TEK SYSTEM MANUAL

Part No. 070-5318-00
Product Group 65

UTek

TOOLS

VOLUME 1

First Printing NOV 1984

Tektronix®
COMMITTED TO EXCELLENCE

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development:

W. N. Joy
O. Babaouglu
R. S. Fabry
K. Sklower
Eric P. Allman
M. K. McKusick
E.Cooper
David Musher
S. J. Leffler

University of California at Berkeley
Department of Electrical Engineering and Computer Science

The MH Mail System is based on software developed by the Rand Corporation.

Portions of this document are based on the RCS Revision Control System, © 1982 Walter F. Tichy.

This documentation is for the use of our customers, and not for general sale.

Specifications subject to change.

Revision
INFORMATION

PRODUCT: 6000 Family UTek Operating System: 64WP02, 64WP05, 64WP06
This manual supports the following versions of this product: V2.0

REV DATE	DESCRIPTION
NOV 1984	Original Issue

Contents

Section 2B An Introduction to the Shell (cont) Page

Section 2C Introduction to the C-Shell

Section 2C Introduction to the C-Shell (cont) Page

Section 2D The UTek System Implementation

Section 2D The UTek System Implementation (cont) Page

Section 2E The UTek Fast File System

Section 2F The Distributed File System

Section 2G MDQS — The Multi-Device Queueing System

Section 3A Introduction to Editing Documentation

Section 3B Advanced Uses of Ed

Section 3B Advanced Uses of Ed (cont) Page

Section 3C Advanced Uses of Ex and Vi

Section 4A Nroff/Troff Tutorial

Section 4A Nroff/Troff Tutorial (cont) Page

Section 4B Nroff/Troff Reference Guide

Section 4B Nroff/Troff Reference Guide (cont) Page

Section 4C The MS Text-Formatting Macros

Section 4D The MM Text-formatting Macros

Section 4D The MM Text-formatting Macros (cont) Page

Section 4D The MM Text-formatting Macros (cont) Page

Section 4E The ME Reference Guide

Section 4F The ME Text-formatting Macros

Section 4G Tbl — A Table Formatting Program

Figures

Examples

Tables

1A

Introduction

UTek is the operating system that runs on your Tektronix 6000 Series Workstation. The operating system provides a software environment that supports your engineering applications. It also provides tools that let you develop programs for the workstation and accomplish everyday tasks such as sending electronic mail, editing and formatting text, and organizing information. UTek is a multi-user operating system, that can run several processes at once.

UTek is a UNIX-based operating system. It is based on 4.2bsd UNIX, with some features of UNIX System V. In addition, Tektronix has added many features to the operating system, including:

- virtual memory
- Multi-Device Queueing System
- Distributed File System
- Local Area Network support
- MH Mail System

UTek is intended for use by a wide variety of users, from the professional software developer to someone who does text processing. The operating system provides programming support for C, Pascal, and Fortran, including many language preprocessors.

The Tektronix family of workstations includes the 6130 workstations and the 6200 Series workstations. Any workstation in this family can communicate and share resources with any other member of the family. The UTek operating system runs on both the 6100 and the 6200 Series workstations.

Facilities Provided by UTek

The following attempts to highlight some of the major facilities provided by UTek. These major facilities include:

- electronic mail
- the shell command interpreter
- text editors
- text formatting
- programming
- programming support

electronic mail
The MH mail handling system lets you send messages to and receive messages from other users in your network. It also provides a way of filing messages for easy retrieval.

the shell
The shell interprets commands that you enter into your terminal and is itself a programming language. It provides variables that let you customize the way other commands on the system work. It lets you string several commands together, and redirect the input and output of commands.

text editors
Three text editors are available in UTek — **ed**, **ex**, and **vi**. **Ed** and **ex** are line-oriented editors, while **vi** is a visual editor that lets you move around the screen very easily.

text formatters
UTek provides several text formatters, depending on the kinds of documents you want to produce. All the UTek text formatters are macros based on the basic formatters **nroff** and **troff**. **Nroff** produces lineprinter or letter-quality printer output, while **troff** produces typeset output.

programming
UTek supports three programming languages: C, Pascal, and Fortran. In addition to these programming languages, UTek provides a number of programming preprocessors.

programming support
UTek provides tools to help you organize programming projects, including the RCS Revision Control System to keep track of changes to source code, and the **make** utility, which defines what portions of large programming projects are interdependent.

