In addition, the character `%` in filenames is replaced by the current file name; the character `#` is replaced by the alternate file name (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).

Multiple Files and Named Buffers

If the command line specifies more than one file to be edited, the first file is edited as previously explained. Command-line arguments and file names for the first and subsequent files to be edited are placed in the argument list (the current argument list can be displayed by using the *args* command). When you are ready to edit the next file in the list, use the *next* command. If you want to destroy the original argument list and associated file names, replacing them with a new list, append the desired new arguments and file names to the *next* command. HP-UX then expands the *next* command with its new arguments. The resulting list of names becomes the new argument list; the old list is destroyed, and *ex* edits the first file on the new list.

Ex has a group of named buffers that are particularly useful for saving blocks of text during normal editing, especially when editing multiple files. These buffers are similar to the normal buffer file, except that only a limited number of operations can be used with them. The buffers have names *a* or *A* through *z* or *Z*. Uppercase and lowercase names refer to the same buffers, but commands *append to* uppercase-named buffers and *replace* lowercase-named buffers.

Read-only Operation

You can use *ex* in read-only mode to look at files that you have no intention of modifying, thus preventing the possibility of accidentally overwriting a file. Read-only mode is active when the **readonly** option is set by:

- Using the `-R` command-line option,
- The *view* command line invocation, or
- By setting the `readonly` option.

  Read-only can be cleared by setting noreadonly (type: `:noreadonly` **RETURN**). It is possible to write, even while in read-only mode, by indicating that you really know what you are doing. You can write to a different file or use the `!` form of write, even while in read-only mode.

Exceptional Conditions

Errors and Interrupts

When errors occur, *ex* prints an error diagnostic and, optionally, rings the terminal bell. If the primary input is from a file, editor processing terminates. If an interrrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, *ex* exits when an interrupt occurs.

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 case) attempts 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. For example, if you were editing the file *resume*, you should change to the directory you were using when the crash occurred, giving the command

```
ex -r resume
```

After checking that the retrieved file is indeed intact, you can write it back over the original unedited file. The system normally sends you mail, telling you when a file has been saved after a crash. The command

```
ex -r
```

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

Editing Modes

Ex has five distinct operating modes:

- **Command mode** where commands are entered when a colon (:) prompt is present and executed each time a complete line is sent.
- **Text-input mode** where *ex* gathers incoming lines of text and places them in the file. Append, insert, and change commands use text-input mode to alter existing text.

  No prompt is printed when you are in text-input mode. To exit this mode, type a period (.) immediately followed by an end-of-line key (**RETURN**). Command mode then resumes.
- **Open** and **visual** modes enable you to perform local editing operations on text in the file. The modes are accessed by commands having the same name. The *open* command displays text, one line at a time, on any terminal, while the *visual* command is designed for CRT terminals that have direct screen-cursor addressing capability so *ex* can use the CRT as a window for file-editing changes.
- **Text insertion** mode operates within *open* and *visual* modes.

These modes are discussed in greater detail in *The Vi Editor.*

Command Structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. Ambiguous abbreviations are resolved in favor of the more commonly used commands (for example, the command *substitute* can be abbreviated *s*, while the shortest available abbreviation for *set* is *se*.

Command Parameters

Most commands accept prefix addresses specifying which line(s) they are to affect. The forms these addresses can take is discussed below. Some commands also accept or require a trailing count specifying the number of lines to be affected by the command (if rounding is necessary, the number is rounded down). Thus the command `10p` prints the tenth line in the buffer while `delete 5` deletes five lines from the buffer, starting with the current line.

Some commands require other information or parameters that are always appended following the command name; for example, option names in a *set* command, a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command as in `1,5 copy 25`.

Command Variants

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

Flags After Commands

The characters #, P, and l can be placed after many commands (a *p* or *l* must be preceded by a blank or tab except in the single special case *dp*). The commands abbreviated by these three characters are 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 can 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.

Comments

Comment commands are ignored by the editor. This feature is useful when making complex editor scripts where explanatory comments are needed. Any line beginning with a double quotation mark (") is treated as a comment and no action results. Comments beginning with " can also be placed at the ends of commands, except in cases where they could be confused as part of text (as in shell escape sequences or in substitute or map commands).

Multiple Commands per Line

Multiple commands can be combined on a single line by separating adjacent commands with a | character. However, global commands, comments, and the shell escape ! must be the last command on a line because they are not terminated by a |.

Reporting Large Changes

Most commands that change the editor buffer contents give feedback whenever the scope of the change exceeds a threshold set by the report option. This feedback helps detect undesirably large changes so that they can be quickly and easily reversed with an *undo*. When using commands that have a 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 the threshold.

Command Addressing

Addressing Primitives

. (period)	The current line, usually the last line affected by the previous command. The default address for most commands is the current line, thus (.) is rarely used alone as an address.
n	The nth line in the buffer file, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", which addresses the entire buffer.
+n −n	An offset relative to the current buffer line. The forms +3, .+3, and +++ are all equivalent. If the current line is 100, they all address line 103.
/<pattern>/ or ?<pattern>?	Search forward (/) or backward (?) respectively for a line containing <pattern> where <pattern> is any regular expression, usually a string of text characters. Searches normally wrap around the end of the buffer. If you only want to print the next line containing <pattern>, the trailing / or ? can be omitted. If <pattern> is omitted or explicitly empty, the last previous regular expression used in a pattern search is substituted for <pattern>.
" ' <x>	Used to locate previously-marked lines. 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 can also be established by the *mark* command, using single lowercase letters <x>. Marked lines are then referred to by <'x>.

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, all but the last one or two are ignored. If the command requires two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses. The default in this case is the current line, so ,100 is equivalent to .,100. Giving a prefix address to a command that expects none produces an error diagnostic.

Command Descriptions

All *ex* commands have the following form:

<address> <command> ! <parameters> <count> <flags>

All parts are optional; the degenerate case is the empty command which prints the next line in the file. To preserve user sanity when operating in visual mode, *ex* ignores a : preceding any command.

In the following command descriptions, the command, its standard abbreviation, and default value (if any) are shown in the left column.

abbreviate <word> <text>
abbr: *ab*

Add the specified abbreviation to the current list. <Word> is the abbreviated form of <text> that is being defined by the command. When using *visual* in input mode, if <word> is typed as a complete word (blanks before and after), *ex* expands the abbreviation, and displays <text> (the expanded form is also used in the buffer).

append <text>
abbr: *a* <text>
Default: current line

Reads input <text> and places it after the specified line. If preceded by period (.) it addresses the last line input or the specified line if no lines were input. If address zero (0) is given, <text> is placed at the beginning of the buffer.

a! <text>
Default: current line

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

args

The members of the argument list are printed, with the current argument delimited by left and right brackets ([ ]).

change <count> <text>
abbr: *c*
Default: current line

Replaces lines specified by <count> with the input <text>. Upon completion, the current line becomes the last line in <text>. If no text is provided, the command is treated as a *delete*.

c! <text>
Default: current line

The variant flag on *change* toggles the setting for the autoindent option during text input.

copy <address> <flags>
abbr: co
Default: current line

A copy of the lines specified by <flags> is placed after <address> which can be zero. Upon completion, the current line (.) addresses the last line of the copy. The *t* command is a synonym for *copy*.

delete <buffer> <count> <flags>
abbr: *d*
Default: current line

Removes the lines specified by <count> and <flags> from the text buffer file, and the line following the last line deleted becomes the new current line. If the deleted lines were originally at the end of the text buffer file, the new last line in the file becomes the current line. If a named <buffer> is specified by a single letter, the deleted lines are saved in that buffer. If the buffer name is lowercase, previous buffer contents are overwritten; if uppercase, the lines are appended to any existing text in the buffer.

edit <file>
abbr: *e* or
ex <file>
(no abbr)

Used to begin an editing session on new file(s). The editor first checks to see if the buffer has been modified since the last write command was issued. If so, a warning is issued and the command is aborted. Otherwise, the command clears the entire editor buffer, makes the named file the current file, and prints the new filename.

After ensuring that this file looks reasonable, the editor reads the source file into its buffer. If the transfer is completed without error, the number of lines and characters in the file is typed. If there were any non-ASCII characters in the file, their non-ASCII high bits are stripped, and any null characters are discarded.

If none of these errors occurred, the file is still 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.

e! <file>

The variant flag on *edit* forces the specified **new** <file> to be transferred to the editor buffer for editing whether or not the current file has been preserved with a write command, and suppresses all complaint messages caused by the forced command.

e+<n> <file>

Causes the editor to begin at line <n> rather than at the last line; <n> can also be an editor command containing no spaces such as */<pattern>*.

file

Prints the current file name and provides the following information: whether the file has been modified since the last write command; whether it is read-only; current line number; number of lines in the buffer; and the relative location of the current line in the buffer (expressed as a percentage).

file <file>

The current filename is changed to <file> which is considered **not edited**.

global/<pattern>/<commands>
abbr: g
Default: all lines

First marks each line among those specified that matches the regular expression in <pattern>. Given <commands> are then executed with the current line initially set to each marked line.

The command list consists of the remaining commands on the current input line and can continue to multiple lines by ending all but the last such line with a slash (/). If <commands> (and possibly the trailing (/) delimiter) is omitted, each line matching <pattern> is printed. Append, insert, and change commands and their associated input are permitted. The (.) terminator normally required for insert, append, and change can be omitted if the end of the <text> associated with the command coincides with the end of the *global* command. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself must not appear in <commands>. The *undo* command is also not permitted there, because *undo* could be used to reverse the entire global command. The options autoprint and autoindent are inhibited during a global (and possibly the trailing / delimiter). 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!/<pattern>/<commands>
abbr: *v*

The variant form of *global* runs <commands> at each line not matching <pattern>.

insert <text>
abbr: *i*
Default: current line

Places <text> before the specified line. Upon completion, the new current line becomes the last line in <text>. If no <text> is given, it is set to the line before the addressed line. This command differs from append only in the placement of text. <Text> must be terminated by a period alone on a single line (except for certain cases in *global*).

i! <text>
Default: current line

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

join <count> <flags>
abbr: *j*
Default: current and next line

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 comma at the end of the line, or none if the first following character is a righ-hand parenthesis. If there is already white space at the end of the line, the white space at the start of the next line is discarded.

j!

The variant of *join* causes a simpler join with no white-space processing. Characters in the lines are simply concatenated, and no spaces are eliminated.

k<x>

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

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 <key> <replacement>

The map command is used to define macros for use in *visual* mode. <Key> should be a single character, or the sequence "#n" where <n> is a digit referring to function key <n>. When the character or key specified by <key> is typed in *visual* mode, the corresponding <replacement> expression is substituted (and displayed). On terminals without function keys, you can type "#n". See *The Vi Editor* for more details.

mark <x>
Default: current line

Places the specified mark <x> on the current line. <x> is a single lowercase letter that must be preceded by a blank or a tab. The marking character <x> can then be used in subsequent commands to address this line. The current line does not change.

move <address>
abbr: *m*
Default: current line only

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

next
abbr: *n*

Starts editing next <file> in argument list.

n!

The variant suppresses warnings about the modifications to the editor buffer not having been written out, and irretrievably discards any changes that may have been made.

n <filelist>
or
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), it is executed after editing the first file in the new list.

number <count> <flags>
abbr: *#* or *nu*
Default: current line

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

open <flags>
or
open/<pattern>/<flags>
Default: current line

Enters intraline editing open mode at each addressed line. If a pattern 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 *The Vi Editor* for more details.

preserve

The current editor buffer is saved just as it would be in a system crash. This command is for emergency use 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*
Default: current line only

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*
Default: current line

Puts back previously deleted or yanked lines. Normally used with delete to move lines, or with yank to duplicate lines. If no buffer is specified, the last deleted or yanked text is restored. By using a named buffer, text can 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, but *ex* issues a warning message if the file has changed since the last write command was issued, and aborts the quit. If you want to save your changes before leaving, give a write command. To discard the changes, use the *q!* command variant to force termination.

q!

Forced *quit* from the editor. Discards changes to the buffer without complaint.

read <file>
abbr: *r*
Default: current line

Copies text from <file> into the editing buffer beginning after the specified line. If no <file> is given, the current file name is used. If <file> is specified, it does not change the current file name unless no current file name exists (in which case <file> becomes the current filename). Sensibility restrictions for the edit command also apply here. If the file buffer is empty and there is no current filename, *ex* treats *read* as an *edit* command.

Address zero 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 last line read becomes the current line.

read !<command>
Default: current line

Places the output of <command> in the editor buffer after the specified line. This is not a variant form of *read*, but 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 an accidental hangup of the phone, a system crash, or after a preserve command. You will be notified by mail when a file is saved (except after a *preserve*).

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>

If <parameter> is not given, only those options whose values have been changed from their defaults are printed. If <parameter> is given, all option values are printed.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecesary 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 set in a single invocation of *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 can be nested.

substitute/<pattern>/<repl>/<options> <count> <flags>
abbr: *s*
Default: current line

On each specified line, the first instance of <pattern> is replaced by the replacement pattern <repl>. If the global indicator option character `g` appears, all instances are substituted. If the **confirm** indication character `c` appears, you are asked to confirm each substitution beforehand. The line to be substituted is typed (with the string to be substituted marked with ' ' characters). To accept the substitution, type `y`. Any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines can be split by substituting new-line characters into the line. The newline in <repl> must be escaped by preceding it with a backslash (\). Other metacharacters available in <pattern> and <repl> are described below.

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 when supported by the terminal driver and operating system.

(.,.) substitute <options> <count> <flags>
abbr: *a*

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

t <address> <flags>
Default: current line

The t command is a synonym for *cpy*.

ta <tag>

The focus of editing switches to the location of <tag>, changing to a different line in the current file where <tag> 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 that can be used by the editor to find the tag; this field is usually a contextual scan using /<pattern>/ to maintain immunity from minor changes in the file. Such scans are always performed as if *nomagic* were 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.

v/<pattern>/<commands>
Default: entire text

A synonym for the global command variant *g!*, running the specified <commands> on each line which does **not** match <pattern>.

version
abbr: *ve*

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

visual <type> <count> <flags>
abbr: *vi*
Default: current line

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 *The Vi Editor* for more details. To exit this mode, type Q.

visual <file>
or
visual +*n* <file>

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

write <file>
abbr *w*
Default: entire file

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.

write>><file>
abbr: *w*>>
Default: entire file

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

w! <name>

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

w !<command>
Default: entire file

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*
Default: current line only

Places the specified lines in the names 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.

z <count>
Default: next line

Print the next count lines, default window.

z <type> <count>
Default: current line

Prints a window of text with the specified line at the top. If type is '-' the line is placed at the bottom; a '.' causes 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.

(<address>,<address>*)*!<command>

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

($) =

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

(.,.)> <*count*> <flags>
(.,.)< <*count*> <*flags*>

Perform intelligent shifting on the specified lines; < shifts left and > shifts 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.

(.,.) & <options> <count> <flags>

Repeats the previous *substitute* command.

(.,.) ~ <options> <count> <flags>

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

Regular Expressions and Substitute Replacement Patterns

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 the the previous regular expression used elsewhere (referred to as a previous scanning regular expression). The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

Magic and Nomagic

The regular expressions allowed by *ex* are constructed in one of two ways dependng 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 disadvatage 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 this discussion of regular expressions assumes that the setting of this option is *magic*.

Summary of Basic Regular Expressions

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

<character>	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.
. (period)	Matches a 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.

<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 '\'.

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 O) 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.

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 nth 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.

Options Descriptions

autoindent
abbr: *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 (**CNTL**-[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.

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 '^' 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 '0' 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
abbr: *ap*
Default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *l*, *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
abbr: *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 ^ ↑ (switch files) or ^| (tag goto) command in *visual*. Note that the *edit* and *ex* commands do not autowrite. In each case, there is a equivalent way of switching when *autowrite* is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for *.I rewind*, *stop!*, *tag!* for *tag*, *shell* for !, and *:e#* and a *:ta!* command from within *visual*).

beautify
abbr: *bf*
Default: nobeautify

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

directory
abbr: *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
(no abbr)
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 suffixes. The suffix r makes the substitution be as in the - command, instead of like &.

errorbells
abbr: *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
abbr: *ht*
Default: ht = 8

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

ignorecase
abbr: *ic*
Default: noic

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
(no abbr)
Default: nolisp

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

list
(no abbr)
Default: nolist

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

magic
(no abbr)
Default: magic for *ex* and *vi*

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
(no abbr)
Default: mesg

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

number
abbr: *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
(no abbr)
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
abbr: *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.

paragraph
abbr: *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
(no abbr)
Default: prompt

Command mode input is prompted for with a ':'.

redraw
(no abbr)
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.

remap
(no abbr)
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
(no abbr)
Default: report = 5 (2 for *edit*)

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
(no abbr)
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 command (double the value of *scroll*).

sections
(no abbr)
Default: sections = SHNHH HU

Specifies the section macros 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.

shell
abbr: *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
abbr: *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
abbr: *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
abbr: *slow*
Terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input or new text to improve throughput when the terminal in use is both slow and unintelligent. See *The Vi Editor* for more details.

tabstop
abbr: *ts*
Default: ts = 8

The editor expands tabs in the output file to be on tabstop boundaries for the purposes of display.

taglength
abbr: *tl*
Default: tl = O

Tags are not significant beyond this many characters. A value of zero (the default) means that all the characters are significant.

tags
(no abbr)
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
(no abbr)
From environment TERM

The terminal type of the output device.

terse
(no abbr)
Default: not invoked

Shorter error diagnostics are produced for the experienced user.

warn
(no abbr)
Default: warn

Warn if there has been 'no write since last change' before a '!' command escape.

window
(no abbr)
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
(no abbr)
Default: not invoked

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
abbr: *ws*
Default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin
abbr: *wm*
Default: wm = O

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *The Vi Editor* in this volume of *HP-UX Concepts and Tutorials* for more details.

writeany
abbr: *wa*
Default: nowa

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

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 less than 512.

Ex Changes - Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 31 to 3.5 of *ex*. Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command Line Options

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

Commands

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 ad 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!".

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.

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

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 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.

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*. O 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/nojustify* mode in *nroff*.

The options *w300*, *w1200*, or *w9600* can be set. They are synonyms for *window*, but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify that you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with

```
:set w300=12w1200=23
```

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. 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 ql 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.

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/user/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

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, ad EXINIT.

Vi Tutorial Update

Deleted features

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.

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.

The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: ":map!".

Change in Default Option Settings

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

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.

The command "Z" is the sae 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.

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.

^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 3l terminals).

The *visual* page motion commands ^F and ^B now treat any preceding counts as number of pages to move, insteadof chages to the window size. That is, 2^F moves forward 2 pages.

Macros

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.

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 useing "map!") *Abbreviate* and *unabbreviate* can be abbreviated to "ab" and "una", respectively.

Table of Contents

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 *vi* 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

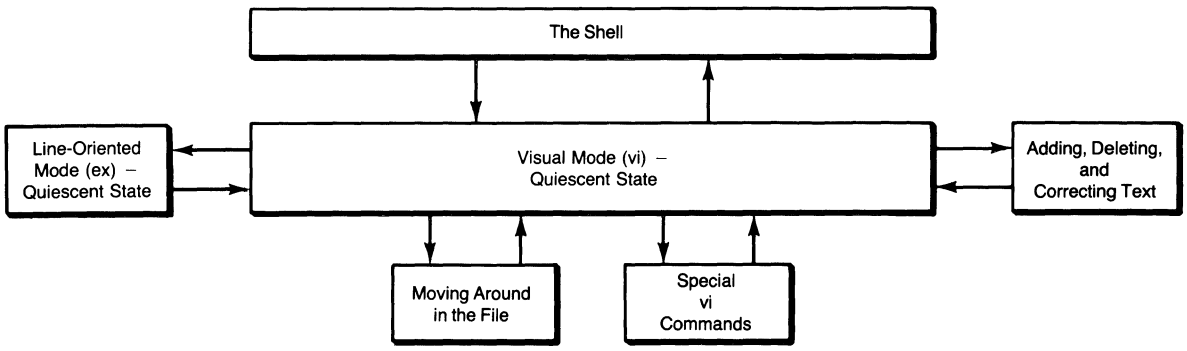
$ **vi** *filename*

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;
*n***G**	command; move cursor to first printable character on line number *n*; default *n* = 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;

f*c* command; move cursor forward to the next instance of the specified character, *c*;

F*c* command; move cursor backwards to the next instance of the specified character, *c*;

t*c* command; move cursor forward to the first character to the left of the next instance of the specified character, *c*;

T*c*	command; move cursor backwards to the first character to the right of the next instance of the specified character, *c*;
;	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,

?\<how[RETURN]

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 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:

c5wyesterday[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 *vi* 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 *vi* 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 *vi* argument list;
!	command; enables portions of the *vi* 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 ccILLUSTRATION 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 *ex* and *vi* initialization constructs; this file is automatically scanned by *ex* 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; refreshe 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 nu|map @ 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 ^ in the following list).

^@ 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 ^@ cannot be part of the file text due to the editor's implementation.

^A Unused.

^B Moves backward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.

^C Unused.

^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 ^D and ^U commands. During an insert, ^D backtabs over **autoindent** white space inserted at the beginning of a line. This white space cannot be backspaced over.

^E Exposes one more line at the bottom of the screen, leaving the cursor at its present position, if possible.

^F Moves forward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.

^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).

^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.

^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.

^J (LF) Same as **down arrow** (see **j**).

^K Unused.

^L (FF) Causes the screen to be cleared and redrawn.

^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, ^M causes the insert to continue onto another line.

^N Same as **down arrow** (see **j**).

^O Unused.

^P Same as **up arrow** (see **k**).

^Q Not a command character. In input mode, ^Q quote the next character, the same as ^V, except that some teletype drivers do not allow ^Q to be seen by *vi*. Use ^V instead.

^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, ^R retypes the current line.

^S Unused.

^T Not a command character. During an insert, with **autoindent** set and at the beginning of the line, inserts **shiftwidth** white space.

^U Scrolls up one page. A preceding integer specifies the number of lines to scroll. This integer is remembered for all future ^D and ^U commands. On a dumb terminal, ^U will clear the screen and redraw it further back in the file.

^V Not a command character. In input mode, ^V 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.

^W Not a command character. During an insert, ^W mimics a **b** 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 ^H).

^X Unused.

^Y Exposes one more line at the top of the screen, leaving the cursor in its present position, if possible.

^Z Unused.

^[(ESC) Cancels a partially formed command, such as a **z** command when no following character has yet been given. It also terminates inputs on the last line (read by commands such as **:**, /, and **?**), 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.

^\ Unused.

^] Searches for the word which immediately follows the cursor. It is equivalent to typing the *ex* command **:ta**, followed by that word, followed by RETURN.

^^ (Control-^), equivalent to the *ex* command **:e #**, 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 **:w** before ^^ will work. If you don't want to write the file, then do a **:e!** # instead).

^_ Unused.

SPACE Same as **right arrow** (see **l**).

! An operator which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by RETURN. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. Thus, **2!}sort**, followed by RETURN, sorts the next two paragraphs by running them through the *sort* command. To read a file or the output of a command into the buffer use **:r**. To simply execute a command use **:!**.

" Precedes a named buffer specification. There are named buffers **1 – 9** used for saving deleted text, and named buffers **a – z** into which text can be placed.

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 \ to insert it, since it normally backs over the last input character you gave.

$ Moves to the end of the current line. If you execute **:set list**, then the end of each line will be shown by printing a $ after the end of the displayed text in the line. Given a count, $ advances to the end of the line that many lines from the current line (i.e. **3$** advances to the end of the line two lines after the current line).