Depending on the model number of your workstation, and the software packages that you purchase, you may not have all the features included in the UTek Operating System.

About This Manual

Before you read this manual, it is recommended that you complete the Online Learning Sessions of the *Learning Guide*.

Another source of information included in your documentation set is the book *Introducing the UNIX System*, by McGilton and Morgan. Before reading this manual become familiar with chapters 1–4, 11, and 13 of *Introducing the UNIX System*.

This manual tells you how to use the major tools provided by UTek. It is intended to serve as both an initial learning guide and a reference manual. For complete information on every command available in the UTek Operating System, see the *UTek Command Reference*.

This manual is organized in seven parts: Introduction, Common Tools, Editing Tools, Text Formatting Tools, Programming Tools, Programming Support Tools, and Utilities. Within each major part of the book are sections that describe each tool. Following is a list of all the tools described in these major parts:

Introduction
This portion explains the features of the UTek system, and how to use the manual.

Common Tools
Topics discussed include: The MH Mail System, the Bourne Shell, the C-Shell, UTek System Implementation, UTek Fast File System, UTek Distributed File system, and the Multi-Device Queueing System.

Editing Tools
Topics discussed include: Advanced Uses of **ed** and Advanced Uses of **ex** and **vi.**

Text Formatting Tools
Topics discussed include: An **Nroff/Troff** Tutorial, **Nroff/Troff** Reference Guide, The **ms** Text Formatting Macros, The **mm** Text Formatting Macros, The **me** Text Formatting Macros, The **me** Reference Manual, and The Table Formatting Program — **tbl**.

Programming Tools
Topics discussed here include: **lint, yacc, curses**, the **f77** Fortran compiler, **ratfor**, Using Pascal on UTek, **lex, m4**, and **efl**. This part also includes a discussion of the debuggers **adb** and **sdb**.

Programming Support Tools
Topics discussed here include: RCS, a Revision Control System, Using **make**, Using RCS and **make** together, and the **awk** programming language.

Utilities
Topics discussed here include: An Interactive Desk Calculator, **dc**, and An Arbitrary Precision Calculator Language, **bc**.

Notation Conventions

The notation conventions listed below are used throughout this manual.

<RETURN>	Special keys are shown as all capital letters, surrounded by angle brackets.
<CTRL-X>	Control characters are shown using the same notation as for special keys. Control characters are created by holding down the key labeled *CONTROL* (or *CTRL* on some keyboards) while typing the indicated key (in this example, x).
file	Filenames, directory names, pathnames, and text for which you substitute your own information when entering a command are in *italics*.
cd	Command names and text you enter exactly as it appears are in **boldface**.
grep(1)	A command followed by a parenthesized number, *command(n)*, is a reference to more information on that command in the *UTek Command Reference — Section n*.

Related Documents

The following books and manuals are available from Tektronix, Inc. to help you use your workstation. Some of these documents came packaged with your workstation.

UTek Manuals

- *Introducing the UNIX System*
  McGilton and Morgan
  (McGraw–Hill Book Company)
- *UTek Command Reference* Manual
- *UTek Tools*

Programming Language Books

- *The C Programming Language*
  Kernighan and Ritchie
  (Prentice–Hall)
- *Pascal User Manual and Report*
  Jensen and Wirth
- *Tektronix ANSI BASIC Keyword Dictionary*
- *Tektronix ANSI BASIC Learning Guide*

Workstation User Manuals

- *Learning Guide*
- *System Administration*

Graphics Package Manuals

- *GKS C*
- *GKS Fortran*

Workstation Service Manuals

- *Service*
- *Diagnostics*

2A

The MH Mail System

Introduction

The UTek MH (mail handling) utility lets you send mail to, and receive mail from, other users on the system, including users on remote workstations connected by a Local Area Network. This section provides a brief introduction to MH and tells you where to find detailed information on each MH command.

Overview

A mail message has two major pieces: the header and the body.