% Moves to the parenthesis or brace which balances the parenthesis or brace at the current cursor position.

& A synonym for the *ex* command, **:&**.

' When followed by another ', 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 **a** – **z**, 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 **^i**.
J	Joins lines together, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the line to be appended is). 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 / or ?, 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 *vi* and goes to *ex* mode. In this mode, whole lines form commands, ending with RETURN. All : commands can be given; the editor supplies the : prompt.
R	Replaces characters on the screen with characters you type (overlay fashion). Terminates with ESC.
S	Changes whole lines; a synonym for **cc**. 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 **p** or **P**. 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 **:x**). 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 ^L (formfeed). Lines beginning with { also stop [[, making it useful for looking backwards through C programs. If the **lisp** option is set, [[also stops at each (it finds at the beginning of a line.

\ — Unused.

]] — Moves forward to a section boundary (see description of [[).

^ — Moves to the first non-space character on the current line.

_ — (Underscore) Unused.

` — When followed by a `, 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

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
texct 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.
.
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 *ed* without a file name argument;
ed *file*	command; invokes *ed* with a file name argument;
P	command; enables/disables *ed* prompt (*);
h	command; explains the last question mark given by *ed*;
H	command; enables/disables verbose error messages; explains the last question mark given by *ed*, 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;
+*n*	operator; increments dot by *n*; default *n* = 1;
–*n*	operator; decrements dot by *n*; default *n* = 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;
n	metacharacter; strips away the special meaning (if any) of the character *n* 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;
*n**	metacharacter; matches zero or more adjacent occurrences of the character *n* 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 *not* enclosed 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 *ed*	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 subsitution.
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 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 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 yourrr 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 yourrr 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
texct 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
texct 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[iau]t**/ 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 *not* selected in the entire file;
V	command; interactive global; on each line *not* 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
texct 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       texct 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
texct 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
texct 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
texct 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,/texct/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
texct 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
texct 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
texct 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.
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,

/texct/
texct entry mode. In command mode, the edytor interprets
s/texct/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 "texct", and then replaces "texct" 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,y***g***/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,*y***G**/*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 *ed*;
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
0r !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

Introduction

Sed is a non-interactive context editor that runs on the HP-UX operating system. It is designed to be especially useful in three cases:

- To edit files too large for comfortable interactive editing;
- To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- To perform multiple "global" editing functions efficiently in one pass through the input.

The remainder of this article explains *sed* operation and use.

Since only a few lines of the input reside in main memory 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 the attendant risk of errors. *Sed* running from a command file is much more efficient than most or all known editors even if that editor can be driven by a pre-written script.

The main interactive editor functions lost when using *sed* are the 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 HP-UX editor, *ed.* Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed.* even seasoned users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without carefully reading this article.

The most striking family resemblance between the two editors is in the class of patterns ("regular expressions") they recognize; both are nearly identical; having virtually identical code.

Overview of Operation

Sed 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 command-line flags (discussed later in this section.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; address format is explained in the Addresses section of this article. Any number of blanks or tabs can separate the addresses from the function. The function must be present; available commands are discussed in the Functions section of this article. Arguments may be required or optional, depending on which function is being used. Refer to Functions section for details.

Tab characters and spaces at the beginning of lines are ignored.

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 Functions section)
- **−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.

Order of Application of Editing Commands

Before any input file is opened or any editing is done, all the editing commands are compiled into a form that 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 which is also the general order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command being 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 Functions section). 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.

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

Examples

The examples shown throughout this tutorial are all based on 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
```

quits after copying the first two lines of input text, and produces the following output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

Addresses: Selecting lines for editing

Addresses are used to determine which lines in the input text or file(s) are to be affected by editing commands. Addresses can be either line numbers or context addresses.

Several commands can be associated with a single address or address-pair by enclosing the group of commands between a pair of curly braces ("{ }").

Line-number Addresses

Line-number addresses are used to specify which lines (in numerical sequence) in the input text are to be modified by the associated *sed* editing commands. As each line is read from the input, an integer decimal line-number counter is incremented. This counter is used to match input lines to the addresses in *sed* commands; when the current line number matches the *sed* address, the command is executed on that line. When editing multiple input files, the counter continues to increment with each line, and does not reset as successive files are opened.

As a special case, the character `$` specifies the last line of the last input file.

Context Addresses

A context address is a pattern (regular expression) enclosed between a pair of slashes (/<regular expression>/). <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 a null character occurring at the beginning of a line.
3. A dollar-sign (`$`) at the end of a regular expression matches a null character occurring at the end of a line.
4. The characters `\n` match an **embedded** 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 a 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 Substitute Function and in the next item of this list.
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 input line's pattern space.

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. Any command having 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

Functions

All functions are named by a single character. In the following summary, the single-character function name is listed in the first column. Possible arguments are enclosed in angles (< >), followed by an expanded English translation of the single-character name. The second column contains the number of addresses allowed with that function and an expanded English translation of what thefunction does. The angles around the arguments are **not** part of the argument, and should not be included in actual editing commands.

Whole-line Oriented Functions

`d`
(delete lines)

(2 addresses allowed) Deletes all lines matched by address(es) and does not write them to the output. **Side effect:** 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.

`n`
(next line)

(2 addresses allowed) 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.

`a\`<text>
(append lines)

(1 address allowed) Causes <text> to be written to the output after the line matched by its address. This command is inherently multi-line; *a* must appear at the end of each 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 each 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> gets written to the output regardless of what later commands do to the line which triggered it. The triggering line can be deleted entirely, but <text> will still be written to the output.

<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.

`i\`<text>
(insert lines)

(1 address allowed) 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.

`c\`<text>
(change lines)

(2 addresses allowed) 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.

C can have two addresses, and therefore affect multiple lines. When two addresses are present, 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 is placed **before** the text of the *a* or *r* function (*r* is described in the Input-Output Function section).

Note: In output text produced by these functions, leading blanks and tabs are eliminated, as in all *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash (the backslash does not appear in the output).

Example:

The following commands, when applied to our example input text:

```
n
a\
XXXX
d
```

produce:

```
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
```

Substitute Function

The *substitute* function changes parts of lines selected by a context search within the line. Command structure is as follows:

s<pattern><replacement><flags>

- (2 addresses allowed) 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> (discussed previously). The only difference between <pattern> and a context address is that the context address must be delimited by slash (/) characters; <pattern> can be delimited by any character other than space or newline.
- 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 in a given substitution command.)
- The <replacement> is not a pattern, and the characters which are special in patterns have no significance in <replacement>. Instead, other characters are special:

<Replacement> Special Characters

`&` 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 between `\(` and `\)`. 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 (`\`).

Substitute Function Flags

The <flags> argument can 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. Note 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 combined total of 10 different file names can be mentioned after `w` flags and *w* functions.

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 Khan

and the command:

```
/X/s/an/AN/gp
```

produces:

In XANadu did Kubhla KhAN

Input-output Functions

`p` (print)

(2 addresses allowed) Writes the addressed lines to the standard output file at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

`w` <filename> (write on <filename>)

(2 addresses allowed) 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 combined total of ten different files can be mentioned in write functions and `w` flags after *s* functions.

`r` <filename> (read the contents of a file)

(1 address allowed) Reads the contents of <filename> and appends <filename> data immediately following the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched the address.

If *r* and *a* functions are executed on the same line, the text produced by the *r* and the *a* functions 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 an *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 *note1* has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was
the grandson and most eminent successor of Genghiz (Chingiz)
Khan, and founder of the Mongol dynasty in China.
```

The following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
   Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was
   the grandson and most eminent successor of Genghiz (Chingiz)
   Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple-input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing embedded newline characters; they are intended principally to provide pattern matches across lines in the input.

`N` (next line)	(2 addresses allowed) The next input line is appended to the current line in the pattern space; the two input lines are separated by an embedded newline character. Pattern matches can extend across the embedded newline character(s).
`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 character was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.
`P` (Print first part of the pattern space)	Print up to and including the first newline character in the pattern space.

The *P* and *D* functions are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

`h` (hold pattern space)	(2 addresses allowed) Copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).
`H` (Hold pattern space)	(2 addresses allowed) Appends the contents of the pattern space to the contents of the hold area. The former and new contents are separated by a newline character.
`g` (get contents of hold area)	(2 addresses allowed) Copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).
`G` (Get contents of hold area)	(2 addresses allowed) Appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline character.
`x` (exchange)	(2 addresses allowed) Interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\en/  :/
```

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
```

Flow-of-Control Functions

These functions do not alter the input lines, but control the application of functions to the lines selected by the address part.

`!`
(Don't)

(2 addresses allowed) Causes the next command (written on the same line), to be applied to all and only those input lines **not** selected by the adress part.

`{`
(Grouping)

(2 addresses allowed) 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.

`:`<label>
(place a label)

(No address allowed) Marks a place in the list of editing commands that can be referred to by *b* and *t* functions. <Label> can be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile-time error diagnostic is generated, and no execution is attempted.

`b`<label>
(branch to label)

(2 addresses allowed) 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 error 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 then done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

`t`<label>
(test substitutions)

(2 addresses allowed) 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.

Miscellaneous Functions

`=`
(equals)

(1 address allowed) Writes to the standard output the line number of the line matched by its address.

`q`
(quit)

(1 address allowed) Writes the current line-to-be-written to the output (if it should be), plus any appended or read text to be written, then terminates execution.

Table of Contents

AWK: A Programming Language for Manipulating Data

Awk: A Programming Language for Manipulating Data

Introduction

Awk is a useful tool for manipulating data and text. Unlike the HP-UX commands that do similar work, *awk* comprises its own programming language. This lets you process input in various ways, such as:

- Generate reports on the contents of files
- Transform the text or data within files
- Manipulate columnar data
- Search files for specific patterns

With *awk*'s ability to search files and generate reports, you can treat some of your ordinary files as databases. The terminology used in *awk* – "records" and "fields" – reinforces this idea.

The *awk* programming language includes such constructs as **for**, **while**, and **if-else**, as well as a set of built-in functions and variables. The language resembles the C programming language. If you are familiar with C, you should be able to master *awk* almost immediately. If you don't know C, you should still find *awk* easy to learn and use.

Awk is named for its designers: Alfred V. **A**ho, Peter J. **W**einberger, and Brian W. **K**ernighan, from Bell Laboratories. For a detailed discussion written by these people, read "Awk–A Pattern Scanning and Processing Language," published by Bell Labs in 1978 and available in many technical libraries.

This article is for the user who is familiar with HP-UX and who has used a programming language. There are examples throughout the article; you should try them as you encounter them. You should take time now to create a small input file, using any of the HP-UX editors, or by typing the following command line:

```
$ cat >hello.awk
```

The *cat* command followed by > allows you to type text directly into the file *hello.awk* from the keyboard. The filename is arbitrary; the *.awk* suffix is just a reminder for you and is optional.

Now type in

```
Hello, world!
Howdy, partner!
```

End the file by typing **CONTROL** and [D] together (the end-of-file character) and then pressing **RETURN**. (**CONTROL** may be marked **CNTL** or [CTRL] on your keyboard; **RETURN** is marked [ENTER] on some keyboards with HP-UX overlays.) The shell prompt should reappear; your example file is ready to use.

The Command Line

You can program *awk* entirely on the same line with the prompt. This is often done in practice. The format is:

```
$ awk 'awk_program' input_filename
```

The dollar sign at the begining of this command line represents the shell prompt in this tutorial. Your shell prompt may be different.

Command-line *awk* programs must be surrounded by single or double quotes, so that the shell will see the whole program as a single argument to the command. Single quotes prevent the shell from interpreting any special characters you may have included in the *awk* program. All examples in this tutorial use single quotes. For more information on shell quoting rules, read "UNIX Programming" in Volume 2 of *HP-UX Concepts and Tutorials*.

If your *awk* program exceeds one line, you can type a backslash (\), then press the **RETURN** key, and continue typing the program. For example:

```
$ awk 'awk_pro\ RETURN
> gram' in\ RETURN
> put_filename RETURN
output of program
  $
```

The > is an auxiliary prompt (which may be different on different systems or shells) that tells you you're still typing a single logical command line.

This maneuver is called *escaping the newline character*. You can use it when invoking any command from the shell.

For some applications you may want to write *awk* programs that are many lines long. It makes sense to store such long programs, and any programs that you often use, in separate files. Note that no compilation step is necessary. The file doesn't have to be executable, just readable.

To invoke *awk* using a program stored in a separate file, use the **−f** option:

```
$ awk -f awk_program_name input_filename
```

You can give an *awk* program input from your keyboard (standard input) by typing a dash ("−") instead of an input filename. Keyboard input is terminated by typing **CONTROL**-D. Your command line would look like this:

```
$ awk 'awk_program' -
```

or

```
$ awk -f awk_program_name -
```

An example of this procedure is shown in the Annotated Example, "Doing Calculations."

Structure of Awk Programs

Awk programs are built of one or more *statements* that have the general form:

pattern {action}

For every line in the input that matches the pattern, the specified action is executed. The action part is always enclosed in braces.

You can specify multiple actions within the action part by separating them with semicolons or newline characters (typing **RETURN** creates a newline character).

Awk processes input one line at a time. For *each* line of input, *awk* scans *all* the patterns in the program. Whenever it finds a pattern that matches the line of input in question, it executes the associated action.

An *awk* statement may consist of the pattern or the action or both. A pattern without an action prints out each input line that matches the pattern (this is the default action); an action without a pattern executes the action on every line of input (the default pattern matches anything).

Predefined Variables

The input is made up of a series of *records*. The default record separator is a newline character; by default, each input line is a record.

Records are divided into *fields*; the default field separator is white space (tabs or blanks). So the input

```
Hello, world!
```

consists of one record (one line) and two fields (the strings "Hello," and "world!", which are separated by a blank).

The output is also made up of fields and records. The default output field separator is a blank and the default output record separator is the newline character.

The variables *FS* and *RS* contain the current input field and record separators; the output separators are in *OFS* and *ORS*. You can change any of them at any time by simply assigning them any single character value.

On the command line, you can use the argument **−Fc**, which sets *FS* to the character value *c*. Use assignment statements (such as RS = "@") to specify new values for any of the other predefined variables or as an alternate way to change *FS*. (Be sure to put double quotes around new separators to ensure that they are interpreted correctly.)

Each field is designated by a *field variable*. In the first record of *hello.awk*, the string "Hello," is stored in the field variable *$1* and "world!" is stored in the field variable *$2*. In general, field *n* of the current record is stored in the variable *$n*. The whole current record is stored in *$0*.

A predefined variable called *NF* contains the number of fields in the current record. The number of the record currently being processed is stored in *NR*; you can find out how many records are in the input by printing *NR* at the end of your program.

Output

The simplest type of *awk* program prints out each line in an input file that matches a specified string. Try this command:

```
$ awk '/Hello/' hello.awk
```

There is no action supplied here, so each record (line) that contains "Hello" somewhere within it is printed. Note that the string is surrounded by slashes, and that the whole *awk* program is surrounded by single quotes. You must always use these slashes around patterns which consist of strings that are regular expressions (described in the section of this article entitled "Regular Expressions and Special Characters"), and you should use single quotes around a command-line program so the shell will see it as one argument and not attempt to interpret any special characters that may be lurking within the program.

The output for the above command is the matching record:

```
Hello, world!
```

The following program contains an action, but it does the same thing as the above actionless program:

```
$ awk '/Hello/ {print $0}' hello.awk
```

This is an example of the **print** action. Since *$0* refers to the entire record, this program prints every record containing "Hello" on the standard output. To print out the second and first fields, in that order, of each record containing "Hello", type:

```
$ awk '/Hello/ {print $2, $1}' hello.awk
```

and you'll get:

```
world! Hello,
```

The comma between the field arguments tells *awk* to put an output field separator (a space by default) between the output fields. Without the comma, the fields would be concatenated (run together).

Redirecting Output to Files

You can send the output of the **print** action into files by using > or >>. The program

```
$ awk '/Hello/ {print $1 >"file1"; print $2 >"file2"}' hello.awk
```

writes the first field, "Hello,", into *file1* and the second, "world!", into *file2* (creating the files if necessary). You must put double quotes around the file names. The program:

```
$ awk '/Hello/ {print $1 >>"file1"}' hello.awk
```

appends the first field to *file1* rather than overwriting it, so now *file1* contains:

```
Hello,
Hello,
```

The file name to which you divert your output may also be a variable or a field. The action:

```
$ awk '/Hello/ {print NF > $2}' hello.awk
```

uses *$2* for the filename. You should take care in cases like this one that the value assigned to the variable is a valid file name. If it is not, you will get an error message and the program will abort.

Formatting Output

You can format your output with the **printf** statement. The *awk* **printf** statement is identical to the **printf** library routine used in the C programming language. The statement's structure is

printf *format, expr, expr, ...*

The format for the list of expressions is specified in the *format* argument. **Printf** prints the expressions in the specified format. For example,

```
$ awk '{printf "%7s  %10.3f\n", $1, NF}' hello.awk
```

prints *$1* (the first field) as a seven-character string, and *NF* as a floating point number in a ten-digit field width with three digits after the decimal point. Try this and get:

```
Hello,       2.000
```

The newline character is **n**, which appears at the end of the *format*. You must specify all spaces, separators, and newlines that you want in the output. Note that you don't have to specify a newline when using **print**, because **print** automatically appends the output record separator (by default, a newline) to its output string.

For a full discussion of **printf**, look in McGilton and Morgan's *Introducing the UNIX™ System*, Kernighan and Ritchie's *The C Programming Language*, or the article "Using the C Library Routines" in Volume 2 of *HP-UX Concepts and Tutorials*. (These are listed in a reference section at the end of this tutorial.)

Details of Awk Programming

The full structure of *awk* programs includes optional statements labeled by the special patterns *BEGIN* and *END*:

```
BEGIN     { action }
  .           .
  .           .
  .           .
pattern   { action }
  .           .
  .           .
  .           .
END       { action }
```

The action in the *BEGIN* statement is executed once before any of the input has been read (hence before any patterns are evaluated). The action in the *END* statement is executed once after all the input has been read. These special statements give you opportunities to set parameters before the program begins or to process or tabulate data after *awk* has seen all of the input. For example,

```
BEGIN {OFS = @}
END   {print NR}
```

changes the output field separator to "@" before any input is read, and prints out how many records are found in the input after all of it has been read.

Designing Patterns

You have many options for writing *awk* patterns, including:

- Regular expressions
- Relational expressions
- Combinations of expressions
- Boolean expressions
- Ranges of patterns

You have a complete set of operators and special characters with which to build patterns.

Regular Expressions and Special Characters

Patterns can be made from regular expressions. Regular expressions are always enclosed in slashes. A simple pattern is a string enclosed in slashes:

```
/world/
```

If entered as a program (`$ awk '/world/' hello.awk`) this expression would print out all lines in an input containing occurrences of "world", both as a field alone (a complete word) and as part of a field, such as "worldly" or "world!"

Between the slashes that delimit regular expressions, you can use most of the standard special characters (or metacharacters) that are recognized by the *ed* editor and by the shell. The available special characters for use between slashes in regular expressions are:

|	perform a logical OR of the regular expressions on either side of the vertical bar.
+	match if there are one or more occurrences of the preceding regular expression.
?	match if there are zero or one occurrence(s) of the preceding regular expression.
[ ]	match any of the characters inside the brackets.
[*x*-*x*]	match any character in the lexical range bounded by the characters on either side of the dash. The range is enclosed by the brackets.
^	match only if the matching regular expression is found at the beginning of the line.
$	match only if the matching regular expression is found at the end of the line.
*	match any combination of characters, including a null string.
.	match any one character in the position of the period.
\	turn off the special meaning of the next character, so the character can represent itself (a maneuver known as escaping the character).
( )	group the evaluation of regular expressions.

For example,

```
$ awk '/^main/' c_program.c
```

matches records beginning with "main", and:

```
$ awk '/Albuquerque|Santa Fe/' article_about_NM
```

matches records containing a reference to either Albuquerque or Santa Fe.

To turn off a special character's special meaning, precede it with a backslash in the expression. For example,

```
/\/.*\//
```

matches any string of one or more characters that is enclosed in slashes.

You can abbreviate a sequence of characters in a string. This is called *character class abbreviation.* For example,

```
$ awk '/[Hh]ello/' hello.awk
```

matches:

```
Hello, world!
```

but it would also have matched a record containing:

```
Well, world, hello!
```

The sequence `[a-zA-Z0-9]` would match all letters, both upper and lower case, and all digits. To use such ranges you need to understand how your character set is arranged. McGilton and Morgan explain this in their book.

In patterns, you can specify that a field or variable (an expression) matches a regular expression using the tilde character "~" to mean "match" and "!~" to mean "don't match." For example, the pattern:

```
$ awk '$1 ~ /[Hh]ello/' hello.awk
```

matches all records that contain either "Hello" or "hello" in the first field. This pattern also matches records containing "Othello" in the first field. To *reject* records with "Hello", use

```
$ awk '$1 !~ /[Hh]ello/' hello.awk
```

Relational Expressions

In *awk* patterns, you can use the relational operators <, <=, ==, !=, >=, and > between expressions. For example, the pattern

```
$ awk '$2 >= $1 + 100' filename
```

selects lines in which the second field is numerically at least 100 greater than the first field.

Relational operations are always *numeric* comparisons (as in the above example) unless *both* operands are strings; in that case a *string* comparison is made. Fields are treated as strings unless there is information to the contrary, so

```
$ awk '$1 > $2' filename
```

automatically performs a string comparison on the first two fields, matching if *$1* has a larger character value (in ASCII) than *$2*.

Note that the regular expressions in the last two examples of the Regular Expressions and Special Characters section match the string "Hello," in the *hello.awk* file as well as "Hello" or "hello", or even "helloes". You can eliminate these various matchings by using a string instead of a regular expression:

```
$ awk '$1 == "Hello," || "hello," ' hello.awk
```

matches only if *$1* is either the string "Hello," or "hello," and nothing else. This generates the same output as the program

```
$ awk ' $1 ~ /^[Hh]ello,$/ ' hello.awk
```

which specifies that *$1* starts with "H" or "h" and ends with a comma. In this case it's shorter to use the regular expression.

If you use regular expressions in your patterns, you can match many strings. But if you use strings in your patterns, you can match only those exact strings in the input. Both tactics are valuable in different situations.

Combinations of Patterns

You can combine several patterns into one using the Boolean operators || (or), **&&** (and), **==** (equal to), and **!=** (not equal to). For example, the pattern

```
$ awk '$1 >= "H" && $1 < "I" && NF == 2 && $2 != "world!"' hello.awk
```

matches records that begin with "H" and have two fields but do not have "world!" as the second field. The record "Hello, world!" won't match, but the record "Howdy, partner!" (or "Houston, Texas" for that matter) will match.

Awk always evaluates the operands of **&&** and || from left to right. The evaluation stops as soon as the expression is found to be true or false. You can use parentheses freely to force the order of evaluation or to increase legibility.

Pattern Ranges

The pattern you specify in a pattern-action statement can consist of two patterns separated by a comma. When you specify the pattern in this way, the action is executed on each record from an occurrence of the first pattern through the next occurrence of the second pattern. For example,

```
$ awk '/Hello/,/Partner/' hello.awk
```

prints all records from the first one matching "Hello" through the next one matching "partner". The statement

```
$ awk 'NR == 10, NR == 30 {print $0}' filename
```

prints records 10 through 30 of some file (try it on one of your files). If you use the above program on a 20-line file, *awk* will print lines 10 through 20 and stop without generating an error.