- The header is composed of several components (see *mh – mail(1MH)* for a complete list of components). The default components are **To, Cc,** and **Subject.**
- The body consists of the text of the message. The body of the message is fre formatted, but must not include graphic or binary data.

The body follows the header and is separated from it by an empty line. When you compose a message, the form that appears on your terminal may show a line of dashes after the header. This line is replaced by an empty line when the message i sent.

Setting Up Your Mail System

To use the MH system, you need a *Mail* directory and an *.mh_profile* file in your login directory. To set these up, enter the following sequence of commands:

Type **comp** (as with any command, followed by a <RETURN>). The system responds with the following question:

> Do you want the recommended MH path "usr/*login-name*/Mail"?

Type **yes** (or **y**). The system responds with another question:

> "Mail" doesn't exist; Create it?

Again, type **yes** (or **y**).

To complete setting up the files in the *Mail* directory, compose the following mail message to yourself as a test.

To:*login-name* <RETURN>
Cc: <RETURN>
Subject:*Test* <RETURN>

This is a test. <RETURN>
<ESC>
ZZ

The words **To**, **Cc**, and **Subject** are prompts. In place of *login-name*, type your login name.

The system next asks:

What now?

Type **send** (or **s**). Before a new prompt appears, the message *You have mail* displays. If your system is running slowly, it may take a while longer to receive this test message. After the prompt, type **inc** or **mail** to incorporate your mail into your *inbox*. The system responds with the following question:

Create folder "usr/*login-name*/Mail/inbox"?

Type **y** to create an *inbox* to receive your mail. After you see a prompt, type **show**. The mail message you composed should now appear on your terminal screen.

In general, any time MH needs a response, you can abbreviate the value to just enough characters to make it distinct from all the other possibilities. For example, **send** can be abbreviated **s**. Also, any time MH asks for a response, you can press <RETURN> to see a list of possible responses.

Sending Mail

The **comp** (compose) command allows you to create, edit, and send a message. **Comp** prompts for the header components:

To:	Enter the login–name of the person(s) you're writing to
Cc:	Enter the login–name of the person(s) to send a copy to
Subject:	Enter the subject of the message

You must enter something for the **To**, but you can leave the **Cc** or **Subject** component blank by pressing <RETURN>. The text of a component may take more than one line, but each continuation line must start with a blank or tab.

Comp types a row of dashes after the header. After the dashes, enter the body of your mail message. The only editing allowed is backspacing.

To end the message, press <ESC> or <CTRL-D>. The system displays another row of dashes, followed by the question **what now?**.

Press <RETURN> to get a list of the options available and their functions.

- If you respond with **send**, the message is saved in *$HOME/Mail/,draft*, sent, and **comp** exits.
- If you **quit** without sending the message, the message is saved in *$HOME/Mail/draft* You can retrieve this draft message by entering **comp -use**.
- If you respond with **edit**<*editor*>, you can edit the message you just entered, with the named editor.
- If you respond with **list**, the message is displayed on the terminal.

Reading Your Mail

When another user has sent you a letter, the message **you have mail** will appear on your terminal screen. Type **inc** (for incorporate).

The **inc** command takes all mail that has been sent to you since the last time you executed the command, numbers each message, and puts the messages into your inbox directory.

Inc also displays a **scan** listing for the new messages, for example:

```
$inc
7+    7/13    Cas       revival of measurement
8     10/9    Norm      NBS people and publications
9     11/26   To:norm    question<<Are there any function
```

The following explains a **scan** listing:

- The first column in the **scan** listing is the message number. The plus sign (7+) indicates the current message.
- The second column is the date the message was sent.
- The third column indicates who sent the message. If you sent yourself a mail message (by including your login-name in the **To** or **Cc** fields), then the **To:** component is displayed in the **scan** listing. For example, **To:norm** indicates that the message went to norm, but that you also sent a copy to yourself.
- The last column is the subject of the message. If the subject is short, the first part of the body of the message is included after the << characters.

Once the mail has been incorporated, you can use the **show** command to copy, print, or display a message. For example:

show	Displays the current message.
show 7	Displays message 7.
show > filename	Copies the current message to file *filename*.
show ¦ lpr	Prints the current message on the line printer.
show next	Displays the message that follows the current one.
show prev	Displays the message previous to the current one.
show last	Displays the last (highest numbered) message.
show all	Displays all messages.
show first–last	Same as show all.

You can do a **scan** at any time to see all of the messages in a folder.

Sending a Reply

The **repl** (reply) command sets up the header of a reply message. The information used to formulate the reply header is obtained from the original message's header. The message being answered is the current message if you do not specify a message number, or **n** if you do specify a number.

After the header is completed, you can finish the body of the message as in **comp**. For example:

```
$repl
To:johnd
Cc:sues
Subject:Re:Staff Meeting
In–reply–to:Your message of 28 Sept 1983 at 1420–PDT (Wednesday)
————————
Thanks for the reminder. I'll be there.
```

Sending Mail to Users on a Remote Workstation

If you have your workstations connected in a Local Area Network, you can send mail to users on remote workstations. If the user to whom you want to send mail does not have an account on your current workstation, you need to know the name of their remote workstation. Then you enter that name, separated from their login–name by the @ symbol, on the **To:** line of the message. For example, to send a message to the login–name john, on the workstation called tiger, enter:

To: john@tiger

If the user to whom you want to send mail has an account on your workstation, and they have a *.forward* file that directs mail to their home workstation, you can enter just their login–name on the **To:** line. For example:

To: john

When you enter this, the *.forward* file knows to send john's mail to the machine called tiger. The *.forward* file simply states john's login–name and home machine. It resides in his home directory, on all the machines where he has accounts, so that all his mail is directed to his home machine tiger:

```
john@tiger
```

If you have accounts on remote workstations, put a *.forward* file stating your login–name and home machine on all the machines where you have accounts.

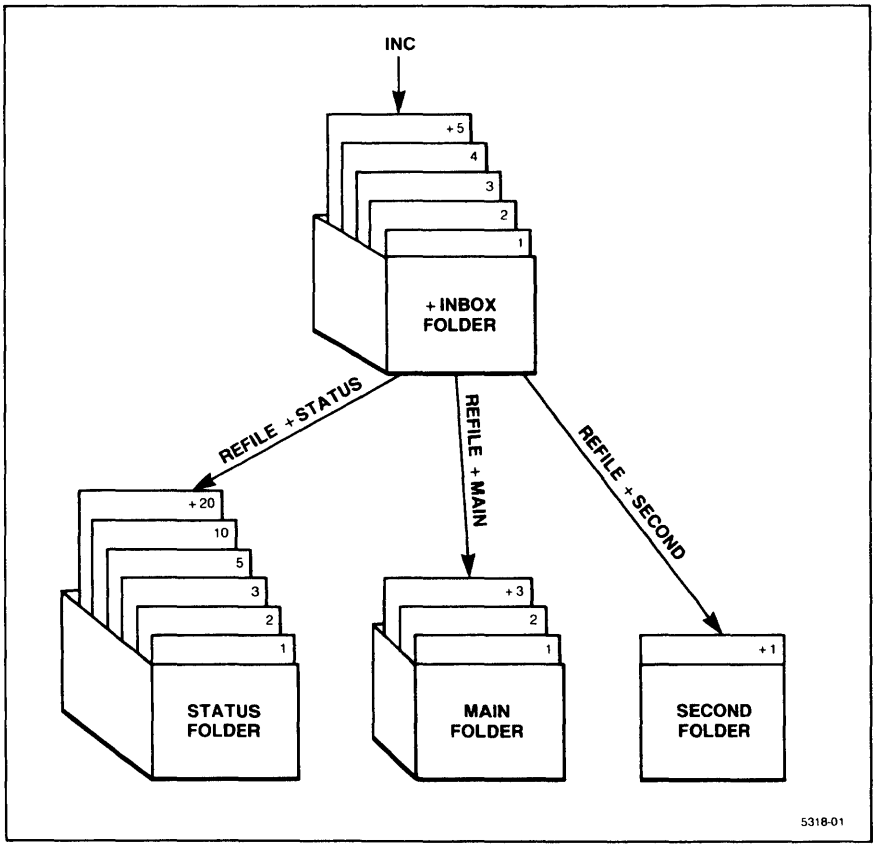


Figure 2A-1. Refiling Mail to Folders.