Designing Actions

Actions consist of one or more statements. A statement can include:

- Arithmetic expressions
- Assignment statements
- Output statements
- Built-in function calls
- Flow-of-control statements

Variables

In *awk* programs, you do not need to write declaration statements for variables. The variables take on numeric (floating point) or string values automatically, according to context. For example,

`x = 1`	gives *x* a numeric value;
`x = "HP-UX"`	gives *x* a string value;
`x = "3" + "4"`	assigns 7 to *x* as if the equation were $x = 3 + 4$ (because context demands numeric values in this case).

Variables are automatically initialized to the null string (numerical value = 0), so you don't need to initialize variables in a *BEGIN* statement. For example, the sums of the first two fields of all records can be computed by a two-line program:

```
      { s1 = s1 + $1; s2 = s2 + $2 }
END   { print s1, s2 }
```

which you can enter and then try by typing

```
$ awk -f two_line_program file_with_numbers
```

Awk does all of its arithmetic internally and in floating point. The available operators are:

+, −, *, and /	addition, subtraction, multiplication, and division
%	the mod operator (for example, the pattern *NF % 2 = = 0* prints all lines in the input that have an even number of fields)
+ + and − −	the increment and decrement operators (like those in C language)
+ =, − =, *=, /=, and %=	the C language assignment operators (for example, *x + = 1* is the same as *x = x + 1*).

Any of these operators may be used in an expression.

Field Variables

You can treat the field variables (*$1*, *$2*, etc.) just as any other variable. You can replace fields with other numbers, assign results to a field, or use fields in expressions. For example,

```
$ awk '{ $1 = $2 + $3; print $0 }' filename
```

accumulates fields 2 and 3 into field 1 and prints out the record with a new field 1. If you use *hello.awk* for *filename*, *awk* will convert the strings to numbers in response to the context and *$1* will turn out to be zero. *$3* is a null string which equals zero.

Fields can be referred to by numerical expressions, such as *$(i)*, *$(n+1)*, or *$(NF*4 + 3/(NR−5))*. (If the expression comes out non-integer, *awk* truncates the decimal portion and uses the remaining integer portion as the result.) For example, to refer to the last field when you're unsure how many fields are in the record, use *$NF*.

Whether a field variable is considered numeric or string depends on context. The matter is not a concern to most *awk* users. You may run into ambiguous cases such as

```
if ($1 == $2)
```

in which *awk* has no criteria for deciding whether to compare strings or numbers. Just as in the relational expressions discussed earlier, *awk* solves the ambiguity by treating fields as strings in such cases.

Arrays

Awk also defines and initializes arrays automatically. To create an array, simply mention it when you need it; *awk* creates the array for you then and there. The subscripts can have numeric values or string values, such as *x*["Hello,"]. The program

```
/Hello/    {x["Hello"]++}
/world/    {x["world"]++}
END        {print x["Hello"], x["world"]}
```

counts the occurrences of "Hello" and "world" in the input, stores the counts in elements of the array, and prints the final results. Enter this progam in a file, and try it using the **−f** option.

Built-in Functions

You can use a number of built-in functions in your *awk* programs. These include both string and arithmetic operations.

length(*x*) returns the length of the argument. For example,

```
$ awk '{print length($1), $0}' hello.awk
```

prints the length of the first field then prints out the entire record.

sqrt(*x*) returns the square root of the argument.

log(*x*) returns the base *e* logarithm of the argument.

exp(*x*) returns the exponential of the argument.

int(*x*) returns the integer part of the argument.

The arguments of functions can be any expression. For all of the above functions, the name of the function alone, with no argument, will cause the function to be performed on the entire record.

substr(*s,m,n*) returns the substring of *s* that begins at position *m* and is at most *n* characters long. For example,

```
$ awk '/Hello/ {print substr($2,3,5)}' hello.awk
```

will produce

```
rld!
```

which is the substring of *$2* – "world!" – starting with the third letter – "r" – and is no more than five characters long (the length of this substring happens to be four).

split(*s,array,sep*) splits the string *s* into *array*[1],...,*array*[n]. (*s* can be a variable.) The number of elements found is returned as *n*. If you don't provide a field separator in the *sep* argument, the current value of *FS* is used by default.

index(*s1,s2*) returns the position in which the string *s2* occurs in the string *s1*. If *s2* is not a subset of *s1*, **index** returns a 0. For example,

```
$ awk '/world/ {print index($2,"r")}' hello.awk
```

prints out *3*, because "r" is the third character in *$2* ("world!").

sprintf(*f,e1,e2...*) places the values of *e1*, *e2*, and so on into the formatted fields specified by *f*. The argument *f* is the format string, which is like the **printf** format string. For example,

```
$ awk '{x = sprintf("%8s %10s", $1, $2);\
> print x}' hello.awk
```

sets *x* to the string produced by formatting strings *$1* and *$2* and prints the result. For a complete discussion of output formatting, look in "Using the C Library Routines" in Volume 2 of *HP-UX Concepts and Tutorials* or Kernighan and Ritchie's *The C Programming Language*.

The other built-in functions that you have already seen are:

print introduced in the Output section of this article

printf introduced in the Formatting Output section

Flow-of-Control Statements

You can use many of the same flow-of-control statements available in C (see *The C Programming Language*). *Awk* provides **if-else**, **while**, and **for** statements, and statement grouping with braces just as in C.

if(*cond*) *stmt* the condition in parentheses is evaluated; if it's true, the statement following the **if** is executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement.

else *stmt* The optional **else** statement is executed if the **if** condition is false. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. For example,

```
$ awk '{if($1 != "Hello,") print $0;\
> else print "Argh!"}' hello.awk
```

prints lines that do not start with "Hello," and prints a complaint when it encounters a line that does. (Note the use of the backslash to fit this long program onto a single command line.)

while(*cond*) *stmt* The condition in parentheses is evaluated; as long as it is true, the statements in braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. The **while** condition is tested before each pass. Try this example:

```
$ awk '{while(i<=2) {print $(i); i++ };\
> i=0}' hello.awk
```

for(*cond*) *stmt* While a variable changes from an initial to a final value, the statement(s) in the braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. Here is the format of the condition:

for (initialize; final; increment){ ... }

Also, you can use:

for (i in *array*) statement

This construction executes *statement* for each element in the specified array. The elements are not necessarily accessed in order. Changing *i* or accessing any new elements during the statement will introduce side effects.

The conditional expression used in the **if**, **while**, and **for** statements can contain any of the standard relational operators (<, <=, >, >=) as well as the match operators ~ and !~ and the logical operators ||, **&&**, ==, and !=. Parentheses for grouping are allowed (and encouraged).

Here are the other flow-of-control statements:

break exits the current **while** or **for** construct.

continue immediately starts the next iteration of the current loop.

next causes *awk* to skip immediately to the next record and begin scanning the patterns from the beginning of the program.

exit causes the program to behave as if the end of the input had occurred (thus **exit** causes execution of the *END* statement if there is one)

Commenting

Comments in *awk* programs begin with the # character and end with the end of the line:

```
/string/ {print $2, $5} #Print fields 2 and 5 if string matches
```

Error Messages

Diagnostic output for *awk* is sparse and cryptic. Most *awk* errors that stop the program are syntax errors. Typical error statements are:

```
syntax error near line 1
illegal statement near line 1
```

Syntax errors often produce an additional message:

```
bailing out near line 1
```

meaning that the program gave up and returned control to the shell.

"Near" means that the line specified in the error message may not be the line that contains the actual error.

The message

```
funny variable xxxxxxxxx
```

is *awk*'s response to a variable it can't deal with, such as a the negative field variable *$(−1)*.

Some *awk* messages are more specific:

```
newline in character class near line 1
```

states the problem clearly.

Except when redirecting output, if you refer to a file that doesn't exist or can't be opened, you'll get the shell message:

```
awk: can't open file
```

Notes on the Design

Awk improves on *grep*, *egrep*, *fgrep*, *sed*, and *ed* by offering numeric processing, logical relations, and variables. *Awk* programs do not require compilation, as C programs do, and you do not need to know the C programming language to use *awk* (though it sometimes helps). *Awk* is one of the few tools on HP-UX that let you conveniently access fields within a line (*cut* is another such tool).

The designers of *awk* tried to integrate strings and numbers, treating all quantities as both, and postponing a choice until the last minute. This is why you can generally ignore the difference between a string and a number as you write a program.

Most *awk* users extract or manipulate information from the inputs. These usages are sometimes referred to as report generation and data transformation.

Notes on Awk Implementation

Aho, Weinberger, and Kernighan wrote *awk* using tools available on HP-UX, including *yacc* and *lex*. The elements that recognize regular expressions are deterministic finite automata, constructed directly from the expressions. When you invoke *awk*, your program is translated into a parse tree by the parser that was generated by *yacc* and *lex*. A simple interpreter executes the parse tree.

Awk is not fast. Breaking input into fields and delaying the evaluation of variable types are inherent bottlenecks. Further, there is no *awk* compiler, so you cannot use faster compiled versions of oft-used programs. The *awk command* is a machine that translates (parses) and interprets a program written in the *awk language* each time the program is run.

Annotated Examples

Generating Reports

One of the practical applications of *awk* is to put text into a different form or to alter its format for a particular requirement. This example shows how text can be selectively extracted and manipulated with *awk*.

The input file is a list of universities from the Big 8, Big 10, and Pac 10 athletic conferences. (If you wish to test this exaple, you must type in part of all of the file, or one like it.) The file lists the universities' names (one or two fields), the states in which they are located, and the seating capacities of their stadiums. The name of this file is *schools*. You can print the file with:

```
$ awk '{print $0}' schools
Arizona          AZ       Pac 10       52000
Arizona State    AZ       Pac 10       70021
Southern Cal     CA       Pac 10       92516
Stanford         CA       Pac 10       84892
UCLA             CA       Pac 10       92516
Washington       WA       Pac 10       59800
Washington State WA       Pac 10       40000
Oregon           OR       Pac 10       41009
Oregon State     OR       Pac 10       40593
California       CA       Pac 10       76780
Michigan         MI       Big 10      101701
Ohio State       OH       Big 10       85290
Indiana          IN       Big 10       52354
Iowa             IA       Big 10       66000
Illinois         IL       Big 10       70906
Michigan State   MI       Big 10       76000
Minnesota        MN       Big 10       62212
Northwestern     IL       Big 10       49256
Purdue           IN       Big 10       69250
Wisconsin        WI       Big 10       77280
Oklahoma         OK       Big 8        75008
Oklahoma State   OK       Big 8        50817
Missouri         MO       Big 8        62000
Iowa State       IA       Big 8        50000
Colorado         CO       Big 8        51805
Nebraska         NE       Big 8        73650
Kansas State     KS       Big 8        42000
Kansas           KS       Big 8        51500
$
```

Suppose you became interested in how many of these 28 schools (print out *NR* to verify that) were located in a particular state. Because each record contains the two-letter abbreviation of the school's state, the command:

```
$ awk '/CA/ {print $0}' schools
```

results in:

```
Southern Cal     CA       Pac 10       92516
Stanford         CA       Pac 10       84892
UCLA             CA       Pac 10       92516
California       CA       Pac 10       76780
```

Similarly, you can print all the records from a particular conference, or all the schools with "State" in their names:

```
$ awk '/State/ {print $0}' schools
Arizona State     AZ        Pac 10        70021
Washington State  WA        Pac 10        40000
Oregon State      OR        Pac 10        40593
Ohio State        OH        Big 10        85290
Michigan State    MI        Big 10        76000
Oklahoma State    OK        Big 8         50817
Iowa State        IA        Big 8         50000
Kansas State      KS        Big 8         42000
$
```

In the last two examples, pattern matching was done using regular expressions. This could have backfired if the file had included records on, say, the Central YMCA Community College of Chicago (first example) or of a college on Staten Island (second example). When designing patterns you should be aware of such potential pitfalls and think of ways around them.

Printing out all these records (lines) may not be all that you want to do. Suppose you're working for a professional soccer league that's looking for stadiums. You've been assigned to find stadiums with seating capacities greater than 60,000. The *schools* file is a limited resource, but it's a fair place to start. To find out which stadiums are big enough for your needs, use an action containing an **if** statement. Because you want to test every record, a pattern is not necessary. Type:

```
$ awk '{if ($(NF) > 60000) print $0}' schools
```

and get:

```
Arizona State     AZ        Pac 10        70021
Southern Cal      CA        Pac 10        92516
Stanford          CA        Pac 10        84892
UCLA              CA        Pac 10        92516
California        CA        Pac 10        76780
Michigan          MI        Big 10       101701
Ohio State        OH        Big 10        85290
Iowa              IA        Big 10        66000
Illinois          IL        Big 10        70906
Michigan State    MI        Big 10        76000
Minnesota         MN        Big 10        62212
Purdue            IN        Big 10        69250
Wisconsin         WI        Big 10        77280
Oklahoma          OK        Big 8         75008
Missouri          MO        Big 8         62000
Nebraska          NE        Big 8         73650
```

Remember, *NF* is the predefined variable that means the number of fields in the current variable. You could do this job with just a pattern:

```
$ awk ' $NF > 60000 ' schools
```

gets the same result (and it's shorter). If you want to save this information, type

```
$ awk '{if ($NF > 60000) print $0 >>big_stadiums}' schools
```

The >> operator appends the output to *big_stadiums*; you should use this operator as you search other files for similar information. Note the use of *$NF* for the stadium-capacity field. This is used because the number of fields per record varies, but the stadium capacity is always in the last field.

Assume you want to find the average size of a group of stadiums. You need to scan all the records to add up the stadium sizes, then divide the total by the number of records to get the average. The final calculation takes place in an *END* statement after you have processed all of the input. Due to the size of this *awk* program, place it in a separate file:

```
$cat >avg_capacity
{ x += $(NF) }  # accumulate capacities in x
END { x /= NR; print "Average Capacity = ", x}
<control-d>
$
```

The <**control-d**> line represents typing **CONTROL**-[D].

No quotes are needed around the program to protect it from shell interpretation, because you are not typing it on the command line. Also, you do not need to initialize *x*; *awk* sets *x* to the null string when it is created. The arithmetic is automatically done in floating point. To run the program, use the **−f** option:

```
$ awk -f avg_capacity schools
Average capacity = 64898.4
$
```

A longer program finds the average stadium capacity of each conference:

```
/Pac/      {xp += $(NF); ip++ }     #accum caps in xp; incr. count
/Big 10/   {x10 += $(NF); i10++}    #accum caps in x10; incr. count
/Big 8/    {x8 += $(NF); i8++}      #accum caps in x8; incr. count
END        {print "Avg. cap. Pac 10 = ", (xp/ip)
            print "Avg. cap. Big 10 = ", (x10/i10)
            print "Avg. cap. Big 8 = ", (x8/i8)
           }
```

The calculations and print statements for each conference cannot be on the same line with the accumulation statements because *awk* runs all the pattern-action pairs on each input record as it arrives. *END* is executed after all the input comes in. You don't know all the data until the *END* statement.

Notice that no semicolons are used in the multi-line *END* statement. *Awk* treats the newline as another expression separator.

The output of this program is

```
Avg. cap. Pac 10 = 65012.7
Avg. cap. Big 10 = 71024.9
Avg. cap. Big 8  = 57097.5
```

The information in the file *schools* forms a small database. *Awk* is useful for retrieving and manipulating information from such databases. Other applications include updating or reformatting input files.

One last job. Who has the biggest stadium?

```
$cat >findbiggest
{ x = $(NF); if (x>y) {y=x; bigrecord = $0}}
END {print bigrecord}
<control-d>
$

$ awk -f findbiggest schools
Michigan          MI      Big 10       101701
$
```

Doing Calculations

The following program finds the mean and the square root of the sum of the squares (root-mean-square or rms) of a list of input numbers. The work is performed on every record, so there are no patterns in this program other than *END*. This example is taken from "A Walk Through Awk," a paper by Leon S. Levy of Bell Laboratories.

```
{sum_of_squares += $1 * $1}  #accum sum of squares
{sum += $1}                  #accum nos for mean calc
END    {mean = sum / NR      #calc mean
        print "mean = ", mean
        rms = sqrt(sum_of_squares/NR)  #calc rms
        print "rms = ", rms
       }
```

Type this program into a file called *meanrms* and type some numbers into an input file, perhaps called *meanrms.data*.

```
$cat >meanrms.data
20
30
55
40
<control-d>
$ awk -f meanrms meanrms.data
mean = 36.25
rms = 38.487
$
```

The <**control-d**> notation means you should press **CONTROL**-[D] to end the new file.

You don't have to create an input file to use this program. You can run it using the standard input (your keyboard):

```
$ awk -f rms_mean -
```

The dash for the input is optional.

Type in each number you want in the calculation, pressing **RETURN** after each entry. After typing the last entry and pressing **RETURN**, type **CONTROL**-[D] (the end-of-file character). *Awk* then executes the *END* statement.

Because *awk* treats variables as strings until otherwise informed, giving character strings to this program produces unexpected answers. Mixing numbers and characters causes *awk* to calculate the mean and rms using the ASCII values of the characters.

Rearranging Data

You can use *awk* to change the format of records in a file. Assume you have this list of names in a file called *poets*:

```
Poe, Edgar Allan
Longfellow, Henry Wadsworth
Shakespeare, William
Frost, Robert
Dickinson, Emily
```

To transpose the first and last names, type

```
$ awk '{print $2, $1}' poets
```

This successfully switches around the first two fields, but it leaves out the middle names on two of the records and it leaves the commas on all the surnames.

To remove the commas, type

```
$ awk '{print $2, substr( $1, 1, length($1)-1 )}' poets
```

This complex-looking program drops the commas by printing a *substring* of the surname field consisting of everything *but* the comma, using the **substr** function. The first parameter, *$1*, is the object of the **substr** function. The second parameter, 1, means that we want the substring to start at the beginning of *$1*. The third parameter, *length($1) – 1*, means that the substring should end one character before the end of *$1* (just before the comma).

Saving the middle names is more awkward. Try adding *$3* to the **print** action:

```
$ awk '{print $2, $3, substr($1, 1, length($1)-1)}' poets
```

The records with middle names come out correctly. But when no middle name is present an extra space occurs between the first and last names. The extra space is an output field separator being printed after *$3*; when *$3* is null (no middle name), you get two output field separators in a row.

One solution is to use an **if-else** statement to detect a middle name in the record. The program

```
{if (length($3) > 0) print $2, $3, substr($1, 1, length($1)-1)
else print $2, substr($1, 1, length($1)-1) }
```

when run:

```
$ awk -f reverse_names poets
```

results in:

```
Edgar Allan Poe
Henry Wadsworth Longfellow
William Shakespeare
Robert Frost
Emily Dickinson
```

Now that you have the format you want, save it by redirecting it to another file:

```
$ awk -f reverse_names poets >poets.awked
```

Another job you may want to do with a file like *poets* is to rearrange the records in alphabetical order. Unfortunately, *awk* cannot reorder records. Try using the *sort* utility. *Sort* is described in the *HP-UX Reference* and in the McGilton and Morgan book starting on page 138.

References

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "Awk–A Pattern Scanning and Processing Language", Bell Laboratories, September 1978. Second Edition. (Not available from HP.)
2. Henry McGilton and Rachel Morgan, *Introducing the UNIX™ System*, McGraw Hill, 1983, pp. 177-184. (UNIX is a trademark of AT&T Bell Laboratories.) HP Part # 98680-90025.
3. *HP-UX Reference for the HP 9000 Series 200/500.* HP Part # 09000-90007.
4. Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall, 1978. HP part # 97089-90000.
5. "Using the C Library Routines," *HP-UX Concepts and Tutorials*, Volume 2 (Program Maintenance and Development).

Table of Contents

Nroff/Troff Text Processors

Introduction

TROFF is **not** *supported on Series 9000 workstations at this time.*

nroff and **troff** are text processors under the HP-UX Operating System. They accept lines of text, interspersed with lines of format control information and format the text into a printable, paginated documentation having a user-designed style. They offer unusual freedom in document styling, including:

- arbitrary style headers and footers.
- arbitrary style footnotes.
- multiple automatic sequence numbering for paragraphs.
- a family of automatic overstriking, bracket construction and line drawing functions.

These text processors 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. They can prepare output directly for a variety of HP terminals and are capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking **nroff** at HP-UX command level is:

```
nroff [options] file_name_list
```

where options represents any number of option arguments and `file_name_list` 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 `file_name_list` are:

option	Effect
-olist	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N-M* and means pages *N* through M inclusive. An initial *-N* means from the beginning (page 1) to page *N* and a final *N-* means use page numbers from *N*.
-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 new-line character.
-nname	Prepends the macro file */usr/lib/tmac.name* to the input `file_name_list`.
-cname	Same as -nname, but uses a compacted form of /usr/lib/tmac.name for the sake of efficiency.
-raN	Register a (register name limited to one character) is set to the value *N*.
-i	Read standard input after the *file_name_list* 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.
-e	Produce equally spaced words in adjusted lines, using full terminal resolution.

Each option is invoked as a separate argument, for example:

```
nroff -o4,8-10  -mabc file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files `file1` and `file2` and invoking the macropackage `abc`.

A reverse-line postprocessor `col(1)` is available for multiple-column output on terminals without reverse-line ability.

Summary and Index

General Explanation

The following is a list of supplied formatting constructs. These "dot constructs" must begin in the far left margin of a line and must be the only entry on that line. The **Notes#** refered to are:

Note	Meaning
1	No effect in **nroff**.
2	Values separated by ";" are for **nroff** and **troff** respectively.
3	The use of "'" as control character (instead of ".") suppresses the break function.
4	Request normally causes a break.
5	Mode or relevant parameters associated with current diversion level.
6	Relevant parameters are a part of the current environment.
7	Must stay in effect until logical output.
8	Mode must be still or again in effect at the time of physical output.
9	Default scale indicator; it not specified, scale indicators are ignored.

Font Control

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ft F	Roman	previous	5	Change to font F = x. Also \fx.

Page Control

Request Form	Initial Value	If No Argument	Notes#	Explanation
.pl ±N	11 in	11 in	9	Page length.
.bp ±N	N = 1	-	3,9	Eject current page; next page number N.
.pn ±N	N = 1	ignored	-	Next page number N.
.po ±N	0; 26/27 in	previous	9	Page offset
.ne N	-	N = 1 V	5,9	Need N vertical space (V = vertical spacing).
.mk R	none	internal	5	Mark current vertical place in register R.
.rt ±N	none	internal	5,9	Return (upward only) to marked vetical place.

Text Filling, Adnusting and Centering

Request Form	Initial Value	If No Argument	Notes#	Explanation
.br	-	-	4	Break.
.fi	fill	-	4,6	Fill output lines.
.nf	fill	-	4,6	No filling or adjusting of output lines.
.ad c	adj,both	adjust	6	Adjust output lines with mode c.
.na	adjust	-	6	No output line adjusting.
.ce N	off	N = 1	4,6	Center following N input text lines.

Vertical Spacing

Request Form	Initial Value	If No Argument	Notes#	Explanation
.vs N	1/6in	previous	6,9	Vertical base line spacing (V).
.ls N	N = 1	previous	6	Output N-1 Vs after each text output line.
.sp N	-	N = 1V	6,9	Save vertical distance N in either direction.
.sv N	-	N = 1V	9	Save vertical distance N.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	5	Turn no-space mode on.
.rs	-	-	5	Restore spacing; turn no-space mode off.

Line Length and Indenting

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ll ±N	6.5 in	previous	6,9	Line length.
.in ±N	N = 0	previous	4,6,9	Indent.
.ti ±N	-	ignored	4,6,9	Temporary indent.

Macros, Strings, Diversion and Position Traps

Request Form	Initial Value	If No Argument	Notes#	Explanation
.de xx yy	-	.yy=..	-	Define or redefine macro xx; end at call of yy.
.am xx yy	-	.yy=..	-	Append to macro.
.ds xx string	-	ignored	-	Define a string xx containing string.
.as xx string	-	ignored	-	Append string to string xx.
.rm xx	-	ignored	-	Remove request, macro or string.
.rn xx yy	-	ignored	-	Rename request, macro or string xx to yy.
.di xx	-	end	5	Divert output to macro xx.
.da xx	-	end	5	Divert and append to xx.
.wh N xx	-	-	9	Set location trap; negative is w.r.t. page bottom.
.ch xx N	-	-	9	Change trap location.
.dt N xx	-	off	5,9	Set a diversion trap.
.it N xx	-	off	6	Set an input-line count trap.
.em xx	none	none	-	End macro is xx.

Number Registers

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nr R ±N M		-	9	Define and set number register R; auto-increment by M.
.ar R c	arabic	-	-	Assign format to register R (c=1,i,I,a,A).
.rr R	-	-	-	Remove register R.

Tabs, Leaders and Fields

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ta Nt ...	0.8; 0.6in	none	6,9	Tab settings; left type, unless t=R(right), C(centered).
.tc c	none	none	6	Tab repetition character.
.lc c	.	none	6	Leader repetition character.
.fc a b	off	off	-	Set field delimiter a and pad character b.

I/O Conventions and Character Translations

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ec c	\	\	-	Set excape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg N	-;on	on	-	Ligature mode on if N>0.
.ul N	off	N = 1	6	Underline (italicize in **troff**) N input lines.
.cu N	off	N = 1	6	Continuous underline in **nroff**; like ul in **troff**.
.uf F	Italic	Italic	-	Underline font set to F (to be switched to by ul).
.cc c	.	.	6	Set control character to c.
.c2 c	'	'	6	Set nobreak control character to c.
.tr abcd...	none	-	7	Translate a to b, etc. on output.

Hyphenation

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nh	hyphenate	-	6	No hyphenation
.hy N	hyphenate	hyphenate	6	Hyphenate; N = mode.
.hc c	\%	\%	6	Hyphenation indicator character c.
.hw word1...		ignored	-	Exception words.

Three Part Titles

Request Form	Initial Value	If No Argument	Notes#	Explanation
.tl 'left'center'right'		-	-	Three part title.
.pc c	%	off	-	Page number character.
.lt ±N	6.5in	previous	6,9	Length of title.

Output Line Numbering

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nm ±N M S I		off	6	Number mode on or off, set parameters.
.nn N	-	previous	6	Do not number next N lines.

Conditional Acceptance of Input

Request Form	Initial Value	If No Argument	Notes#	Explanation
.if c anything		-	-	If condition c true, accept anything as input, for multi-line ust\{anything\}.
.if !2c anything		-	-	If condition c false, accept anything as input.
.if N anything		-	9	If expression N>0, accept anything.
.if !N anything		-	9	If expression N≤0, accept
.if 'string1'string2' anything			-	If string1 identical to string2, accept anything.
.if !'string1'string2' anything			-	If string1 not identical to string2, accept anything.
.ie c anything		-	9	If portion of if-else; all above corms (like if).
.el anything		-	-	Else portion of if-else.

Environmental Switching

Request Form	Initial Value	If No Argument	Notes#	Explanation
E.ev N	N = 0	previous	-	Environment switched (push down).

Insertions From Standard Input

Request Form	Initial Value	If No Argument	Notes#	Explanation
.rd prompt	-	prompt = BEL	-	Read insertion.
.ex	-	-	-	Exit from **nroff**.

Input/Output File Switching

Request Form	Initial Value	If No Argument	Notes#	Explanation
.so filename		-	-	Switch source file (push down).
.nx filename		end-of-file	-	Next file.
.pi program		-	-	Pipe output to program (**nroff** only).

Miscellaneous

Request Form	Initial Value	If No Argument	Notes#	Explanation
.mc c N	-	off	6,9	Set margin character c and separation N.
.tm string	-	newline	-	Print string on terminal, (HP-UX standard message output)
.ig yy	-	.yy=..	-	Ignore till call of yy.
.pm t	-	all	-	Print macro names and sizes; if t present, prent only total of sizes.
.fl	-	-	4	Flush output buffer.

Escape Sequences

Escape Sequence	Meaning
\\	\ (to prevent or delay the interpretation of \).
\e	Printable version of the current escape character.
\'	(acute accent); equivalent to \(a.a
\`	(grave accent); equivalent to \g.a
\-	Minus sign in the current font.
\.	Period (dot) (see de).
\(space)	Unpaddable space-size space character.
\0	Digit width space.
\	Zero width in **nroff**.
\^	zero width in **nroff**.
\&	Non-printing, zero width charactr.r
\!	Transparent line indicator.
\"	Beginning of comment.
\$N	Interpolate argument $1 \leq N \leq 9$.
\%	Default optional hyphenation character.
\(xx	Character named xx.
*x, *(xx	Interpolate string x or xx.
\a]	Non-interpreted leader character.
\b 'abc...'	Bracket building function.
\c	Interrupt text processing.
\fx	Change to font named x.
\h'N'	Local horizontal motion; move right N (negative left)

\kx	Mark horizontal input place inregister x
\l'Nc'	Horizontal line drawing function (optionally with c)
\L'Nc'	Vertical line drawing function (optionally with c)
\nx, \n(xx	Interpolate number register x or xx
\o'abc...'	Overstrike characters a, b, c, ...
\p	Break and spread output line
\r	Reverse line in **nroff**.
\t	Non-interpreted horizontal tab.
\u	1/2 line in **nroff**.
\v'N'	Local vertical motion; move down N (negative up).
\w'string'	Interpolate width of string.
\x'N'	Extra line-space function (negative before, positive afte).)
\zc	Print c with zero width (without spacing).
\{	Begin conditional input.
\}	End conditional input.
\(new-line)	Concealed (ignored) new-line.
\X	X, any character not listed aboe.e

The escape sequences \\, \., \'', \$, *, \a, \n, \t,and \(newline) are in copy mode.

Predefined General Number Registers

Register Name	Description
%	Current page number.
ct	Character type (set by width function).
dl	Width (maximum) of last completed diversion.
dn	Height (verticalsize) of last completed diversion.
dw	Current day of the week (1-7).
dy	Current day of the month (1-31).
hp	Current horizontal place on input line.
ln	Output line number.
mo	Current month (1-12).
nl	Vertical position of last printed text base-line.
sb	Depth of string below base in e (generated by width function).
st	Height of string above base line (generated by width function).
yr	Last two digits of curret tyear.

Predefined Read-Only Number Registers

Register Name	Description
.$	Number of arguments available at the current macro level.
.A	Set to 1 in , if -a option used; always 1 in .
.H	Available horizontal resolution in basic units.
.T	Set to 1 in , if -T option used; always 0 in .
.V	Available vertical resolution in basic units.
.a	Post-line extra line-space most recently utilized using \x'N'.
.c	Number of lines read from current input file.
.d	Current vertical place in current diversion; equal to nl, if no diversion.
.h	Text base-line high-water mark on current page or diversion.
.i	Current indent.
.l	Current line length.
.n	Length of text portion on previous output line.
.o	Current page offset.
.p	Current page length.
.s	Current point size.
.t	Distance to the next trap.
.u	Equal to 1 in fill mod and 0 in nofill mode.
.v	Current vertical line spacing.
.w	Width of previous character.
.x	Reserved version-dependent register.
.y	Reserved version-dependent register.
.z	Name of current diversion.

Nroff/Troff Reference

The Basics

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 a period (.) or single-quote ('), 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 single-quote control character suppresses the break function. The break function forces output of a partially filled line and is caused by certain requests. The control character may be separated from these (spaces and/or tabs) for esthetic reasons. Names must be followed by either a space or new-line character. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an escape character, normally a backslash (\). 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.

Formatter Resolution

nroff internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices.

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	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	EM = *S* points	6×*S*	*C*
n	EN = Em/2	6×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

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**; Vs 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, a vertical bar (|) 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 to the 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.

Numerical Expressions

Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, -, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) maybe 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.

Notation

Numerical parameters are indicated in this manual in two ways. ±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.

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). 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 mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The usable page width on a 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 output are output-device dependent.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.pl ±N	11in	11in	9	Page length set to ±N. The internal limitation is about 75 inches in and about 136 inches in. The current page length is available in the .p register.
.bp ±N	N=1	-	3,4,9	Begin page. The current page is ejected and a new page is begun. If ±N is given, the new page number will be ±N. Also see request ns.
.pn ±N	N=1	ignored	-	Page number. The next page (when it occurs) will have the page number ±N. A pn must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the % register.
.po ±N	0; 26/27in	previous	2,9	Page offset. Values separated by ";" are forand respectively. The current left margin is set to ±N. The initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In the maximum (line-length) + (page-offset) is about 7.54 inches. The current page offset is available in the .o register.
.ne N	-	N=1V	5,9	Need N vertical space. If the distance, D, to the next trap position is less than N, a forward vertical space of size D occurs, which will spring the trap. If there are no remaining traps on the page, D is the 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	5	Mark the current vertical place in an internal register (both associated with the current diversion level), or in register R, if given. See rt request.
.rt ±_N	none	internal	5,9	Return upward only to a marked vertical place in thecurrentdiversion. If _N (w.r.t. currentplace) is given, the place is _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 absoluteplace stored ina explicit register; e. g. using the sequence .mkR ... .sp nRu.

Text 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. 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 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 textbase-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 characterinto the control character using tr.

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	-	-	4	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	-	4,6	Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	4,6	Nofill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length.
.ad c	adj,both	adjust	6	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator c 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	-	6	No adjust. 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	4,6	Center the next N input text lines within the current (line-length minus indent). If N=0, any residual count is cleared. A break occurs after each of the N input lines. If the input line is too long, it will be left adjusted.

Vertical Spacing

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; default is 10-point type on a 12-point spacing. The current V is available in the .v register. Multiple-V line separation (e.g. doublespacing) may be requested with ls.

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 \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'), 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.

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 maybe reserved using sv.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.vsN	1/6in;12pts	previous	6,9	Set vertical base-line spacing size V. Transient extra vertical space available with \x'N'.
.lsN	N=1	previous	6	Line spacing set to ±N. N-1 Vs (blank lines) 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 N	-	N=1V	4,9	Space vertically in either direction. If N is negative, the motion is backward (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 N	-	N=1V	9	Save a contiguous vertical block of size N. If the distance to the next trap is greater than N, N vertical space is output. No-space mode has no effect. If this distance is less than N, no vertical space is immediately output, but N is remembered for later output. Subsequent sv requests will overwrite any still remembered N.
.os	-	-	-	Output saved vertical space. No-space mode has no effect. Used to finally output a block of vertical space requested by an earlier sv request.
.ns	space	-	4	No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests without a next page number. The no-space mode is turned off when a line of output occurs, or with rs.
.rs	space	-	4	Restore spacing. The no-space mode is turned off.
Blank text line		-	4	Causes a break and output of a blank line exactly like sp1.

Line Length and Indenting

The maximum line length for fill mode may be set with ll. The indent maybe 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 linelength and indent are available in registers .l and .i respectively. The length of three-part titles produced by tl is independently set by lt.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ll ±N	6.5 in	previous	6,9	Line length is set to ±N.
.in ±N	N=0	previous	4,6,9	Indent is set to ±N. The indent is prepended to each output line.
.ti ±N	-	ignored	4,6,9	Temporary indent. The next output text line will be indented a distance ±N with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

Macros,Strings, Diversion, and Position Traps

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 new-line 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 string sx and xx are interpolated at any desired point with *x and *(xx respectively. String references and macro invocations may be nested.

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 new-lines indicated by \(new-line) are eliminated.
- Comments indicated by \" are eliminated.
- \t and \a are interpreted as ASCII horizontal tab and SOH respectively.
- \\ is interpreted as \.
- \. is interpreted as ".".

These interpretations can be suppressed by prepending a backslash (\). 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.

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 new-line 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 Nth argument ($1 \leq N \leq 9$). If an invoked argument doesn't exist, a null string results. For example, the macro xx maybe defined by

```
xx \"begin definition
Today is \\$1the \\$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 backslash (\). 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 a input-level pushdown, 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 backslash (\)) to delay interpolation until argument reference time.

Diversions

Processed output may be diverted into a macro for purposes such as footnote processing 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.

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).

Traps

Three types of trap mechanisms are available:

- page traps
- diversion traps
- and an input-line-count traps

Macro-invocation traps may be planted using what 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. 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 xx yy	-	.yy=..	-	Define or redefine the macro xx. The contents of the macro begin on the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with .yy, whereupon the macro yy is called. In the absence of yy, the definition is terminated by a line beginning with "..". A macro may contain de requests provided the terminating macros differ or the contained definition terminator is concealed; ".." can be concealed as "\\.." which will copy as "\.." and be reread as "..".
.am xx yy	-	.yy=..	-	Append to macro (append version of de).
.ds xx string	-	ignored	-	Define a string xx containing string. Any initial double-quote in string is stripped off to permit initial blanks.
.as xx string	-	ignored	-	Append string to string xx (append version of ds).

.rm xx	-	ignored	-	Remove request, macro, or string. The name xx is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn xx yy	-	ignored	-	Rename request, macro, or string xx to yy. If yy exists, it is first removed.
.di xx	-	end	5	Divert output to macro xx. 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.
.da xx	-	end	5	Divert, appending to xx (append version of di).
.wh N xx	-	-	9	Install a trap to invoke xx at page position N; a negative N will be interpreted with respect to the page bottom. Any macro previously planted at N is replaced by xx. A zero N refers to the top of a page. In the absence of xx, the first found trap at N, if any, is removed.
.ch xx N	-	-	9	Change the trap position for macro xx to be N. In the absence of N, the trap, if any, is removed.
.dt N xx	-	off	5,9	Install a diversion trap at position N in the current diversion to invoke macro xx. Another dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it N xx	-	off	6	Set an input-line-count trap to invoke the macro xx after N lines of text input have been read (control or request lines don't count). The text may be in-line text or text interpolated by in-line or trap-invoked macros.
.em xx	none	none	-	The macro xx will be invoked when all input has ended. The effect is the same as if the contents of xx had been at the end of the last file processed.

Number Registers

A variety of parameters are available to the user as predefined, named number registers. 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.

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
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	*N*+*M*
\-*x*	*x* incremented by *M*	*N*−*M*
\n+(*xx*	*xx* incremented by *M*	*N*−*M*
\n−(*xx*	*xx* decremented by *M*	*N*−*M*

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**.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nr R ±N	M	-	9	The number register R is assigned the value ±N with respect to the previous value, if any. The increment for auto-incrementing is set to M.
.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. The read-only registers and the width function 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.

Tabs, Leaders, and Fields

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 *next-string*
Left	*D*	Following *D*
Right	*D*−*W*	Right adjusted within *D*
Centered	*D*−*W/2*	Centered on right end of *D*

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.

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 the sharp sign (#), and the padding indicator is a carot (^), *#^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 Nt ...	8n; 0.5in	none	6,9	Set tab stops and types. t=R, right adjusting; t=C,centering; t absent, left adjusting. Tab stops are preset in **nroff** every 8 nominal character widths. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
.tc c	none	none	6	The tab repetition character becomes c, or is removed specifying motion.
.lc c	.	none	6	The leader repetition character becomes c, or is removed specifying motion.
.fc a b	off	off	-	The field delimiter is set to a; the padding indicator is set to the space character or to b, if given. In the absence of arguments the field mechanism is turned off.

I/O Conventions and Character Translations

Input Character Translations

The ASCII control characters horizontal tab, SOH, and backspace are discussed elsewhere. The new-line 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. All others are ignored.

The escape character, backslash (\) introduces:

- escape sequences
- causes the following character to mean another character
- or to indicate some function

The backslash (\) 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 beused 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 c	\	\	-	Set escape character to \, or to c,if given.
.eo	on	-	-	Turn escape mechanism off.

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.

nroff automatically underlines characters in the underline font, specifiable with uf, normally that on font position 2 (normally Times Italic). 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 N	off	N=1	E	Underline in **nroff** the next N input text lines. Actually, switch to underline font, saving the current font for later restoration; other font changes within the span of a ul will take effect, but the restoration will undo the last change. Output generated by tl is affected by the font change, but does not decrement N. If N>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu N	off	N=1	6	A variant of ul that causes every character to be underlined in **nroff**.
.uf F	Italic	Italic	-	Underline font set to F. In **nroff**, F may not be on position 1 (initially Times Roman).

Control Characters

Both the control character period (.) and the no- break control character single-quote (') 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 c	.	.	6	The basic control character is set to c, or reset to ".".
.c2 c	'	'	6	The nobreak control character is set to c, or reset to "'".

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 abcd....	none	-	7	Translate a into b, c into d, 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 input to output time.

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-processoror to imbed control lines in a macro created by a diversion.

Comments and Concealed New-lines

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 backslash (\). The sequence\(new-line) is always ignored-except in a comment. Comments may be imbedded at the end of any line by prefacing them with \". The new-line 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 .\".

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 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.

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 **\kxword\h'nxu+2u'word** will embolden word by backing up to almost its beginning and overprinting it, resulting in **word**.

More Functions

Overstriking

Automatically centered overstriking of up to nine characters is provided by the overstrike function **\o*'string'***. The characters in string are overprinted with centers aligned; the total width is that of the widest character.

Zero-width Characters

The function **\z*c*** will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations.

Line Drawing

The function **\l'N*c*'** 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'0\(ul' ..
```

or one to draw a box around a string

```
.de bx \(br\\$1\(br\l'0\(rn'\l'0\(ul' ..
```

such that .us "underlined words" and .bx "words in a box"

The function **\L'N*c*'** will draw a vertical line consisting of the (optional) character **c** stacked vertically 1 line apart, with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* (\(br); the other suitable character is the *bold vertical* (\(bv). The line is begun without an 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 compenstaing motions are made; the instataneous 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 ½-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 possible 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.

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) or hyphenation indicator characters - such as mother-in-law - are always subject to splitting after those characters, whether or not automatic hyphnenation is on or off.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nh	hyphenate	-	6	Automatic hyphenation is turned off.
.hyN	on,N = 1	on,N = 1	6	Automatic hyphenation is turned on for N≥1, or off for N = 0. If N = 2, last 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 value are additive, i.e. N = 14 will invoke all three restrictions.
.hc c	\%	\%	6	Hyphenation indicator character is set to c or to the defalult \%. The indicator does not appear in the output.
.he word1...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Version 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.

Output Line Numbering

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a 3 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 6 the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and line generated by **tl** are not numbered. Numbering can be temporarily suspended with **nn**, 9 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 files).

Request Form	Initial Value	If No Argument	Notes#	Explanation
.nm ±N M S I		off	6	Line number mode. If ±N is given, line numbering is turned on, and the next output line numbered is numbered ±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 futher use in number register **ln**.
.nn N	-	N = 1	6	The next N text output lines are not numbered.

As an example, the paragraph protions 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'000'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 number line, with M=5, with spacing S untouched, and with the indent I set to 3.

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
.if c anything		-	-	If condition c true, accept anything as input; in multi-line case use \{anything\}
.if !c anything		-	-	If condition c false, accept anything
.if N anything		-	9	If expression N>0, accept anything
.if !N anything		-	9	If expression N≤0, accept anything
.if 'string1'string2' anything			-	If string1 identical to string2, accept anything
.if !'string1'string2' anything			-	If string1 not identical to string2, accept anything
.ie c anything		-	-	If portion of if-else; all above forms (like if)
.il anything		-	-	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.

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 6 in their Notes column; in addition, partially collected lines and words are in the environbment. 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.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.ev N	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 must be done with .ev rather than specific reference.

Insertions From the Standard Input

The input can be temporarily switched to the system's 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 HP-UX, the standard input can be the user's keyboard, a pipe, or a file.

Request Form	Initial Value	If No Argument	Notes#	Explanation
.rd prompt	-	prompt = BEL	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, prompt (or a BEL) is written onto the user's terminal. rd behaves line a macro, and arguments may be placed after prompt.
.ed	-	-	-	Exit from **nroff**. 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**; the process would ultimately be ended by an **ex** in the insertion file.

Input/Output File Switching

Request Form	Initial Value	If No Argument	Notes#	Explanation
.so filename		-	-	Switch source file. The top input (file reading) level is switched to filename. The effect of an so 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. so's may be nested.
.nx filename		end-of-file	-	Next file is filename. The current file is considered ended, and the input is immediately switched to filename.
.pi program		-	-	Pipe output to program (**nroff** only). This request must occur before any printing occurs. No arguments are transmitted to program.

Miscellaneous

Request Form	Initial Value	If No Argument	Notes#	Explanation
.mc c N	p	off	3,9	Specifies that a margin character c appear a distance N to the right of the right margin after each non-empty text line (except those produced by tl). If the output line is too-long (as can happen in nofil mode) the character will be appended to the line. In N is not given, the previous N is used; the initial N is 0.2 inches in **nroff**. The margin character used with this paragraph was a 12-point boxrule.
.tm string	-	newline	-	After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user's terminal.
.ig yy	-	.yy=..	-	Ignore input lines. ig behaves exactly like de except that the input is discarded. The input is read in copy mode, and any auto-incremented registers will be affected.
.pm t	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if t is given, only the total of the sizes is printed. The sizes is given in blocks of 128 characters.
.fl	-	-	4	Flush output buffer. Used in interactive debugging to force output.

Output and Error Messages

The output from **tm**, **pm** and the prompt **rd**, as well as various error messages are written onto HP-UX's standard message output. The latter is different from the standard output, where **nroff** formatted output goes. Be 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**. 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 an asterisk (*) in **nroff**. 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

Introduction

Although **nroff** has, 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 such as page margins and footnotes are deliberately not built into **nroff**. 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.

Page Margins

Header and footer macros are usually defined to describe the top and bottom 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 o hd
.wh -li 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. 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 single-quote (') 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 \{\
'sp0.5i-1               \"tl base at 0.5i
.tl "-%-"               \"centered page number
.ps                     \"restore size
.ft                     \"restore font
.vs \}                  \"restore vs
'sp1.0i                 \"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, *fon*, 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. 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 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 sl\\n(.s            \"current size
.ps
.nr se \\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
```

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 requstss a temporary dent.nt

```
.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 one 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 outpt tdevice vertal alresolution, 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 kS 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**.

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
..0
```

The intended usage is ".lp label"; label will begin at 0.2 inch, and cannot exceed a length of 0.3inch 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.

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 to                  \"footer
.ie\\n+(cl<2\{\
.po +3.4i               \"next column:3.1+0.3
.rt                     \"back to mark
.el \{\
.po\\nMu                \"restore left margin
. ---
.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.

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
..
.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
.it"\\n(.z"fy" .di \"end overflow diversion
.nr x 0 \"disable fx
.ev \}                       \"po 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+\\nv)\
.ch fo\\n(nlu+1v            \"it didn't fit
..
.de fs                      \"separator
\\1'1i'                     \"1 inch rule
.br
..
.de fz                      \"get leftover footnote
.fn
.nf                         \"retain vertical size
.fy                         \"where fx put it
.ef
..
.nr b 1.0i                  \"bottom margin size
.wh o hd                    \"header trap
.wh 12i fo                  \"footer trap, temp position
.wh -\\nbu fx               \"fx at footer position
.ch fo -\\nbu               \"conceal fs 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 redivert 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.