Refile creates and files messages into a folder. The syntax is:

$refile *mesg* +*folder*

Mesg is any message name and defaults to the current message if none is specified. If you try to **refile** a message into a folder that does not exist, the system asks:

Create folder "/usr/login–name/Mail/folder"?

Type **yes**(or **y**) and the folder will be set up.

The command **folder** +*folder* lists the current folder, the number of messages in it, the range of the messages, and the current message within the *folder*. For example:

```
$folder +status
```

This command will give the following information:

```
status has 6 messages ( 1-20); cur=20;
```

This line tells the folder name, the number of messages in the folder, the low to high range of message numbers, and which message is current.

Renumbering Your Mail Messages

When you remove messages, the message number of the other messages does not change. The **folder -pack** command renumbers the messages in a specified folder. For example, your *inbox* folder may contain messages 1, 2, 4, 8, 56, 99, 136, 146, 147, and 304. If you enter:

```
folder -pack +inbox
```

The messages are are renumbered as 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

For Further Information

This section has given the basic information you need to use the MH mail System. Several more commands are available, and each command has several options. For each command, you can use the **-help** option to list out the options that are available. For example, **comp -help** lists options to the **comp** command. When MH Mail asks you a question, hitting <RETURN> lists possible responses.

Table 2A-1 lists commands available to use in MH Mail. For more information about each command, refer to the *UTek Command Dictionary*.

Messa

A message is aut
that contains Mh
message by num

When referring tc
example:

```
show 5 las
show 6-9
scan first-
scan all
show 7 ne
show
```

Each message is

The command **folder** +*folder* lists the current folder, the number of messages in it, the range of the messages, and the current message within the *folder*. For example:

```
$folder +status
```

This command will give the following information:

```
status has 6 messages ( 1-20); cur=20;
```

This line tells the folder name, the number of messages in the folder, the low to high range of message numbers, and which message is current.

Renumbering Your Mail Messages

When you remove messages, the message number of the other messages does not change. The **folder -pack** command renumbers the messages in a specified folder. For example, your *inbox* folder may contain messages 1, 2, 4, 8, 56, 99, 136, 146, 147, and 304. If you enter:

```
folder -pack +inbox
```

The messages are are renumbered as 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

For Further Information

This section has given the basic information you need to use the MH mail System. Several more commands are available, and each command has several options. For each command, you can use the **-help** option to list out the options that are available. For example, **comp -help** lists options to the **comp** command. When MH Mail asks you a question, hitting <RETURN> lists possible responses.

Table 2A-1 lists commands available to use in MH Mail. For more information about each command, refer to the *UTek Command Dictionary*.

Message Naming

A message is automatically numbered when it enters your *inbox* folder (a directory that contains Mh mail messages is called a folder). You can refer to a specific message by number, or by the words **first, last, prev, next**, or **cur**.

number	The number of a message.
first	The first message in inbox.
last	The last message in inbox.
prev	The previous message.
next	The next message.
cur	The current message.

When referring to a message, you can use any combination of these names. For example:

show 5 last	Shows message 5 and the last message.
show 6–9 14	Shows messages 6–9 and 14.
scan first–5	Scans the first through fifth messages.
scan all	Scans all messages (first through last).
show 7 next cur	Shows messages 7 and 8, and the current message.
show	Shows the current message.

Each message is stored as a single file, and can be treated like any other file.

Table 2A-1
SUMMARY OF MH MAIL COMMANDS

Command/File	Description
comp(1MH)	Composes a message
folder(1MH)	Sets or lists current folder or message
inc(1MH)	Incorporates new mail
mhl(1MH)	Produces formatted listings of messages
mhpath(1MH)	Prints pathnames of messages and folders
.mh – mail(5MH)	File with user parameters
next(1MH)	Shows the next message
pick(1MH)	Selects messages by content
prev(1MH)	Shows the previous message
rmf(1MH)	Removes folder
rmm(1MH)	Removes messages
scan(1MH)	Summarizes a message header
send(1MH)	Sends a message
show(1MH)	Shows messages

2B

An Introduction to the Shell

Introduction

The shell is a command programming language that provides an interface to the UTek operating system. Its features include control–flow primitives, parameter passing, variables, and string substitution. Programming constructs such as **while, if then else, case,** and **for** are available. Two–way communication is possible between the shell and commands. String–valued parameters, typically filenames or options, can be passed to a command. Commands set a return value that you can use to determine control–flow. Standard output from a command can also be used as shell input.