The Last Page

After the last input file has ended, and invoke the end macro, 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 lin, ,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 of Contents

MM: Memorandum Macros

MM
Memorandum Macros

Introduction

This tutorial 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 HP-UX text formatters ***nroff*** and ***troff*** (see *HP-UX Reference* for more details). References of the form *name*(<n>) point to page *name* of section <n> of the *HP-UX Reference.*

Conventions

Each section of this tutorial explains a single facility of MM. In general, the earlier a topic is discussed, the more necessary the information is for most users. Some of the later sections can be completely ignored if MM defaults are acceptable. Likewise, each section progresses from general case to special-case facilities. It is recommended that you read a section in detail only to point where there is enough information to obtain the desired format, then skim the rest of the section because some details may be of limited use.

In the synopses of macro calls, square brackets ([ ]) surrounding an argument indicate that it is optional. Ellipses (. . .) show that the preceding argument can appear more than once.

Since there are large differences in certain aspects of *nroff* and *troff* operation, in cases where the *nroff* formatter output differs from *troff*, *nroff* is explained first, followed by *troff* 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.

Document Structure

Input for a document to be formatted with the MM text-formatting macro package has four major segments, any of which can be omitted. When present, the segments **must** appear in the following order:

1. **Parameter setting segment** sets the general style and appearance of the document. You can control page width, margin justification, numbering styles for heading and lists, page headers and footers, and many other properties of the document. You can also add macros or redefine existing ones. This segment can be omitted entirely if you are satisfied with default values. It produces no actual output, but performs only the formatter setup for the rest of the document.
2. **Beginning Segment** includes those items that occur only once, at the beginning of a document; e.g., title, author's name, and date.

3. **Body segment** is the actual text of the document. It may be as small as a single paragraph or consist of hundreds of pages. It may have a hierarchy of headings up to seven levels deep (see Paragraphs and Headings section). 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, alphabetical sequencing, and "marking" of list items (see Lists section). The body can contain various types of displays, tables, figures, references, and footnotes (described in later sections).
4. **Ending segment** contains those items that occur only once at the end of a document such as signatures and lists of notations (such as "Copy to:" lists). Certain macros can be invoked here to print information that is wholly or partially derived from the rest of the document, such as the cover sheet or table of contents.

Existence and size of these four segments varies widely, depending on document type and individual needs. 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.

Input Text Structure

In order to make it easy to edit or revise input file text at a later time,

- Keep input lines short,
- Break lines at the end of clauses,
- Begin each new sentence on a new line.

Definitions

Formatter refers to either the *nroff* or *troff* text-formatting program.

Requests are built-in commands recognized by the formatters. Although you should seldom need to use these requests directly, you will encounter occasional references to some of the requests. For example, the request

```
.SP
```

inserts a blank line in the output at the where the request appears 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 you can define still others if you need them.

Table 1 at the end of this tutorial 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 where the macro is described. The second line illustrates a typical call to 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. The string's value can be obtained by referencing its name, preceded by "`\*`" (for 1-character string names) or "`\*(`" (for 2-character string names). For example, the string *DT* in MM normally contains the current date, thus the input line

```
Today is \*(DT.
```

could produce the following output:

```
Today is July 4, 1984
```

The current date can be replaced by a command such as

```
.ds DT 01/01/85
```

by invoking a macro designed for that purpose (see memorandum and Released-paper Style Documents section for date changes). Table 2 at the end of this tutorial lists the string names used by MM. A brief description, documentation location reference, and initial (default) values are listed for each.

Number registers fill the role of integer variables. These registers are used as flags and for arithmetic and automatic numbering. A register can be set to a specific value by using a *.nr* request and be referenced by preceding its name with `\n` (for 1-character names) or `\n(` (for 2-character names). For example, the following line sets the value of register *d* to one more than the value of register *dd*:

```
.nr d 1+\n(dd
```

Table 3 is an alphabetical list of number register names including a brief description, documentation location reference, initial (default) value, and acceptable range of values ([m:n] means values from m through n) for each.

Naming conventions are explained in the section on Extending and Modifying MM Macros near the end of this tutorial.

Usage

This section describes how to access MM, illustrates HP-UX command lines appropriate for various output devices, and describes command line flags for the MM text formatting macro package.

The *mm* Command

The *mm*(1) command can be used to prepare documents through the *nroff* formatter and MM (*nroff* is invoked with the *−cm* flag active). 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 (such as −rC3) are passed to the *nroff* formatter or to MM as appropriate. The following options can occur in any order, but must appear before the file names:

Option	Meaning
`-e`	Invokes *neqn*. Also causes *neqn* to read */usr/pub/eqnchar* [(see *eqnchar*(7)].
`-t`	Invokes *tbl(1)*.
`-c`	Invokes *col(1)*.
`-E`	Invokes *−e* option of tthe *nroff* formatter.
`-y`	Uses −mm (uncompacted macros) instead of −cm.
`-12`	Use 12-pitch instead. Set pitch switch on terminal to 12, if necessary.
`-T450`	Output to DASI 450 (default terminal type unless $TERM is set [see *sh*(1)]). Equivalent to `-T1620`.
`-T450-12`	Output to DASI 450 in 12-pitch mode.
`-T300`	Output to DASI 300 terminal.
`-T300-12`	Output to DASI 300 in 12-pitch mode.
`-T300s`	Output to DASI 300S terminal.
`-T300s-12`	Output to DASI 300S in 12-pitch mode.
`-T4014`	Output to Tektronix 4014 terminal.
`-T37`	Output to TELETYPE Model 37.
`-T382`	Output to a DTC-382 terminal.
`-T4000a`	Output to a Trendata 4000A.
`-TX`	Format output for an EBCDIC line printer.
`-ThP`	Output to HP 264x terminal (implies `-c`).
`-T43`	Output to TELETYPE Model 43 (implies `-c`).
`-T40/4`	Output to TELETYPE Model 40/4 (implies `-c`).
`-T745`	Output to Texas Instruments 700 Series terminal (implies `-c`).
`-T2631`	Format output for HP 2631 printer where `-T2631-e` and `-T2631-c` can be used for expanded and compressed modes, respectively (implies `-c`).
`-TlP`	Output to a device with no reverse or partial line motions or other special features (implies `-c`).

Any `-T` option given that is not listed here does not produce an error. It is equivalent to `-TlP`.

Note

While HP-UX allows use of the options listed here, some of them refer to devices that are not included in the list of HP-UX supported peripherals. While MM may interact correctly with some or all of these devices part or all of the time, correct operation is not guaranteed. When using output devices that are not included in the current list of supported peripherals for the version of HP-UX installed on your system, proceed at your own risk.

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 precompacted version of the macros to be loaded. The −mm flag causes 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 so it is ready to process the input text files.

Typical Command Lines

The prototype command lines are as follows (parameters are explained in the next topic):

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] -cm
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 terminal or printer type because the output may require some features that are specific to a given terminal (such as paper motion control). Some HP-UX supported terminal types and appropriate command lines for them are listed here. More information can be found in the next topic and in the *HP-UX Reference*.

Here are some examples:

- For DASI 450, 10 pitch, 6 lines/inch, 0.75-inch offset, 6-inch line length (60 characters) since this is the default terminal type, no −T option is needed (unless $TERM is set to another value). Use:

```
mm file ...
or
nroff -T450 -h -cm file ...
```

- For DASI 450, 12 pitch, 6 lines/inch, 0.75-inch offset, 6-inch line length (72 characters), use:

```
mm -12 file ...
or
nroff -T450-12 -h -cm file ...
```

- For HP 264x CRT terminal family, use:

```
mm -THP file ...
or
nroff -cm file ... | col | hp
```

- For any output device incapable of reverse paper motion and lacking hardware tab stops:

```
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 is used with the *nroff* formatter, either the −c option must be specified to *mm*(1) [*mm*(1) uses *col*(1) automatically for some terminal 1 types], or the *nroff* output must be postprocessed by *col*(1). In the latter case, the −T37 terminal type must be specified for *nroff*, the −h option must **not** be specified, and the output of *col*(1) must be processed by the appropriate temrinal filter. *mm*(1) with the `-c` option handles all this automatically.

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 by using *.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 definitions are:

Register	Definition
-rA<n>	<n> = 1 same as invoking the .AF macro without an argument
-rC<n>	Sets type of copy (such as DRAFT) to be printed at bottom of each page. <n> = 1: OFFICIAL FILE COPY. <n> = 2: DATE FILE COPY. <n> = 3: DRAFT with single spacing and default paragraph style. <n> = 4: 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.
-rE<n>	Controls font of Subject/Date/From fields. <n> = 0: fields are bold (default for *troff* formatter). <n> = 1: fields are Roman font (regular-text default for *nroff* formatter).
-rL<k>	Sets length of physical page to <k> lines. For *nroff* formatter, <k> is an unscaled number representing lines. For *troff* formatter, <k> must be scaled. Default value is 66 lines per page. This flag is used, for example, when directing output to certain printers.
-rN<n>	Specifies page numbering style. <n> = 0: (default) all pages get the prevailing header. <n> = 1: page header replaces footer on page 1 only. <n> = 2: page header omitted from page 1. <n> = 3: "section-page" numbering (.FD and .RP define footnote and reference numbering in sections. <n> = 5: default page header is suppressed, but user-defined header is not affected. <n> = 6: "section-page" and "section-figure" numbering is used.

<n>	Page 1 Numbering	Remaining Pages Numbering
0	header	header
1	header replaces footer	header
2	no header	header
3	"section-page" footer	same as page 1
4	no header	no header unless .PH defined
5	"section-page" footer, "section figure"	same as page 1

Contents of the prevailing header and footer do not depend on number register's <n> value. <n> controls only whether the header (<n> = 3) or footer (<n> = 5) is printed as well as the page-number style. If header and footer are null, the value of <n> is irrelevant.

−rO<k>	(Register name: uppercase "O") Offsets output <k> spaces to the right. For *nroff*, <k> is an unscaled number representing lines or character positions. For the *troff* formatter, <k> must be scaled. This flag is sometimes helpful for positioning output on terminals. Default, if not set on command line, is 1.9 cm (0.75 inch).
−rP<n>	Specifies that document page numbering begin with <n>. Can also be set by using *.nr* request in input text.
−rS<n>	Sets point size and vertical spacing for the document. Default <n> is 10-point type with 12-point vertical spacing, six lines per inch. Applies to *troff* formatter only.
−rT<n>	Provides register settings for certain output devices. <n> = 0: (default) no registers set. <n> = 1: line length = 80; page offset = 3. <n> = 2: 84 lines per page; underlining inhibited. Applies only to *nroff*.
−rU1	Controls section head underlining (applies to *nroff* only). Causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined.
−rW<k>	Sets page width (line length and title length) to <k>. For *nroff* formatter, <k> unscaled, represents character positions. For *troff*, <k> must be scaled. Can be used to change page width from default value of 6 inches (60 characters in 10-pitch or 72 characters in 12-pitch).

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 in preceding list>
.so /usr/lib/tmac/tmac.m
<remainder of text>
```

In this case, you must not use the *−cm* or *−mm* flag [nor the *mm*(1) or *mmt*(1) command]. The *.so* request has the equivalent effect, but the registers 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.

Formatting Concepts

Basic Terms

Normal formatter action fills (assembles or constructs) output lines from one or more input lines. Output lines can be justified so that both left and right margins are aligned. As lines are being filled, words can also be hyphenated when necessary. Any of these capabilities can be enabled or disabled by using *.SA*, *Hy*, and the *.nf* and *.fi* formatter requests. Disabling **fill** mode also disables justification and hyphenation.

A few formatter requests and most MM macros cause a **break**. When a break occurs, filling of the current output line ceases, the line is printed, and subsequent text begins on a new line (blank lines may be inserted before text resumes for separation of headings, paragraphs, or other items).

Formatter requests can be used with MM, but should be avoided when possible because of the consequences and side-effects each request might produce. Use MM macros (avoiding formatter requests except when necessary) because:

- Overall document style is easier to control and change when necessary.
- Complicated features such as footnotes and tables of contents are easily obtained.
- As a user, you are insulated from the peculiarities and complexities of the formatter language.

Arguments and Double Quotes

For any macro call, a null argument is an argument of zero width. Null arguments often have special meaning, and the preferred form is "". Omitting an argument is not equivalent to a null argument (as in the case of the *.MT* macro, for example). Null arguments can occur anywhere in an argument list, while omitted arguments are allowed only at the end of an argument list.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes. A double quote is a single character that is not to be confused with a consecutive pair of apostrophes (acute accents) or grave accents. If this convention is not carefully observed, the argument is treated as several separate arguments.

Double quotes are not permitted as part of the value of a macro argument. If it is necessary to have a macro argument value, two grave accents (` `) and/or two acute accents (' ') can be used instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times (as when headings are printed in text, then later in the table of contents).

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 text alignment. To prevent distortion, specify an **unpaddable space**, a space that cannot be expanded during justification, by:

- Typing a backslash followed by a space (\ ). This character pair directly generates an unpaddable space.
- Sacrificing some seldom-used character [such as a tilde (~)] to be translated into a space during output.

Character translation occurs after justification, so a chosen translation character can be substituted anywhere you want to place an unpaddable space. The tilde (~) is commonly used with the translation macro for this purpose. To use the tilde in this way, insert the following at the beginning of the document:

```
.tr ~
```

If you need to print a tilde in the actual output, you can temporarily "recover" by typing:

```
.tr ~~
```

before it appears in text. To resume its use as an unpaddable space, repeat the `.tr ~` after a break or after the line containing the tilde has been output.

Note

Do not use the tilde in this fashion when the tilde is used within equations.

Hyphenation

Formatters do not perform hyphenation unless requested. To disable hyphenation in a body of text, specify:

```
.nr Hy 1
```

once at the beginning of the document input file. Hyphenation within footnotes and across page boundaries is discussed in the Footnotes section of this tutorial.

Formatters hyphenate words automatically when hyphenation is enabled. You can override automatic hyphenation and specify where hyphenation is to be allowed in certain words. You can also specify that other words are not to be hyphenated. To exercise either option, use a hypenation indicator (initially the 2-character sequence "\%"). Placing the sequence immediately before a word prevents hypenation. If the sequence occurs within a word, hyphenation can occur only where the sequence has been inserted. During formatting, the sequence is removed from the input text, and is not included in the formatter output. You can specify hyphenation in a small list of words containing a total of up to about 128 characters.

To redefine the hyphenation sequence, use the command:

```
.HC <hyphenation indicator>
```

The circumflex (^) is commonly used for this purpose by inserting the command:

```
.HC ^
```

near the beginning of the document text file (before text starts).

Note

Any word containing hyphens or dashes (also known as *em* dashes) is hyphenated (if necessary) immediately after the hyphen or dash, even if the formatter hyphenation function is not enabled.

You can construct a small list of exception words with correct hyphenation points indicated by using the *.hw* request. Here are examples using the words "computer", "program", and "knowledge" (these and other words tend to be incorrectly hyphenated by typesetting formatters):

```
.hw com-put-er
.hw prog-ram
.hw know-ledge
```

Tabs

Macros *.MT*, *.TC*, and *.CS* use the formatter tabs (*.ta*) request to set tab stops. They then restore the default tab settings (every eight characters in *nroff* formatter; every half inch in *troff* formatter) when finished. Users are responsible for tab settings other than the default values.

Default tab settings are at 9, 17, 25, ..., 161 for a total of 20 tab-stop settings. When using the *ta* command, multiple tab values are separated by commas, spaces, or any other non-numeric character. Tab stops can be set to any desired value; for example:

```
.ta 9 17 25 46 72 84 106
```

Tab characters are interpreted with respect to their position in the *input line*; not their position in the *output line*. Therefore, in general, tab characters should be used *only* in lines that are processed in no-fill (*.nf*) mode (discussed near the beginning of this section).

The table formatter (*tbl*) changes tab stops, but does not restore default tab settings.

BEL Character

The non-printing character BEL is used as a delimiter in many macros to compute the width of an argument or to delimit arbitrary text such as in page headers and footers, headings, lists, and tables. Using BEL characters in input text files, and especially in arguments to macros, produces unpredictable output.

Bullets

A bullet (•) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". To maintain compatibility with the *troff* formatter, MM uses the following string to produce a bullet:

```
\*(BU
```

The bullet list (*.BL*) macro uses this string to automatically generate bullets for bullet-listed items.

Dashes, Minus Signs, and Hyphens

The *troff* formatter treats a dash, a minus sign, and a hyphen as dissimilar entities, while *nroff* treats them all identically. Therefore:

- If you use *nroff*, you can use the minus sign (-) for hypens, minus signs, and dashes.
- If you usually use *troff*, follow *troff* escape conventions.
- If you use both formatters, use care in input text preparation. Unfortunately, these graphic characters cannot be represented in a way that is both convenient and compatible with both formatters.

Here is a suggested approach:

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 automatically generate an em dash for each list item.
Hyphen	Type - and use as-is for both formatters. The *nroff* formatter will print it as-is, while the *troff* formatter prints a - (a true hyphen).
Minus	Type `\-` for a true minus sign, regardless of formatter. The *nroff* formatter ignores the "\"; *troff* prints a true minus.

Trademark String

A trademark string `\*(Tm` is included in MM. It places the letters "TM" one-half line above the text that it follows. For example:

```
The book
.I
Introducing the UNIX
.R
\h'-1'\*(Tm
.I
System
.R
is available from the library.
```

yields:

> The book *Introducing the UNIX™ System* is available from the library.

Use of Formatter Requests

Use of formatter requests should be generally avoided because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do basic formatter requests. However, some formatter requests are useful with MM, namely the following:

Request	Function
.af	Assign format
.br	Break
.ce	Center
.de	Define macro
.ds	Define string
.fi	Fill output lines
.hw	Hyphenation of 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 character

The *.fp*, *.lg*, and *.ss* requests are also sometimes useful for the *troff* formatter. Use of other requests without a full understanding of their implications frequently leads to disastrous results.

Paragraphs and Headings

Paragraphs

.P<type>
One or more lines of text.

The *.P* macro is used to control paragraph style.

Paragraph Indentation
An indented or non-indented paragraph is defined by the <type> argument.

<type>	Result
0	Left-justified
1	Indent

In a left-justified paragraph, the first line begins at the left margin. The first line of an indented paragraph is indented the amount specified by the *Pi* register (default value = 5). For example, to obtain a 10-space indent, include the following line near the beginning of the file before text:

```
.nr Pi 10
```

A document input file possesses a default paragraph type obtained by including .P before each paragraph that does not follow a heading. Default paragraph type is controlled by the *Pt* number register. The initial *Pt* value of zero provides left-justified paragraphs with no first-line indent.

To force a first-line indent in all paragraphs, place this command in the beginning lines of the document input file:

```
.nr Pt 1
```

To indent the first line of all paragraphs **except** those immediately following headings, lists, and displays, use this line instead:

```
.nr Pt 2
```

Both the *Pi* and the *Pt* register must have non-zero values before any paragraphs can be indented unless indentation is forced or suppressed by a .P0 or .P1 macro request.

Note

Values specifying indentation must be unscaled, and are treated as character positions; i.e., the number of ens. The *nroff* formatter treats an en as a single character width. In the *troff* formatter, an en is a space width; (in points) equal to half the current point size for characters.

To override *Pi* and *Pt* registers, use the .P0 macro request to force left justification; .P1 to indent the first line of an individual paragraph by the amount specified by *Pi*.

If the new-paragraph request, .P, occurs inside a list, the paragraph indent is added to the current list indent.

Numbered Paragraphs

Numbered paragraphs can be produced by setting register *Np* to 1. This produces paragraphs numbered within first-level headings (for example, 1.01, 1.02, 1.03, 2.01, etc.).

To produce numbered paragraphs with the text following the first line indented such that it is aligned with the beginning of text in the first line (so that the number stands out on the left side of the paragraph), use the *.nP* macro instead of *.P*. Paragraphs are numbered within second-level headings. Here is an example:

```
.H1 "FIRST HEADING"
.H2 "Second Heading"
.nP
```
One or more lines of text.

Spacing Between Paragraphs

Number register *Ps* controls the spacing between paragraphs. Default *Ps* value is **1**, yielding one blank space (one half of a vertical space).

Numbered Headings

```
.H <level> [<heading text>] [<heading suffix>]
```
Zero or more lines of text

The *.H* macro provides seven <level>s of numbered headings. Level 1 is the highest (most significant or dominant) order; level 7 the lowest (least significant) order. The <heading suffix> argument is added to the <heading text>, and can be used for footnote marks or other information that is not to be included in the table of contents.

Note

A *.P* macro is not needed immediately after a *.H* or *.HU* because the *.H* macro performs the *.P* function. However, in order to maintain uniformity throughout the text, it is a good practice to start every paragraph with a *.P*. A *.P* immediately following a heading command is ignored.

Normal Appearance

The effect of the *.H* macro varies according to argument <level>. First-level heads are preceded by two blank lines (one vertical space). All other heads are preceded by one blank line (one half of a vertical space). Other effects are as follows:

`.H 1` <heading text>	Produces a bold-font heading followed by a single blank line (one half of a vertical space). Text following the head begins on a new line and is indented according to current paragraph type. Use of full capital letters makes the heading stand out.
`.h 2` <heading text>	Produces a bold-font heading followed by a single blank line (one half of a vertical space). Text is handled the same way as first-level heads. Initial capitals are usually used in second level heads.
`.H` <n> <heading text>	Produces an underlined (italicized) heading followed by two spaces. <n> can have any value, 3 thru 7. Text following the head begins on the same line; i.e., these are run-in headings.

Appropriate numbering and spacing (both vertical and horizontal) occur, even if the <heading text> is omitted from the *.H* macro call.

If you are satisfied with normal default operation and appearance of headings, skip to the topic "Unnumbered Headings".

Altering Appearance of Numbered Headings

Heading appearance can be readily changed by setting certain registers and strings at the beginning of the document file. Thus, document style and appearance is easily altered by changing a few lines rather than editing commands throughout the document.

Prespacing and Page Ejection

First level headings (.H 1) are normally preceded by two blank lines (one vertical space); all other headings by one-half that much. If a multi-line heading would split across page boundaries, the entire heading is moved to the next page. You can force all first-level headings to the top of the next page by using:

```
.nr Ej 1
```

in the beginning part of the document input text file. To force second- or third-level heads to begin at the top of a new page (sometimes useful in long documents), change the number at the end of the command to correspond with the largest heading-level number to be moved to the next page. For example, to start all first-, second-, and third-level heads on a new page, use:

```
.nr Ej 3
```

Spacing After Headings

Three registers control text appearance following a *.H* call:

Hb **Heading break level** defines the lowest order head (highest <n>) that causes a break (in effect terminating the heading line). Headings with higher <n> values are not separated from the text that follows the heading (run-in headings). Default value is 2.

Hs **Heading space level** defines the lowest order head (highest <n>) that is always followed by a blank line (one half of a vertical space) before subsequent text. All heading levels of value <n> or smaller are automatically followed by a blank line. Default value is 2.

Hi **Post-heading indent** defines the indent style for text that follows the heading (stand-alone headings only; does not apply to run-in headings):

Hi = 0: text is left-justified.
Hi = 1 (default): indent according to paragraph type as defined by value of *Pt.*
Hi = 2: text is aligned with beginning of first word of heading so that the heading number stands out more clearly.

For example, to place a blank line after all first, second, and third-level heads; allow no run-in headings; and to force all text following headings to be left-justified (regardless of the value of *Pt*), use the following commands in the beginning of the document input text file:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

Centered Headings

Use *Hc* to obtain centered headings. Headings are centered when the value of the heading level (<n>) is less than or equal to the value of *Hc* **and** the heading is a stand-alone (not a run-in) heading. Default value of *Hc* is 0.

Bold, Italic, and Underlined Headings

Control by level: Any heading that is underlined by *nroff* is italicized by *troff*. The heading font string *HF* contains seven codes that specify fonts for heading levels 1 thru 7. Acceptable codes, code interpretations, and defaults for HF codes are:

	HF Code/Function			Default for Heading Level
Formatter	**1**	**2**	**3**	**1 2 3 4 5 6 7**
nroff	no underline	underline	bold	3 3 2 2 2 2 2
troff	Roman	italic	bold	3 3 2 2 2 2 2

Default for level 1 and 2 headings is bold. Remaining levels are underlined by *nroff*, italicized by *troff*. You can alter *HF* in any combination to meet your needs. Any value omitted from the right end (less significant headings) are assumed to have a value of 1. Thus, the following request produces bold for the first five levels and Roman font for the remaining two levels when processed by *troff*:

```
.ds HF 3 3 3 3 3
```

NROFF Underlining Style: The *nroff* formatter underlines in either of two styles:

- Normal style (*.ul* request) underlines only letters and digits.
- Continuous style (*.cu* request) underlines **all** characters including spaces.

By default, MM attempts to use continuous underlining on any heading that is to be underlined and short enough to fit on a single line. If the heading is to be underlined, but does not fit on a single line, normal underlining is used instead.

Use the `-rU1` flag when invoking *nroff* to force normal underlining of all headings in a document.

Heading point sizes: Use the *HP* string (*troff* only) to specify point size for each heading level as follows:

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, heading text (*.H* and *.HU* is printed in the same point size as the document body except that bold stand-alone headings are printed one point size smaller than the body. *HP* is similar to *HF*, and can be specified to contain up to seven values corresponding, left-to-right, to heading levels 1 thru 7. For example,

```
.ds HP 12 12 10 10 10 10 10
```

prints first and second-level heads in 12-point size; remaining heads in 10-point. Relative values can also be specified:

```
.ds HP +2 +2 -1 -1
```

If absolute sizes are specified, they are used independent of body text point size. Relative size is based on specified text body point size, even if the body point size changes. Only values on the right can be omitted. If not present or ZERO, default is assumed.

Note

Only the heading point size is affected by *HF*. If vertical spacing (user-exit macros *.HX* or *.HZ*) is not adjusted accordingly, overprinting may result.

Marking Styles: Numerals and Concatenation

Heading marks are (usually) numerical identifiers preceding the heading text. They are printed on the same line as the heading text.

```
.HM [arg1] . . . [arg7]
```

Registers *h1* thru *H7* are counters for the seven heading levels. Register values are normally printed using Arabic numerals. You can override the default to substitue other markings for use in outlines and other document styles. This macro can have up to seven arguments; each argument a string that indicates what marking type is to be used. Allowable arguments are:

Argument	Meaning
1	Arabic (default for all levels)
0001	Arabic with enough leading zeros 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.

The default heading mark construction successively appends the current mark value for each preceding heading level down to the current heading with a period between each value (for example first-level head #4, second-level head #5, third-level head #2, fourth-level head #1 produces the heading mark for the fourth level head: **4.5.2.1**). To suppress the concatenation of levels, reducing the mark to a single number followed by a period, set the heading-mark **type** register *Ht* value to 1. For example, a commonly used "outline" document style is obtained with the following sequence of requests:

```
.HM I A 1 a i
.nr Ht 1
```

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 the counters for levels 3 through 7 to zero. Usually, the value of *Hu* should be set to make unnumbered headings (if any) be the highest-order (lowest <n>) 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 for the main body of a document.

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:

1. Use the content level register *Cl* to specify what level headings are to be saved, and
2. Invoke the *.TC* macro 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.

Because of how headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many heading levels in a large document while also processing displays and footnotes. If this happens the `Out of temp file space` formatter error message (Table 4) is issued. The only remedy is to save fewer levels and/or reduce the number of words in the headings being saved.

First-Level Headings and Page Numbering Style

Unless you specify otherwise, 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 *mm* command line. As a side effect, this also sets *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 to ensure that the correct section number is printed.

User Exit Macros

Note

This topic 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>

Use the *.HX*, *.HY*, and *.HZ* macros to obtain a final level of control over the previously described heading mechanism. These macros are not defined by MM – they 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 you can respecify these features. All-default actions occur if these macros are not defined. If you have defined *.HX*, *.HY*, or *.HZ*, the definition you supplied is interpreted at the appropriate time. Therefore, these macros can influence handling of all headings because the *.HU* macro is actually a special case of the *.H* macro.

If you originally invoked the *.H* macro, the derived level (<dlevel>) and the real level (<rlevel>) are both equal to the level given in the *.H* invocation. If you originally invoked the *.HU* macro, <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, produced blank lines (vertical spaces) to precede the heading, and accumulated the "heading mark" (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 <rlevel> is nonzero, this string contains the "heading mark". Two unpaddable spaces (to separate the mark from the heading) have been appended to this string. If <rlevel> is 0, this string is null.
register ;0	This register indicates what type of spacing is to follow the heading. 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 break **and** a blank line (one-half a vertical space) are 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 the *.ne* request that is issued before the heading is actually printed. On entry to *.HX*, its value is 3 if <dlevel> = 1; otherwise it is 1. The number of lines requested by *.ne* includes a number of blank lines equal to the value contained in "register ;3" (one-half vertical space per line) plus the number of lines in the heading.

You can alter the values of }0, }2, and ;3 within *.HX*. Here are some examples of actions that can be performed by defining *.HX* to include the lines shown:

To 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)

To separate a run-in heading from the text with a period and two unpaddable spaces:

```
.if\\n(;0=0 .ds }2 .\<sp>\<sp>
```

To ensure that at least 15 lines are left on the page before printing a first-level heading:

```
.if\\$1=1 .nr ;3 15-\\n(;0
```

To add three additional blank lines before each first-level heading:

```
.if\\$1=1 .sp 3
```

To indent level 3 run-in headings by five spaces:

```
.if\\$1=3 .ti 5n
```

If temporary strings or macros are used within *.HX*, choose their names with care.

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; you may want to change a page header or footer, e.g.:

```
.de HZ
.if\\$1=1 .PF"Section\\$3"
...
```

Hints for Large Documents

For convenience, a large document is often organized into one input text file per section. If the files are numbered, it is helpful to use enough digits in the names of these files to accommodate the maximum number of sections; i.e., use suffix number 01 through 20 rather than 1 through 9 and 10 through 20.

If you want to format individual sections of long documents with the correct section numbers, it is necessary to set register *H1* to one less that the number of the section just before the corresponding `.H\|t1` call. For example, at the beginning of Section 5 of a document, 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.

Lists

This section 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.

List Macros

In order to avoid repetitive typing of arguments that define 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 can 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, you can create your own customized sets of list macros with relatively little effort.

List-Initialization Macros

List-initialization macros are implemented as calls to the more basic *.LB* macro.

They are:

.AL	Automatically Numbered or Alphabetized List
.BL	Bullet List
.DL	Dash List
.ML	Marked List
.RL	Reference List
.VL	Variable-item List

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 *.AL* was 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 postions"; i.e., number of ens.

Spacing before the list and between list 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 (the maximum list nesting level).

The <type> argument can be included to obtain a different type of sequencing. Its value indicates the first element in the sequence desired. If <type> is omitted or null, <type> = 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> is null, the value of *Li* is used.

If the third argument [1] is given, a blank line (one-half of a vertical space) is produced before the first item, but no blank lines are output between items in the list.

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). In the default case, the text of bullet and dash lists lines up with the beginning of the first line of indented paragraphs.

If the second argument [1] is specified, no blank lines separate items in the list.

Dash List

.DL [<text indent>] [1]

The *.DL* macro is indentical to *.BL* except that a dash is used as the list item mark instead of a bullet.

Marked List

.ML <mark> [<text indent>] [1]

The *.ML* macro is much like *.BL* and *.DL* macros but expects you to specify an arbitrary mark which can consist of more than a single character.

Text is indented <text indent> spaces if <text indent> is not null; otherwise, the text is indented one more space than the width of <mark>.

If the third argument [1] is specified, no blank lines separate items in the list.

Note

<Mark> must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified.

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 beginning of 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* used.

If the second argument [1] is specified, no blank lines separate items in the list.

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 commonly 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 the beginnning of the mark, and defaults to 0 if omitted or null.

If the third argument [1] is specified, no blank lines separate items in the list.

Here is an example of *.VL* macro usage:

```
.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 space between
"mark" and "1".
.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 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".

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.

In the preceding example, the tilde argument on the last `.LI` is required. Otherwise, a "hanging indent" would have been produced. To produce a "hanging indent", use *.VL* and call *.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

<Mark> must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified.

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 creates a single blank line (one-half a vertical space) before its list item although this can 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.

If the current mark (in the current list) is a null string **and** the <mark> argument of *.LI* is omitted or null, the resulting effect is that of a "hanging indent"; i.e., the first line of the subsequent text is moved to the left, starting at the same place where <mark> would have started.

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, *.LE* outputs a blank line (one-half of 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 *.P*, *.H*, or *.LI*.

.H and *.HU* automatically clear all list information. You can omit the *.LE* macros that would normally occur just before either heading macro to prevent receiving an `LE:mismatched` error message, but such a practice is not recommended because errors will result if the list text is separated from the heading at some later time (such as when additional text is inserted).

Example of Nested Lists

Here is an example of input for several lists. The corresponding output follows. The *.AL* and *.DL* macro calls shown are examples of list-initialization macros. Input text is:

```
.AL
.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 lst 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 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.

List-Begin Macro and Customized Lists

`.LB` <text indent> <mark-indent> <pad> <type> [<mark>] [<LI space>]\[<LB space>]

List-initialization macros described previously suffice for almost all cases. However, you may occasionally need more control over the layout of lists, obtainable by using the basic list-begin macro (*.LB*). *.LB* is used by the other list-initialization macros, and 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). <Mark indent> is usually set to 0.
- Within the mark area, the <mark> is left-justified if <pad> is 0. If <pad> is greater than 0, the number of blanks indicated by the value of <pad> are appended to the <mark>; and 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). This is summarized in the following table.

type	mark	result
0	omitted	hanging indent
0	string	string is the mark
>0	omitted	Arabic numbering
>0	one of: 1, A, a, I, i	alphabetic sequencing

Each non-zero 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.

- 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, *.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 of 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 <LI space> = 1 and <LB space> = 0, yielding one blank line before each item in the list. Such lists are usually terminated with a *.LE 1* macro to insert a blank line following the list.
- For a more compact list, <LI space> = 0 and <LB space> = 1, yielding no space between items and a blank line before and after the list. The *.LE 1* macro is used at the end of the list.
- If both <LI space> and <LB space> are zero and *.LE* is used to end the list, a list with no blank lines results.

The next topic shows how to build upon the supplied list of macros to obtain other kinds of lists.

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 promotes 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. For example, a macro could be constructed such that levels 1 through 5 of the list to be formatted have the following appearance:

A.

[1m

(•) bullet here

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, remember 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. It 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>\\ng .LC \\ng            \"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.

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 some large corporations) which have special requirements for the first page and for the cover sheet. Data needed (title, author, date, case numbers, etc.) is entered in the same way for both styles; an argument to the *.MT* macro indicates which style is being used.

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>] [<text>] [<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 other associated input lines) can 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 input text. If these macros are omitted, the first page will have only the page header followed by the body of the document.

Title

```
.TL [<charging case>] [<filling 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 numbers are entered similarly. For example:

```
.TL "12345,67890" 987654321
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 can 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, the "Charging and Filing Case" followed by the number will appear on a line following the title. If the two case numbers are dirrerent, separate lines for "Charging Case" and "File Case" will appear after the title.

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 can be given. Each will appear in the signature block for memorandum style 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 PH 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 can be suppressed on the first page of a memorandum by setting register *Au* to 0; (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 can 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, the company name and location are centered. The paragraph on memorandum types contains information regarding authors from different locations.

TM Numbers

```
.TM [<number>] ...
```

If the memorandum is a technical memorandum, the TM numbers are supplied via the *.TM* macro. Up to nine numbers can be specified. For example:

```
.TM 7654321 77777777
```

This macro call is ignored in the released-paper and external-letter styles.

Abstracts

```
.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 (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.

In released-paper style (<type> argument of the .MT macro is 4), and in technical memorandum style, if the first argument (<arg>) of *.AS* is:

0 Abstract is printed on page 1 and on the cover sheet.

1 Abstract apppears only on the cover sheet (if any).

In memoranda-for-file style and in all other documents (other than external letters) if the <arg> of *.AS* is:

0 Abstract appears on page 1 and no cover sheet is printed.

2 Abstract appears only on the cover sheet which is produced automatically (i.e., without invoking the .CS macro).

No abstract or cover sheet can be obtained with an external-letter style (<type> argument of .MT macro is 5).

Notations such as a "copy to" list are allowed on memorandum-for-file cover sheets; *.NS* and *.NE* commands must apppear after *.AS 2* and *.AE*. Headings and displays are not permitted within an abstract.

The abstract is printed with ordinary text margins (the indentation to be used for both margins can be specified as the <indent> argument of .AS). Values that specify indentation must be unscaled, and are treated as "character positions"; i.e., as the number of ens.

Other Keywords

```
.OK [<keyword>] ...
```

Topical keywords should be specified on a technical memorandum cover sheet. Up to nine such keywords or keyword phrases can be specified as arguments to the *.OK* macro. If any keyword contains spaces, it must be enclosed within double quotes.

Memorandum Types

```
.MT [<type>] [<addressee>]
```

The *.MT* macro controls the format of the top part of the first page of a memorandum or released-paper document, and the format of the cover sheets. Allowable <type> arguments and corresponding document functions are:

Type	Document Function
" "	No memorandum type printed
0	No memorandum type printed
(omitted)	MEMORANDUM FOR FILE
1	MEMORANDUM FOR FILE
2	PROGRAMMER'S NOTES
3	ENGINEER'S NOTES
4	Released-paper style
5	External-letter style
"string"	String (enclosed in quotes)

If <type> indicates a memorandum-style document, the corresponding statement indicated under "Document Function" is printed after the last line of author information. If <type> is longer than one character, the <type> string itself is repinted as in this example:

```
.MT "Technical Note #5"
```

A simple letter is produced by calling *.MT* with a null (but not omitted) or 0 argument.

The <addressee> argument to *.MT* is the letter-addressee's name. 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 cannot be used if the <type> 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 has been edited to specify a different company). If the optional second argument to .MT 4 is given, 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 company location codes are defined as strings that are the addresses of the corresponding company locations. These codes are needed only until the *.MT* macro is invoked. Thus, following the *.MT* macro, the user can reuse these string names, In addition, the macros for the end of a memorandum and their associated lines of input are ignored when the released-paper style is specified.

Company locations in "as-shipped" MM refer to Bell Telephone Laboratories. To use other company locations, edit the *.MT* macro, or include author affiliations in the released-paper style by specifying the appropriate *.AF* macro, 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.

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.

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 logo or company-name stamp.

.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`, 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` do not affect *.AF*. The only *.AF* use appropriate for *troff* is when specifying a replacement for "Bell Laboratories".

The command line option `-rEn` controls the font of the "Subject/Date/From" block.

Example
Input text for a typical document could resemble the following:

```
.TL
MM/*(EMMemorandum Macros
.AU "H. R. Douglas" DWS PY
.Au "E. S. Mahoney" JRM PY
.AU "W. C. Archer (January 1984 Revision)" ECP PY
.AU "R. E. Taylor (June 1984 Revision)" NWS PY
.MT 4
```

End-of-Memorandum Macros
At the end of a memorandum document (but not 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.

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 can be specified as an argument to *.FC*.

.SG 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 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) are omitted.
- 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 the second argument is given, reference data is placed on the same line as 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 invocation (without arguments) of the *.SG* macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

"Copy to" and Other Notations

```
.NS [<arg>]
zero or more notation lines
.NE
```

Many types of notation (such as a list of attachments or "Copy to" lists) can 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 printed notations are:

<arg>	Notations
none	Copy to
" "	Copy to
0	Copy to
1	Copy to
2	Copy (with att.) to
3	Att.
4	Atts.
5	Enc.
6	Encs.
7	Under Separate Cover
8	Letter to
9	Memorandum to
"string"	Copy (string) to

If <arg> consists of more than one character, it is placed whithin parentheses between the words "Copy" and "to". For example:

```
.NS "with att. 1 only"
```

generates

```
Copy (with att. 1 only) to
```

as the notation. More than one notation can 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
```

.NS and *.NE* can also be used at the beginning after *.AS 2* and *.AE* to place the notation list on the memorandum-for-file cover sheet. If notations are given at the beginnning without *.AS 2*, they are saved for output at the end of the document.

Approval Signature Line

```
.AV <approver's name>
```

The *.AV* macro can 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

One-Page Letter

At times, you may want more space on the page, thus 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 `-rLn` 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 for placing the signature or notations.

Note

This works **only** for one-page letters and memos.

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 and centering, as well as fill-mode processing.

Note

Displays and footnotes cannot be nested in any combination. Although lists and paragraphs are permitted, no headings (.H or .HU) can be used within displays or footnotes.

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:

<format>	Meaning
" "	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:

<fill>	Meaning
" "	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 for *nroff*, and is treated as ens. It can be scaled when using *troff*, and 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. Even though their initial values are the same (default), these two registers are independent.

The display format argument value 3 (or CB) horizontally centers 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 the width of the longest line). With *eqn/neqn*, this format must be used in order for the "mark" and "lineup" features to work when dealing with centered equations.

By default, a blank line (one half of 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 register *Ds* to 0.

The following example shows how to use 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 for the United States of America."

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 that 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 of a vertical space) occurs before and after a floating display.

You can exercise precise control over the output positioning of floating displays by using two number registers, *De* and *Df*. When a floating display is encountered by *nroff* or *troff*, it is processed and placed in a queue of displays waiting for output. Displays are removed from the queue and printed in the order entered, which is the same order that they appeared in the input file. If a new floating display is encountered and the display queue is empty, the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the *Df* register code setting. The *De* register code controls whether any additional text will follow the display if it is printed on the current page.

As long as the display queue contains one or more displays, new displays are automatically placed 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 display-output processing is complete, the display is removed from the queue.

When the end of a section (using section-page numbering) or the end of a document is reached, all remaining displays are automatically output and removed from the queue. 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).
1	Each floating display is followed by a page eject such that only one floating display appears on a given page. No other text follows the display.

Any *De* <code> value other than 0 is interpreted the same as <code> = 1.

***Df* Register**

Code	Action
0	Floating displays are not output until end of section (if section-page numbering is enabled) or end of document (all other cases).
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, beginning at the top of a new page (or column when in 2-column mode).
3	Output one floating display on current page (or column) if there is space; otherwise, begin output at 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.
5	Output a new floating display on the current page if there is room (also the default condidition). Output as many displays (at least one) as will fit on the page starting at the top of a new page or column.

Note

If *Df* <code> = 4 or 5, and *De* <code> = 1, each display is always followed by a page eject causing a new top of page to be reached. This causes the output of at least one more display.

Also, for any *Df* <code> greater than 5, the action performed is the same as when <code> = 5.

Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

You can use the .TS (table start) and .TE (table end) macros to access the *tbl*(1) program. These macros delimit text to be examined by *tbl* and set proper spacing around the table. The display function and the *tbl* delimiting function are independent. To ensure that blocks containing any mixture of tables, equations, filled test, unfilled text, and caption lines are kept together, the .TS/.TE block should be enclosed within a display (.DS/.DE). Floating tables can be enclosed inside floating displays (.DF/.DE).

You can also use .TS and .TE to process tables that extend over several pages.

If a table heading is needed for each page of a multipage table, the H argument should be included with the .TS macro call as above. Type the table title after the options and format information, using as many lines as you need, then follow the title with the .TH macro. The .TH macro must be included when .TS H is used for multipage tables (this restriction is imposed by MM, not *tbl*).

You may need to build long tables from smaller .TS H/.TE segments, requiring that table headers be suppressed for all tables except the first table on a page. To do this, use the .TH (table header) macro with the argument N. The N suppresses all table headers **except** the first header appearing on a given page. 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
```

outputs a table heading at the top of the first table segment and suppresses the heading at the top of the second segment when both appear on the same page. However, if the table continues at the top of the next page, a new header is output at the top of each page until the end of the table is reached.

This feature is helpful when a single table must be broken into segments because of table complexity (such as 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 appears 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)] can be used for terminals (such as the DASI 450) that are capable of finer printing resolution. This produces better alignment of features such as lines forming the corner of a box. The −e option does not affect *col*(1).

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 is used as a label for the equation.

The default label location is at the right margin in the "vertical center" of the general equation. Set the *Eq* register to 1 to move the label position to the left margin.

The equation is centered for centered displays. Otherwise, the equation is adjusted to the margin opposite the label.

Figure, Table, Equation, and Exhibit Titles

```
.FG [<title>] [<override>] [<flag>]
.TB [<title>] [<override>] [<flag>]
.EC [<title>] [<override>] [<flag>]
.EX [<title>] [<override>] [<flag>]
```

The `.FS` (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. 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 *.EX* replaces "Figure" with "Exhibit". The output title is centered if it can fit on a single line. Otherwise, the first line is centered, and the remaining title lines are lined up with the first character of the first line Number format can be changed by using the *.af* formatter request. Caption format can 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> is omitted or 0, <override> is used as a prefix to the number; if <flag> = 1, <override> is used as a suffix; and if <flag> = 2, override replaces the number. If `-rN5` is given, "section-figure" numbering is set automatically and any 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.

List of Figures, Tables, Equations, and Exhibits

A list of figures, tables, exhibits, and equations are printed following the table of contents if number registers *Lf*, *Lt*, *Lx*, and *Le* are all set to 1. Default value for *Lf*, *Lt*, and *Lx* is 1; default for *Le* is 0.

Titles of these lists can be changed by redefining the following strings (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
```

Footnotes

There are two macros (.FS and .FE) that delimit footnote text, 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).

Automatic Footnote Numbering

Footnotes can 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 places the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

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 *.FS*, if present, is used to mark footnote text. Otherwise, the number retrieved from the string *F* is used.