The shell can modify the environment in which commands run. It redirects input and output to files, and invokes processes that communicate through *pipes*. The shell finds commands by searching directories in the file system in a sequence that is defined by the user. The shell can read commands from the terminal or from a file that stores command procedures.

This section describes the Bourne shell available on the UTek Operating System. The C–Shell, which allows constructs similar to the C programming language, is also available. See Section 2C for details on the C–Shell.

The *Simple Commands* part of this section covers most of the everyday requirements of casual users. The *Shell Procedures* part of this section describes those features of the shell intended for use within shell programs. These include the control–flow primitives and string–valued variables provided by the shell. The last part, *Keyword Parameters*, describes the more advanced features of the shell.

Simple Commands

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

```
who
```

This command prints the names of users logged into the system. The command:

```
ls -l
```

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

Background Commands

To execute a command, the shell normally creates a new process and waits for it to finish. But you can run a command without waiting for it to finish. For example:

```
cc pgm.c&
```

This command calls the C compiler to compile the file *pgm.c*. The trailing ampersand (&) instructs the shell not to wait for the command to finish. The prompt returns and you can continue working on other things. To help keep track of a background process, the shell reports its process number when you place the command in the background. You can obtain a list of currently active processes by using the **ps** command.

Input/Output Redirection

Most commands produce output (called *standard output*) to the terminal. You can redirect this output to a file using the notation >. For example:

ls –l >*filename*

The notation >*filename* is interpreted by the shell, instead of being passed as an argument to **ls**. If the *filename* does not exist, the shell creates the file; otherwise, the original contents of the file are replaced with the output from **ls**.

You can append output to the end of a file using the notation >>. For example:

ls –l >>*filename*

In this case, the output of the **ls** command is appended to the end of the file. If the file does not exist, it is created.

The *standard input* of a command can be taken from a file instead of the terminal using the < notation. For example:

wc<*filename*

The **wc** command reads the standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then you can use the **–l** option:

wc –l <*filename*

Pipelines and Filters

The standard output of one command can be connected to the standard input of another by using a *pipe* operator, indicated by a vertical bar character (¦) between the commands. For example:

ls –l ¦ wc

Two or more commands connected in this way constitute a *pipeline*. Its overall effect is the same as the following command, except that no intermediate file is used.

ls –l >*filename*;**wc** <*filename*

When they are connected with a pipes, the two processes run in parallel. Pipes are unidirectional, and the two processes are synchronized by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

A *filter* is a command that reads standard input, transforms it in some way, and prints the result as output. One such filter, **grep**, selects from its input lines that contain a specified string. For example:

ls ¦ grep old

This command prints lines from the output of **ls** that contain the string *old*. Another useful filter is **sort**. For example:

who ¦ sort

This command prints an alphabetically sorted list of all the users who are logged onto the system.

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

ls ¦ grep old ¦ wc –l

This command prints only the number of filenames in the current directory that contain the string *old*.

Filename Matching

Many commands accept arguments that are filenames. For example:

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

This command prints only information relating to the file *main.c*. The **ls -l** command by itself prints the same information about all files in the current directory.

The shell provides a mechanism for generating a list of filenames that match a pattern. For example:

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

This command generates as arguments to **ls** all filenames that end in *.c* in the current directory. The asterisk pattern matches any string, including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a hyphen matches any character between the pair.

For example:

```
[a-z]*
```

This command matches all names in the current directory beginning with one of the letters *a* through *z*.

The input:

/usr/fred/test/?

matches all names in the directory */usr/fred/test* that consist of a single character. If no filename is found that matches the pattern, then the string refers to a file named ?.