Automatically numbered and user-labeled footnotes can 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, 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.
```

Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

Footnote text (enclosed within the .FS/.FE pair) should immediately follow the word to be footnoted in the input text, so the "*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 to "^*F" or label when they follow an end-of-sentence punctuation mark (i.e., period, question mark, or exclamation point).

The example at the end of this tutorial illustrates several footnote styles as well as numbered and labelled footnotes.

Footnote Text Format Style

`.FD [<arg>] [1]`

You can control formatting style within the footnote text 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 followng 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 definition of the *.ad*, *.hy*, *.na*, and *.nh* (adjust, hyphenation, no adjust, and no hyphenation, respectively) requests in the *nroff* tutorial.

<arg>	HYPHENATION	ADJUST	TEXT INDENT	LABEL JUSTIFICATION
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 the 11, it is handled as if `.FD 0` were specified. If the first argument is omitted or null, the effect is equivalent to `.FD 10`" in *nroff* and `.FD 0` in *troff*. These are also the respective initial default values.

If the second argument is specified, automatically-numbered footnotes begin again with 1 whenever a first-level heading is encountered. 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. You can suppress end-of-page hyphenation by specifying an even .FD argument.

Footnotes are separated from the body to the text by a short line rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. *Troff* sets footnotes in a type size two points smaller than the text point size.

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, number register *Fs* is set to the desired value. For example:

```
.nr Fs 2
```

produces two blank lines (a 6-point vertical space) between footnotes.

Page Headers and Footers

Text printed at the top of each page is called a **page header**. Text printed at the bottom of each page is called a **page footer**. Up to three lines of text can be associated with the header. The first line is printed on every page, the second on even pages; the third on odd pages only. Thus each printed page header can have up to two lines of text: the line that occurs at the top of every page, and the line for even- or odd-numbered pages. The same arrangement applies to page footers.

This section describes the default appearance of page headers and footers and explains how to change them. The terms **header** and **footer** (not qualified by even or odd) is used to mean the page-header or page-footer line that occurs on every page.

Default Headers and Footers

Each page has a default page header consisting of a centered page number. There is no default footer or default even/odd header/footer except in section-page numbering schemes explained later in this section.

In a memorandum or a released-paper style document, the page header on the first page is automatically suppressed unless a break occurs before the .MT macro is called. Macros and text in the following categories do not cause a break, and can precede the memorandum types (.MT) macro in an input text file.

- Memorandum and released-paper style document macros (*.TL*, *.AU*, *.AT*, *.TM*, *.AS*, *.AE*, *.OK*, *.ND*, *.AF*, *.NS*, and *.NE*)
- Page header and footer macros (*.PH*, *.EH*, *.OH*, *.PF*, *.EF*, and *.OF*)
- The *.nr* and *.ds* requests.

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 an apostrophe (') as the delimiter because it occurs within one of the parts, it can be replaced uniformly by any other character. In formatted output, the parts are left-justified, centered, and right-justified, respectively.

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 can be changed by using the *.af* macro request.

If "debug mode" is set by including the `-rD1` flag on the command line, 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) followed by the current line number within the current input file.

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.

Odd-Page Header

```
.OH [<arg>]
```

The *.OH* macro is identical to *.EH* except that it applies to odd-numbered pages.

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, 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 instead of being a blank line.

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.

Odd-Page Footer

```
.OF [<arg>]
```

The *.OF* macro is identical to *.EF* except that it applies to odd-numbered pages.

First-Page Footer
The default first page footer is a blank line. If you specify `.PF` and/or `.OF` before the end of the first page of the document, the specified lines will be printed 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.

Default Header and Footer With Section-Page Numbering

Pages can be numbered sequentially within sections by "section-number page-number". 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, and the default page header is blank.

Strings and Registers in Header and Footer Macros

String and register names can 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 because string or register invocation is 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. For example:

```
.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, specify (see Numbered Headings Marking Styles topic for explanation of *H1*):

```
.PF"'''-\\\\n(H1-\\\\nP'"
```

If you arrange for string *a]* to contain the current section heading that 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.

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''Revision3'"
.OH"'Revision3"\\\\nP'"
```

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.
- *.PX* is a page header user-exit macro that is invoked (without arguments) when the normal environment has been restored and the "no-space" mode is already in effect.

The default 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
...
```

String *}t* contains the header, string *}e* contains the even-page header, and string *}o* contains the odd-page header as defined by the *.PH*, *.EH*, *.EH*, and *.OH* macros, respectively. To obtain specialized page titles, redefine the *.TP* macro to produce the desired header processing. Formatting within *.TP* 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, *.TP* could be defined as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
...
```

The *.PX* macro can be used to provide text that is to appear at the top of each page after the normal header. Tab stops can be included to align it with columns of text in the body of the document.

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 are 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 appears on the table of contents, text pages, and the memorandum-for-file cover sheet, but it is not printed on technical memorandum or released-paper cover sheets.

Top and Bottom (Vertical) Margins

```
.VM [<top>] [<bottom>]
```

The *.VM* (vertical margin) macro is used to specify additional space at the top and bottom of the page. This space precedes the page header and follows the page footer. *.VM* 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 can be decreased by redefining *.TP*).

Proprietary Marking

```
.PM [<code>]
```

The *.PM* (proprietary marking) macro appends a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer to the page footer. Allowable <code> arguments include:

<code>	Disclaimer
none	turn off previous disclaimer, if any
P	PRIVATE
N	NOTICE
BP	BELL LABORATORIES PROPRIETARY
BR	BELL LABORATORIES RESTRICTED

You can alter the disclaimer options by editing the *.PM* macro. To alternate disclaimers, use the .BS/.BE macro pair.

Private Documents

```
.nr Pv <value>
```

The word "PRIVATE" can 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:

<value>	Meaning
0	do not print PRIVATE (default)
1	PRIVATE on first page only
2	PRIVATE on all pages

If <value> = 2, the user-definable *.TP* macro cannot 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.

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.)

These macros normally appear once at the end of the document, after the Signature Block and Notations macros, and can 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 is produced at the end because it may not be wanted.

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 *CI* register. Arguments to *.TC* control: spacing before each entry, placement of associated page numbers, and any additional text preceding the word "CONTENTS" on the first page of the table of contents.

Spacing before each entry is controlled by <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 of a vertical space). The <slevel> argument does not control what heading levels have been saved (saving headings is the function of the *CI* register).

<Tlevel> and <tab> control placement of the 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 not more than four arguments, either 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 four or fewer arguments, you 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 was defined at *.TY*, the word "CONTENTS" is suppressed. Defining *.TY* as an empty macro suppresses "CONTENTS" with no replacement:

```
.de TY
...
```

By default, first-level headings are left-justified in the table of contents. Subsequent levels are then aligned with the text of headings at the preceding level. These indentations can 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, if *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*. Default page numbering for table of contents is lowercase Roman numeral. If *Oc* is set to 1, the *.TC* macro does not print any page number but instead resets the *P* register to 1. It is the user's responsibility to provide an appropriate page footer to specify page number placement. The same *.PF* (page footer) macro used in the document body is usually adequate.

The list of figures, tables, and such is produced separately unless *Cp* is set to 1, causing these lists to appear on the same page as the table of contents.

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 (cover sheets were discussed in released-paper section). All other information for the cover sheet is obtained from data given before the `.MT` macro call. If the technical memorandum style is used, the *.CS* macro generates the "Cover Sheet for Technical Memorandum". Data appearing in the lower left corner of the technical memorandum cover sheet (counts of: pages of text, other pages, total pages, figures, tables, and references) is generated automatically (0 is used for "other pages"). These values can be changed by supplying the corresponding arguments to the *.CS* macro. If the released-paper style is used, all arguments to *.CS* are ignored.

References

There are two macros (.RS and .RF) that delimit reference text, a string that automatically numbers the subsequent references, and an optional macro (.RP) that produces references pages within the document.

Automatic Numbering of References

Automatically numbered references can be obtained by typing `\*(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.

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
```

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 can be used later in the document as the string call `\*(aA` to reference text that must be labeled with a prior reference number. The reference, as output, is enclosed in brackets one-half line above the text to be referenced. No `.RS`/`.RF` pair is needed for subsequent references.

Reference Page

`.RP` [<arg1>] [<arg2>]

A reference page with the default title, "References", is generated automatically at the end of the document (before table of contents and cover sheet) and is listed in the table of contents. This page contains the reference items (i.e., reference text enclosed within .RS/.RF pairs). Reference items are separated by a space (one-half of a vertical space) unless the *Ls* register is set to 0 to suppress this spacing. You can change the reference page title by defining the *Rp* string.

```
.ds Rp "New Title"
```

The *.RP* (reference page) macro can be used to produce reference pages anywhere else within a document (such as after each major section). *.RP* is not needed to produce a separate reference page with default spacings at the end of the document.

The two *.RP* macro arguments are used to control resetting of reference numbering and page skipping.

<argl>	Meaning
0	reset reference counter (default)
1	do not reset reference counter

<arg2>	Meaning
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 *.RP*, a single blank line separates the references from the following/preceding text. You may want to adjust spacing. For example, to produce references at the end of each major section:

```
.sp 3
.RP 1 2
.H 1"Next Section"
```

Miscellaneous Features

Bold, Italic, and Roman Fonts

```
.B [<bold arg>] [<previous-font arg>] . . .
.I [<italic arg>] [<previous-font arg>] . . .
.R
```

When called without arguments, *.B* changes the font to bold; *.I* to underlining (italic). This condition continues until *.R* occurs, causing the Roman font to be restored. Thus:

```
.I
here is some text.
.R
```

produces underlined text from *nroff* and italic text from *troff*(1)

If the *.B* or *.I* macro is called with one argument, that argument is printed in the appropriate font (*.I* is underlined by *nroff*), then the previous font is restored after argument output is complete. If two or more arguments (maximum six) are included 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) and 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 the specified font with the one that was prevailing when the macro was invoked. To alternate specific pairs of fonts, use one of the following macros:

.IB .BI .IR .RI .RB .BR

Each macro takes a maximum of six arguments and alternates arguments between the specified fonts, in the order specified.

If your output terminal cannot underline, insert the following at the beginning of the document to eliminate all underlining:

```
.rm ul
.rm cu
```

Note that font changes in headings are handled separately.

Right Margin Justification

```
.SA [<arg>]
```

The *.SA* macro sets right-margin justification for main-body 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 *nroff* and for justification in *troff*.

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 so you can specify justification or no justification for the remainder of the text by inserting ".SA 0" or ".SA 1" once at the beginning of the document.

SCCS Release Identification

The *RE* string contains the SCCS release and the MM text formatting macro package current version level. For example:

```
This version \*(RE of the macros.
```

produces

This 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` on the command line. This causes the *RE* string to be output as part of the page header.

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 treated 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.

You can have full-page-width footnotes and displays when in 2-column mode, although the default 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:

<arg>	Meaning
N	Default
WF	Wide footnotes (even in 2-column mode).
-WF	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
-FF	DEFAULT: Turn off FF. Footnote style follows settings of WF or -WF.
WD	Wide displays (even in 2-column mode).
-WD	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.
-FB	Floating displays on current page do not cause a break.

Note

The `.WC WD EF` command produces wide displays and footnotes on the current page while `.WC N` reinstates default actions. If conflicting `.WC` settings are given, the last one encountered is used. `.WC WF -WF` is treated as `.WC -WF`.

Column Headings for Two-Column Output

Note

This section is intended only for users accustomed to writing formatter macros.

When handling 2-column output formats, 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 to provide header lines for both the entire page and each of the two 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 ⇒ stands for the tab character.

This example produces two lines of page header text plus lines of headers over each column. Tab stops are for the 65-en overall line length.

Vertical Spacing

```
.SP [<lines>]
```

Several methods can be used to obtain vertical spacing, each with different effects. The *.sp* request spaces the number of <lines> specified unless the no-space (*.ns*) mode is on, in which case the *.sp* request is ignored. 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. No-space can be disabled by the *.rs* (restore spacing) request.

.SP is used to avoid the accumulation of vertical space by successive macro calls. Several successive *.SP* calls do 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 use *.SP* for spacing. For example, `.LE 1` immediately followed by `.P` produces only a single blank line (one-half of 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 values are acceptable.

.SP (as well as *.sp*) is also inhibited by the *.ns* request.

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 (in which case it does nothing). `.SK` <n> skips <n> pages. For example, `.SK` always positions text that follows it at the top of a new page (skip to next page), while `.SK 1` always leaves one page blank except for the header and/or footer on the blank page.

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.

Setting Point Size and Vertical Spacing

`.S` [<point size>] [<vertical spacing>]

The default *troff* point size (obtained from the MM register *S* is 10 points, and the vertical spacing is 12 points (six lines per inch). Prevailing point size and vertical spacing can be changed by invoking the *.S* macro.

The mnemonics **D** (default value), **C** (current value), and **P** (previous value) can 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, *.S* defaults to **P**.
- If the first argument is specified but the second is not, the (default) **D** is used for the vertical spacing.

Default value for vertical spacing is always two points greater than the current point size. Footnotes 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
.Sn""     =.S n C
.Sn       =.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 tha 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
```

You can use the *.SM* macro to reduce the size of a string by one point:

`.SM` <string1> [<string2>] [<string3>]

If <string3> is omitted, <string1> is made smaller and is concatenated with the <string2> if <string2> is specified. If <string1>, l<string2>, and <string3> are present (even if any are null), <string2> is made smaller and all three strings are concatenated. For example:

Input	Output
`.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

Producing Accents

The following strings can be used to produce accents for letters:

	Input	Output
Grave accent	`c\*'`	c̀
Acute accent	`e\*'`	é
Circumflex	`o\*^`	ô
Tilde	`n\*-`	ñ
Cedilla	`c\*,`	ç
Lowercase umlaut	`u\*:`	ü
Uppercase umlaut	`U\*;`	Ü

Inserting Text Interactively

`.RD` [<prompt>] [<diversion>] [<string>]

The *.RD* (read insertion) macro stops standard document output and reads text from the standard input until two consecutive newline characters are found. When the two newline characters are encountered, normal output is resumed.

- <Prompt> is printed at the terminal. If not given, *.RD* signals the user with a BEL on terminal output.
- <Diversion> saves all text typed in after <prompt> in a macro whose name is that of the <diversion>.
- <String> saves the first line following the prompt in the named string for future reference.

.RD follows the formatting conventions currently in effect (the following example assumes that *.RD* is invoked in no-fill mode [*.nf*]). The following command:

```
.RD Name aA bB
```

combined with the following text from the standard input device:

```
Name:  J. Jones
16 Elm Rd.,
Piscataway
```

produces the following in the *.aA* macro:

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string bB (`\*B`) contains "J. Jones".

A newline character from the input device followed by an EOF (user-specifiable CONTROL d) resumes normal output.

Errors and Debugging

Error Terminations

When a macro detects an error, the following sequence is followed:

1. A BREAK occurs.
2. The formatter output buffer (which may contain some text) is printed to avoid confusion regarding location of the error.
3. 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).
4. Processing terminates unless regiser *D* has a positive value, in which case processing continues even though the output is guaranteed to be incorrect 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 as a result of 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 `-o` list option of the formatter causes the last page of the document to not be printed, a harmless "broken pipe" message may result.

Disappearing Output

Output disappearances are usually the result of an unclosed diversion (such as a missing .DE or .FE macro call). 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 '^\.[EDFR]EFNQ]' <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.

Extending and Modifying MM Macros

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 in a single internal table by the formatters. Therefore, there must be no duplication among such names. Number register names are kept in a separate table.

Names Used by Formatters

requests:	aa (most common) an (currently only one: c2)
registers:	aa (normal) .x (normal) .s (currently only one: .$) a. (currently only one: c.) % (page number)

Names Used by MM

macro and strings:	A, AA, Aa (accessible to users; e.g., macros P and HU, strings F, BU, and Lt) nA (accessible to users; currently only two: 1C and 2C) aA (accessible to users; currently only one: nP) s (accessible to users; currently only the seven accents))x, }x,]x, >x, ?x (internal)
register	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)

Names Used by CW, EQN/NEQN, and TBL Programs

The *cw*(1) program is the constant-width font preprocessor for *troff*. 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+ al nn na ^a #a #s

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. Here is a possible naming convention.

macros:	aA (e.g., *bG*, *kW*)
strings:	as (e.g., *c)*, *f]*, *p}*)
registers:	a (e.g., *f*,*t*)

Sample Extensions

Appendix Headings

Here is a method for 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 "*title*", which can be saved for the table of contents, if desired. To center appendix titles, set register *Hc* to 1.

Hanging Indent With Tabs

The following example illustrates the use of the hanging indent feature of variable-item lists. 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 set by *.ta* request and `\t` is interpreted as a tab key by formatters). Since the first argument can begin with a period or apostrophe, the "\&" is prepended to it so that the formatter will not interpret such a line as a formatter request or macro call.

Note

Formatters interpret the 2-character sequence "\&" as a "zero-width" space. This provides a method for removing the the special meaning of leading periods and apostrophes without affecting output text. Formatters translate `\t` into a tab, and `\c` is used to concatenate input text that follows the macro call to the line built by the macro.

Here is the macro and an example of its use:

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
...
  .
  .
  .
.ta 8 14 20 24
.VL
.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 line.
.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.

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 command sequences to produce documents. Reasonable macro argument default values are provided where possible.
- *Parameter based* – 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 meet their respective needs with a wide range of styles.
- *Extension by moderately expert users* – A strong effort has been expended 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 printing terminal using the *nroff* formatter. Macros can be conveniently used with both 10- and 12-pitch terminals. In addition, output can be displayed on an appropriate CRT trminal. Macros have been constructed to provide compatibilty with the *troff*(1) formatter so that output can be produced on phototypesetters and printing/CRT terminals.
- *Minimization of input* – The design of macros attempts to minimize repetitive typing. For example, to produce a blank line after all first- or second-level headings, only one parameter is required, and it is set at the beginning of the document; eliminating the need for a blank line after each heading affected.
- *Decoupling input format from output style* – There is but one way to prepare the input text although the user can 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 can be varied from document to document or within a single document.

Footnote Examples

Input Text File:

```
.P
.FD 10
This example illustrates several footnote styles
for both labelled 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 pemitted,
footnote text is not indented,
and the label is placed at the beginning of the first line.
.FE
```

NROFF Printed Output:

```
This example illustrates several footnote styles for both
labelled and automatically numbered footnotes.  With the
footnote style set to the NROFF default, process the first
footnote 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, 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.
```

---

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 nt 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 necessariy illustrated by this example, hyphenation is permitted.

\+ This is the fourth 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.

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.

Tables

This section contains tables of reference information including macro names, string and numeric registers, and error messages. Numbers inside braces ({ }) refer to the page number where the topic is explained in greater detail.

Table 1: Mm Macro Names Sumary

Macro	Description {Page} Command Syntax
1C	1-column processing {59} .1C
2C	2-column processing {59} .2C
AE	Abstract end {35} .AE
AF	Alternate format of "Subject/Date/From" block {37} .AF [company-name]
AL	Automatically incremented list start {23} .AL [type] [text-indent] [1]
AS	Abstract start {35} .AS [arg] [indent]
AT	Author's title {34} .AT [title]...
AU	Author information {34} .AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
AV	Approval signature {40} .AV [name]
B	Bold {58} .B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]
BE	Bottom block end {52} .BE
BI	Bold/Italic {58} .BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]
BL	Bullet list start {24} .BL [test-indent] [1]
BR	Bold/Roman {58} .BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]
BS	Bottom block start {52} .BS

Table 1: Mm Macro Names Summary (continued)

Macro	Description {Page} Command Syntax
CS	Cover sheet {56} .CS [pages] [other] [total] [figs] [tbls] [refs]
DE	Display end {41} .DE
DF	Display floating start {42} .DF [format] [fill] [right-indent]
DL	Dash list start {25} .DL [text-indent] [1]
DS	Display static start {41} .DS [format] [fill] [right-indent]
EC	Equation caption {46} .EC [title] [override] [flag]
EF	Even-page footer {50} .EF [arg]
EH	Even-page header {50} .EH [arg]
EN	End equation display {45} .EN
EQ	Equation display start {45} .EQ [label]
EX	Exhibit caption {46} .EX [title] [override] [flag]
FC	Formal closing {38} .FC [closing]
FD	Footnote default format {48} .FD [arg] [1]
FE	Footnote end {47} .FE
FG	Figure title {46} .FG [title] [override] [flag]
FS	Footnote start {47} .FS [label]
H	Heading – numbered {15} .H level [heading-text] [heading-suffix]
HC	Hyphenation character {10} .HC [hyphenation-indicator]
HM	Heading mark style {18} (Aragic or Roman numerals, or letters) .HM [arg1]...[arg7]
HU	Heading – unnumbered {19} .HU heading-text

Table 1: Mm Macro Names Summary (continued)

Macro	Description {Page} Command Syntax
HX[1]	Heading user-exit X (before printing heading) {20} .HX dlevel rlevel heading-text
HY[1]	Heading user-exit Y (before printing heading) {20} .HY dlevel rlevel heading-text
HZ[1]	Heading user-exit Z (after printing heading) {20} .HZ dlevel rlevel heading-text
I	Italic (underline in the *nroff* formatter) {58} .I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]
IB	Italic/Bold {58} .IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]
IR	Italic/Roman {58} .IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]
LB	List begin {29} .LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
LC	List-status clear {32} .LC [list-level]
LE	List end {28} .LE [1]
LI	List item {27} .LI [mark] [1]
ML	Marked list start {25} .ML mark [text-indent] [1]
MT	Memorandum type {36} .MT [type] [addressee] or .MT [4] [1]
ND	New date {37} .ND new-date
NE	Notation end {39} .NE
NS	Notation start {39} .NS [arg]
nP	Double-line indented paragraphs {15} .nP
OF	Odd-page footer {50} .OF [arg]
OH	Odd-page header {50} .OH [arg]
OK	Other keywords for the Technical Memorandum cover sheet {36} .OK [keyword]...

1 See note at end of table

Table 1: Macro Names Summary (continued)

Macro	Description {Page} Command Syntax
OP	Odd page {61} .OP
P	Paragraph {14} .P [type]
PF	Page footer {50} .PF [arg]
PH	Page header {50} .PH [arg]
PM	Proprietary marking {53} .PM [code]
PX[1]	Page-header user exit {52} .PX
R	Return to regular (Roman) font {58} .R
RB	Roman/Bold {58} .RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]
RD	Read insertion from terminal {63} .RD [prompt] [diversion] [string]
RF	Reference end {56} .RF
RI	Roman/Italic {58} .RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]
RL	Reference list start {25} .RL [text-indent] [1]
RP	Produce reference page {57} .RP [arg] [arg]
RS	Reference start {56} .RS [string-name]
S	Set *troff* formatter point size and vertical spacing {62} .S [size] [spacing]
SA	Set adjustment (right-margin justification) default {59} .SA [arg]
SG	Signature line {38} .SG [arg] [1]
SK	Skip pages {61} .SK [pages]

1 See note at end of table.

Table 1: Mm Macro Names Summary (continued)

Macro	Description {Page} Command Syntax
SM	Make a string one point-size smaller {62} .SM string1 [string2] [string3]
SP	Space vertically {61} .SP [lines]
TB	Table title {46} .TB [title] [override] [flag]
TC	Table of contents {54} .TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]
TE	Table end {44} .TE
TH	Table header {44} .TH [N]
TL	Title of memorandum {33} .TL [charging-case] [filing-case]
TM	Technical Memorandum number(s) {35} .TM [number]...
TP[1]	Top-of-page macro {52} .TP
TS	Table start {44} .TS [H]
TX[1]	Table of contents user exit {55} .TX
TY[1]	Table of contents user exit {55} (suppresses "CONTENTS") .TY
VL	Variable-item list start {25} .VL text-indent [mark-indent] [1]
VM	Vertical margins {53} .VM [top] [bottom]
WC	Footnote and Display Width control {60} .WC [format]

1 Macros referencing this footnote are not generally called (invoked) by users. They are "user exits" defined by the user and called by MM from within header, footer, or other macros.

Table 2: String Names Summary

String Name	Description {page}
BU	Bullet {11}
Ci	Table of contents indent list {54} Up to seven args (must be scaled) for heading levels 1 through 7
DT	Date {37} Current date (unless overridden) Month, Day, Year (e.g., July 16, 1986)
EM	Em dash string {12} Produces an em dash in the *troff* formatter and a double hyphen in *nroff*
F	Footnote numberer { } *NROFF*: `\u\\n+(:P\d` *TROFF*: `\v'-.4m'\s-3\\n+(:*P\s0\v'.4m'`
HF	Heading font list {17} Up to seven codes for heading levels 1 through 7 Defaults: 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 {18} Up to seven codes for heading levels 1 through 7
Le	Title for LIST OF EQUATIONS {46}
Lf	Title for LIST OF FIGURES {46}
Lt	Title for LIST OF TABLES {46}
Lx	Title for LIST OF EXHIBITS {46}
RE	SCCS Release and Level of MM {59} Release.Level (e.g., 10.129)
Rf	Reference numberer {56}
Rp	Title for references {57}
Tm	Trademark string {12} Places the letters "TM" one-half line above the text that follows. Seven accent strings are also available {63}

Note 1: If the released-paper style is used, then, in addition to the above strings, certain Bell Telephone Laboratories location codes are defined as strings; these location strings are not used after the .MT macro is called {36}. 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: See page 3 for notes on setting and referencing strings.

Table 3: Numeric Register Names Summary

Register	Description {Page} Default, [Allowable-value range]
A[2]	Handles preprinted forms and Bell System logo {7} 0, [0:2]
Au	Inhibits printing author's location, department, room, and extension in "from" portion of memorandum {34}
C[2]	Copy type {7} Original, Draft, etc. 0 (Original), [0:4]
C1	Contents level {19} Level of headings saved for table of contents 2, [0:7]
Cp	Placement of list of figures, etc. {55} 1 (on separate pages), [0:1]
D[2]	Debug flag {7} 0,[0:1]
De	Display eject register for floating displays {43} 0, [0:1]
Df	Display format register for floating display {43} 5, [0:5]
Ds	Static display pre- and post-space {42} 1, [0:1]
E[2]	Controls font of the Subject/Date/From fields {7} 1 (*nroff*) 0 (*troff*), [0:1]
Ec	Equation counter, used by .EC macro {46} 0, [0:?], incremented by one for each .EC call.
Ej	Page-ejection flag for headings {16} 0 (no eject), [0:7]
Eq	Equation label placement {45} 0 (right-adjusted), [0:1]
Ex	Exhibit counter, used by .EX macro {46} 0, [0:?], incremented by one for each .EX call.
Fg	Figure counter, used by .FG macro {46} 0, [0:?], incremented by one for each .FG call.
Fs	Footnote space (i.e., spacing between footnotes) {49} 1, [0:?]
H1 through H7	Heading counters for levels 1 through 7 {18} 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.

2 See notes at end of table.

Table 3: Numeric Register Names Summary

Register	Description {Page} Default, [Allowable-value range]
Hb	Heading break level (after .H and .HU) {16}
Hc	Heading centering level for .H and .HU {17} 0 (no centered headings), [0:7]
Hi	Heading temporary indent (after .H and .HU) {16} 1 (indent as paragraph), [0:2]
Hs	Heading space level (after .H and .HU) {16} 2 (space only after .H 1 and .H 2), [0:7]
Ht	Heading type {19} For .H: single or concatenated numbers 0 (concatenated numbers: 1.1.1, etc.), [0:1]
Hu	Heading level for unnumbered heading (.HU) {19} 2 (.HU at the same level as .H 2, [0:7]
Hy	Hyphenation control for body of document {10} 0 (atuomatic hyphenation off), [0:1]
L[2]	Length of page {7} 66, [20:?] (11i, [2i:?] in *troff* formatter) For *nroff*, these values are unscaled numbers representing lines or character positions; for *troff*, values must be scaled.
Le	List of equations {46} 0 (list not produced) [0:1]
Lf	List of figures {46} 1 (list not produced) [0:1]
Li	List indent {24} 6 (nroff) 5 (troff), [0:?]
Ls	List spacing between items by level {24} 6 (spacing between all levels) [0:6]
Lt	List of tables {46} 1 (list produced) [0:1]
LX	List of exhibits {46} 1 (list produced) [0:1]
N[2]	Numbering style {7} 0, [0:5]
Np	Numbering style for paragraphs {15} 0 (unnumbered) [0:1]

[2] See notes at end of table.

Table 3: Numeric Register Names Summary

Register	Description {Page} Default, [Allowable-value range]
O[2]	Offset of page {8} .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 {55} 0 (lowercase Roman), [0:1]
Of	Figure caption style {46} 0 (period separator), [0:1]
P[2]	Page number manager by MM {8} 0, [0:?]
Pi	Paragraph indent {14} 5 (*nroff*) 3 (*troff*), [0:?]
Ps	Paragraph spacing {15} 1 (one blank space between paragraphs), [0:?]
Pt	Paragraph type {14} 0 (paragraphs always left justified), [0:2]
Pv	"PRIVATE" header {53} 0 (not printed), [0:2]
Rf	Reference counter, used by .RS macro {56} 0, [0:?], incremented by one for each .RS call.
S*[2]	The *troff* formatter default point size {8} 10, [6:36]
Si	Standard indent for displays {42} 5 (*nroff*) 3 (*troff*), [0:?]
T*[2]	Type of *nroff* output device {8} 0, [0:2]
Tb	Table counter, used by .TB macro {46} 0, [0:?], incremented by one for each .TB call.
U*[2]	Underlining style (*nroff*) for .H and .HU {8} 0 (continuous underline when possible), [0:1]
W*[2]	Width of page (line and title length) {8} 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 {6, 7, 8}.

2 Page 3 has notes on setting and referencing registers. Any register having a single-character name can be set from the command line.

Table 4: Error Messages

An MM error message has a standard part followed by a variable part. The standard part has the form:

```
ERROR: (filename) input line n:
```

Variable parts consist of a desciptive message usually beginning with a macro name. They are listed here in alphabetical order by macro name, each with a more complete explanation.

Mm Error Messages

Error Message	Description
Check TL, AU, AS, AE, MT sequence	The correct order of macros at the start of a memorandum is shown in {..}. 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 {..}. Something has disturbed this order. Occurs if the .AS 2 {..} 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 { }.
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 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: <value>	The first argument to the .H macro must be a single digit from one to seven, but <value> 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.
H:missing FE	A heading macro (.H or .JU) occurs inside a footnote.

Table 4: Error Messages (continued)

***Mm* Error Messages**

Error Message	Description
HU:missing arg	The .JU 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 {28}. 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 closing .RF.
RS:missing RF	A previous .RS macro was not matched by a closing .RF.
S:bad arg: <value>	The incorrect argument <value> has been given for the .S macro {..}.
SA:bad arg:<value>	The argument to the .SA macro (if any) must be either 0 or 1. the incorrect argument is shown as <value>.
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 {60}.

Formatter Error Messages

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.

Error Message	Description
Cannot do ev	Caused by: 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: Error Messages (continued)

Error Message	Description
Cannot execute <filename>	Given by the .! request if the <filename> is not found.
Cannot open <filename>	Indicates one of the file 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 (vis .hw 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 generate through the misuse of \c of the .cu request or very long equations produced by *eqn(1)/ neqn.*
Non existent 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 .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.
Too many page numbers	Indicates the list of pages specified to the -o 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 .rr 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 .rm 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 \c of the .cu request, or very long equations produced by *eqn (1)/neqn.*

Table of Contents

Tbl
A Program to Format Tables

Tbl is an HP-UX document formatting command that is used as a preprocessor for *troff* or *nroff* formatters, making even fairly complex tables easy to specify and enter.

Tables are made up of columns which can be independently centered, right-adjusted, left-adjusted, or aligned by decimal point location. Headings can be placed over single columns or groups of columns. A table entry can contain equations, and can accommodate multiple rows of text. Horizontal or vertical lines can be drawn as desired in the table, and any table or element can be enclosed in a box.

Tbl takes input text containing *tbl* commands and data, processes it, and produces output that is compatible with *troff* and *nroff* formatters. If the source contains commands and data for other formatters, *tbl* processes only those parts of the job that can be successfully handled, leaving the remainder for other commands and programs supported on HP-UX.

Usage

To use *tbl* for constructing a simple table, execute the following HP-UX command:

`tbl` *filename* `|troff` **RETURN** (or (ENTER))

For more complex situations involving multiple files and/or *ms* or *mm* macro requests as well as tables, the normal command is:

`tbl` *file1 file2* `...troff-ms` **RETURN** (or (ENTER))

The usual options can be used on *troff*. *Nroff* is similar to *troff*, but many display and printing devices driven by *nroff* cannot reproduce boxed tables. If a file name is "-", the standard input device is used at that point.

For line printers that do not have adequate driving tables or post-filters, the special *−TX* command-line option to *tbl* produces output having no fractional line motions. The only other command-line options recognized by *tbl* are *−ms* and *−mm*. They are converted into commands to fetch the corresponding macro files. While it is usually more convenient to place these arguments on the *troff* formatter part of the command line, they are also accepted by *tbl*.

Whenever *eqn* and *tbl* are used together on the same file, *tbl* should be used first. If there are no equations within the table(s), either sequence works, but, since *eqn* usually produces a larger expansion of the input, it is usually faster to execute *tbl* first. On the other hand, if the table contains equations, *tbl* **must** be used first to prevent scrambling the output. Avoid using equations in *n*-style columns because *tbl* attempts to split numerical format items into two parts. The *delim*<xy> table option prevents splitting numerical columns between delimiters. For example, if the *eqn* delimiters are `$$` (set by `delim $$`), a numerical column such as 1245 $\pm$ 16$ is divided after 1245, not after 16.

Tbl accepts up to 35 columns, but the actual number that can be processed may be less, depending on the availability of *troff* formatter number registers. Number register names used by *tbl* **must** be avoided in tables. These include 2-digit numbers from 31 thru 99 and strings of the form 4x, 5x, *x, x+, x|, ^x, and x-, where x is any lowercase letter. The names **, *-, and *^ are also used in certain circumstances. To conserve register names, the *n* and *a* key-letters (key-letters are discussed in "Format Section" later in this tutorial) share a register; hence, the restriction that they cannot be used in the same column.

As an aid in writing layout macros, *tbl* defines a number register TW which is the table width. The TW number register is defined before the *.TE* macro is invoked, and can be used in the *.TE* expansion.

Another important macro *T#* defines and produces the bottom lines and side lines of a boxed table when it is invoked at the end of a table or page. Including this macro in the page footer enables you to create multi-page tables with headers repeated on each successive page in the table by using the H argument with the *.TS* macro. If the `table start` macro is written as follows:

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if no heading exists). Material preceding the .TH is placed at the top of each page of the table. The remainder of the table is placed on one or more pages, as needed. This feature is provided by the *ms* and *mm* macros, not by *tbl*.

Input Commands

The input to *tbl* is text for a document, where tables are preceded by *.TS* (table start command) and followed by *.TE* (table end command). *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The *.TS* and *.TE* lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the *.TS* or *.TE* lines are copied but otherwise ignored, and can be used by document-layout macro commands.

The format of the input is as follows:

```
text
.TS
table
.TE
text
.TS
table
.TE
text
. . .
```

where the format of each *table* is:

```
.TS
options;
format.
data
.TE
```

Each table structure is independent, and contains:

- Global options,
- A format section describing individual columns and rows in the table,
- Data to be printed in the table

The format section and data are required, but the options list can be omitted.

Global Options

A single line of options affecting the entire table can be placed immediately following the `.TS` line. It must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. Recognized options include:

center	Center the table (default is left-adjust).
expand	Make the table as wide as the current line length.
box	Enclose the table in a box.
allbox	Enclose each table item in a box.
doublebox	Enclose the table in two boxes.
tab <x>	Use <x> instead of tab to separate data items where <x> is any character.
linesize <n>	Set lines or rules (e.g., from *box*) in <n> point type.
delim <xy>	Recognize <x> and <y> as the *eqn* delimiters (<x> and <y> are any character).

Tbl tries to keep boxed tables on one page by issuing appropriate *.ne* (need) requests. These requests are calculated from the number of lines in the tables. If there are spacing commands embedded in the input, the *.ne* requests may be inaccurate, in which case, normal *troff* procedures, such as keep-release macros should be used. If you need a multi-page boxed table, use *.TS H* and *.TH* macros which are designed for this purpose.

Format Section

The format section of the table specifies column layout. Each line in the format section corresponds to one line of table data (except that the last format line corresponds to all the following data lines up to the next *.T&* command line. Each line contains a **key-letter** for each column of the table. **Key-letters** for each column can be separated by spaces or tabs for better readability. Allowable key letters include:

L or *l*	Indicates a left-adjusted column entry.
R or *r*	Indicates a right-adjusted column entry.
C or *c*	Indicates a centered column entry.
N or *n*	Indicates a numerical column entry to be aligned with other numerical entries so that the units digits of numbers line up.
A or *a*	Indicates an alphabetic subcolumn where all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example later in this tutorial).

S or *s* — Indicates a spanned heading; i.e., the previous-column continues across this column (not allowed for first column).

^ — Indicates a vertically spanned heading; i.e., the previous-row entry continues down through this row (not allowed for first row).

When numerical column alignment (*n*) is specified, a location for the decimal point is sought. The right-most dot (.) adjacent to a digit is used as a decimal point unless there is no dot adjoining a digit, in which case the rightmost digit is used as a units digit. If no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& can be used to unconditionally override dots and digits or align alphabetic data (strings). When included in a string, the \& is positioned such that the & is placed in the location that would otherwise be occupied by a dot, but the \& is suppressed before sending the final output.

In the following example, input items on the left are aligned on the right as they would be in a numerical table column.

```
13                          13
4.2                          4.2
26.4.12                    26.4.12
abcdefg                    abcdefg
abcdefg\&                abcdefg
43\&3.22                     433.22
749.12                       749.12
```

Note

When numerical data is found in the same column with wider *L*- or *r*-type table entries, the widest <number> is centered relative to the wider *L* or *r* items (*L* is used instead of *l* for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of *a* type data, as previously explained. However, alphabetic subcolumns (requested by the *a* key-letter) are always slightly indented relative to *L* items (if necessary, the column width is increased to force this). This is not true for *n*-type entries.

Do not use *n* and *a* items in the same column.

For readability, separate the key-letters describing each column. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus, a simple 3-column format might appear as:

```
css
lnn.
```

The first line specifies a heading centered across all three columns. Each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. Here is an example table using such a format:

```
c s s
l n n.
Overall title
Item a
34.22
9.1
Item b
12.65
.02
Items: c,d,e
23
5.8
Total
69.87
14.92
```

Here are some additional features of the key-letter system:

Horizontal lines

A key-letter can be replaced by underscore (_) to indicate a horizontal line in place of the corresponding column entry, or by equals (=) to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining the current column, the horizontal line is extended to meet both nearby lines. If any data is provided for this column of the current table, it is ignored and an error message is printed.

Vertical lines

A vertical bar can be placed between column key-letters, producing a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter of the current line and/or to the right of the last key-letter of the current line produces a line at the corresponding edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns

A number can follow the key-letter, specifying the amount of separation between the current column and the next column. The number normally specifies the separation in *en* spaces where one *en* is about the width of the letter "n" (more precisely, the width of an en space is equal to half the number of points (1 point = 1/72 inch) in the current type size).

If the **expand** option is used, these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed, worst case (largest space requested) governs.

Vertical spanning

Vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by `t` or `T`, any corresponding vertically spanned item will begin at the top line of its range.

Font changes

A key-letter followed by a string containing a font name or number preceded by the letter *f* or *F* indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters *B*, *b*, *I*, and *i* are shorter synonyms for *fB* and *fI*. Font change commands given with the table entries override these specifications.

Point size changes

A key-letter followed by `p` or `P` and a number specifies the point size of corresponding table entries. If the number is a signed digit, it is taken as an increment or decrement from the current point size. If point size and column separation are both specified, one or more blanks must separate the two parameters.

Vertical spacing changes

A key-letter followed by *v* or *V* and a number specifies the vertical line spacing to be used within a multi-line corresponding table entry. If the number is a signed digit, it is taken as an increment or decrement from the current vertical spacing. If column separation and vertical spacing are both specified, the two parameters must be separated by one or more blanks or by some other specification. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication

A key-letter followed by the letter *w* or *W* and a width value in parentheses specifies minimum column width. If the largest element in the column is not as wide as the width value given after the *w*, the specified width is used. If the largest element in the column is wider than the specified value, the element width is used. Column width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value (if units are not specified, the default is *ens*. If the width specification is a unitless integer, the parentheses can be omitted. If the column width specification is changed during table construction, the **last** value encountered determines the result.

Equal-width columns

A key-letter followed by the letter *e* or *E* indicates equal-width columns. All columns whose key-letters are followed by *e* or *E* are made the same width, permitting the user to get a group of regularly spaced columns.

Staggered columns

A key-letter followed by *u* or *U* indicates that the corresponding entry is to be moved up one-half line. This makes it easy to have a column of differences between numbers in an adjoining column. The *allbox* option does not work with staggered columns.

Zero-width item

A key-letter followed by *z* or *Z* indicates that the corresponding data item is to be ignored in calculating column widths. This is sometimes useful when allowing headings to run across adjacent columns where spanned headings would be inappropriate.

Default

Column descriptors missing from the end of a format line are assumed to be *L* and the longest line in the format section defines the number of columns in the table. Data that would normally appear in extra columns is ignored.

The chronological order of the above features is immaterial; they need not be separated by spaces (except where indicated to avoid ambiguities involving point size and font changes). As an example, a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as:

```
np12w(2.5i)fI 6
```

Data to be Printed

Data for the table follows the format commands. Normally, each table input line (one or more column entries) is typed as one line of data. Very long input lines can be broken by placing a backslash character (\) at the end of the line (when successive lines are combined, the \ vanishes). Data items for successive columns on the same line are separated by tabs, or by whatever character has been specified in the *tab* global option. Here are some special cases of interest:

***Troff* commands within tables**

An input line beginning with a dot (.) followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. For example, space within a table can be produced by .sp requests in the data.

Full-width horizontal lines

An input line containing only the character underscore (_) or equals sign (=) results in a single or double line, respectively, extending the full width of the table.

Single-column horizontal lines

An input table entry containing only the character underscore (_) or equal sign (=) results in a single or double line extending the full width of the column. Such lines are extended to meet horizontal or vertical lines adjoining the column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline character.

Short horizontal lines

An input table entry containing only the string _ results in a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters

An input table entry containing only a string of the form \R<x> (where <x> is any character) is replaced by repetitions of the character <x> as wide as the data in the column. The sequence of <x>s is not extended to meet adjoining columns.

Vertically spanned items

An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter ^.

Text blocks

To include a block of text as a table entry, precede it by `T{` and follow it by `T}`. Thus the sequence

```
. . . T{
block of
text
T} . . .
```

places, as a single entry in the table, something that cannot be conveniently typed as a simple string between tabs. Note that the `T}` end delimiter must begin a line, but additional columns of data can follow after a tab on the same line.

If more than twenty or thirty text blocks are used in a table, various limits in *troff* are likely to be exceeded, producing diagnostics such as "too many string/macro names" or "too many number registers".

Text blocks are removed from the table, processed separately by *troff*, then replaced in the table as a solid block. If no line length is specified in the block of text itself or in the table format, the default is calculated using:

$$L \times C \div (N+1)$$

where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table.

Other parameters (point size, font, etc.) used in setting the block of text are:

a) Those in effect at the beginning of the table (including the effect of the *.TS* macro),

b) Any table format specifications of size, spacing and font that use *p*, *v*, and *f* modifiers to the column key-letters, and

c) *Troff* requests within the text block itself are also recognized. However, *troff* commands within the table data but not within the text block do not affect that block.

Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table can be arranged as several single-page tables if this proves to be a problem.

Other difficulties with formatting may arise because in the calculation of column widths, all table entries are assumed to be in the font and size being used when the *.TS* command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry `\s+3 data\fP\s0`). *Troff* requests can be sprinkled in a table, but care must be taken to avoid confusing the width calculations. Once the table structure is started, it is not possible to change the number of columns, space between columns, global options such as *box*, or to set the columns equal in width.

Additional Command Lines

To change table format after many similar lines (as with sub-headings or summarizations) use the *.T&* (table continue) command to alter column parameters. Here is an outline of the general technique:

```
.TS
options;
format.
data
. . .
.T&
format.
data
.T&
format.
data
.TE
```

This technique enables you to keep each table line close to its corresponding format line.

Examples

The following example tables show various techniques and command options. While each table is structured for illustrating a particular concept, other options are also illustrated such as font changes, for example.

Table Using "box" Option

Input:

```
.TS
box;
ccc
lll.
Language⇄ Authors⇄ Runs on
.sp
Fortran⇄ Many⇄ Almost anything
Pl/1⇄ IBM⇄ 360/370
C⇄ AT&T⇄ 11/45,H6000,370
BLISS⇄ Carnegie-Mellon⇄ PDP-10,11
IDS⇄ Honeywell⇄ H6000
Pascal⇄ StanfordM370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	AT&T	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Table Using "allbox" Option

Input:

```
.TS
allbox;
c s s
c c c
n n n.
Common Stock
Year◊ Price◊ Dividend
1971◊ 41-54◊ $2.60
2◊ 46-55◊ 2.70
3◊ 46-55◊ 2.87
4◊ 40-53◊ 3.24
5◊ 45-52◊ 3.40
6◊ 51-59◊ .95*
.TE
*(first quarter only)
```

Output:

Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	46-55	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

*(first quarter only)

Table Using "vertical bar" Key-letter Feature

Input:

```
.TS
box;
c s s
c| c| c
l| l| n.
Major New York Bridges
_
Bridge⇕ Designer⇕ Length
_
Brooklyn⇕ J.A. Roebling⇕ 1595
Manhattan⇕ G. Lindenthal⇕ 1470
Williamsburg⇕ L.L. Buck⇕ 1600
_
Queensborough⇕ Palmer &⇕ 1182
⇕ Hornbostel
_
⇕ ⇕ 1380
Triborough⇕ O.H. Ammann⇕ _
⇕ ⇕ 383
_
Bronx Whitestone⇕ O.H. Ammann⇕ 2300
Throgs Neck⇕ O.H. Ammann⇕ 1800
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn Manhattan Williamsburg	J.A. Roebling G.Lindenthal L.L. Buck	1595 1470 1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O.H. Ammann	1380 383
Bronx Whitestone Throgs Neck	O.H. Ammann O.H. Ammann	2300 1800

Table Using Horizontal Lines in Place of Key-letters

Input:

```
.TS
box;
L L L
L L _
L L| LB
L L _
L L L.
january\t february\t march
 april\t may
june\t july\t Months
august\t september
october\t november\t december
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Table Using Additional Command Lines

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c |c s s
c | c s s
c | c | c | c
Food⇄ Percent by Weight
\^⇄ _
\^⇄ Protein⇄ Fat⇄ Carbo-
\^⇄ \^⇄\^⇄ hydrate
_
.T&
l | n | n | n.
Apples⇄ .4⇄ .5⇄ 13.0
Halibut⇄ 18.4⇄ 5.2⇄ . . .
Lima Beans⇄ 7.5⇄ .8⇄ 22.0
Milk⇄ 3.3⇄ 4.0⇄ 5.0
Mushrooms⇄ 3.5⇄ .4⇄ 6.0
Rye Bread⇄ 9.0⇄ .6⇄ 52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	. . .λ
Lima Beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Table Using Text Blocks

Input:

```
.TS
allbox;
cfI s s
c cw(2i) cw(2i)
l l l.
New York Area Rocks
.sp
Era\t Formation\t Age (years)
Precambrian\t Reading Prong\t >1 billion
Paleozoic\t Manhattan Prong\t 400 million
Mesozoic\tT{
.na
Newark Basin, incl.
 Stockton, Lockatong, and Brunswick
 formations
.ad
T}\t 200 million
Cenozoic\t Coastal Plain\t T{
.na
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation
.ad
T}
.TE
```

Output:

New York Area Rocks

Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick Formations	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation

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.

Part No. 97089-90004
E 0784
Microfiche No. 97089-99004

Printed in U.S.A.
First Edition, July 1984