This mechanism is useful both to save typing and to select names according to some pattern. You can also use it to find files. For example:

echo /usr/fred/*/core

This command finds and prints the names of all *core* files in subdirectories of */usr/fred*. This last process can take a long time, because it requires a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. A period (.) at the start of a filename must be explicitly matched. For example, this input echoes all file names in the current directory not beginning with a period:

echo *

This input echoes all filenames that begin with a period:

echo .*

This command avoids inadvertent matching of the names "." and "..", which stand for the current directory and the parent directory.

Quoting in the Shell

Characters that have a special meaning to the shell are called metacharacters. Here are some examples:

```
< > * ? ¦ &
```

Any metacharacter preceded by a backslash (\) is *quoted*; it loses its special meaning. In the output, the backslash disappears. For example, this command echoes a single dollar sign:

```
echo ?
```

And this command echos a single backslash.

```
echo \\
```

To allow long strings to be continued over more than one line, the sequence \<RETURN> is ignored.

The backslash is convenient for quoting single characters. When you want to quote more than one character, using many backslashes is clumsy and difficult. You can quote a string of characters by enclosing the string between single quotes. For example:

```
echo xx'****'xx
```

This command echoes the output:

```
xx****xx
```

The string placed within quote marks cannot contain a single quote, but it can contain the newline character. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism, using double quotes, is also available. It prevents interpretation of some, but not all, metacharacters. Details of quoting are described in the topic *Evaluation and Quoting*.

Prompting by the Shell

When you use the shell from a terminal, it issues a prompt indicating that it is ready for a command from the terminal. By default, this prompt is $. You can change the prompt by entering:

PS1=*newprompt*

This sets the prompt to the string *newprompt* until you log off the system. For information on resetting the default prompt permanently, see the section *Using UTek on the Workstation* in your *System Guide.*

When the shell needs further input for some commands, it issues a secondary prompt. By default, this secondary prompt is >. To change this secondary prompt enter:

PS2=*newprompt*.

Shell Procedures

You can use the shell to read and execute commands contained in a file. For example, this calls the shell to read commands from the file *filename.*

sh *filename [arguments]*

Such a file is called a *shell program*, a *command procedure* or a *shell procedure.* Arguments may be supplied with the command. In the file arguments are referenced using the *positional parameters* $1, $2, and so on. For example, if the file *wg* contains:

```
who ¦ grep $1
```

then the call:

```
sh wg fred
```

is equivalent to the command:

```
who ¦ grep fred
```

In addition to containing the command procedures, the file *wg* must also contain one line that invokes the shell to run the command procedures. So at the beginning of the file *wg* (or any other file containing a shell procedure) enter:

#!/bin/sh

When a UTek system file has the execute (x) attribute, you can execute a file directly, without invoking a special shell on the command line. To make a file executable, you can use the **chmod** command. For example, to make the file *wg* executable enter:

chmod +x wg

This command is equivalent to entering:

sh wg fred

This lets you use shell procedures and programs interchangeably. In either case, the shell creates a new process to execute the command.

As well as providing names for the positional parameters, the number of positional parameters when you invoke the shell is available as $#. The name of the file being executed is available as $0.

A special shell parameter $* is used to substitute for all positional parameters except $0. A typical use of this is to provide some default arguments.

Control Flow — for

A frequent use of shell procedures is to loop through the arguments ($1, $2, ...) executing commands once for each argument. An example of such a procedure is *tel*, which searches the file */usr/lib/telnos.* The file *telnos* contains lines of the form:

fred mh0123
bert mh0789

The text of **tel** is:

```
for i
do
   grep $i /usr/lib/telnos
done
```

The following command prints those lines in */usr/lib/telnos* that contain the string *fred*:

tel fred

The following command prints those lines containing *fred*, followed by those lines containing *bert*.

tel fred bert

The **for** loop notation is recognized by the shell. It has the general form:

for *name* **in** *word1* **and** *word2*
do
command-list
done

A *command-list* is a sequence of one or more simple commands, separated or terminated by a newline character or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. A *name* is a shell variable that is set to the words *word1 word2* in turn, each time the *command-list* following **do** is executed. If you eliminate **in** *word1 word2*, then the loop is executed once for each positional parameter, that is, **in** $* is assumed.

You can replace **do** with a left brace ({), and **done** with a right brace (}). Another example of the use of the **for** loop is the **create** command, whose text is:

```
for i do >$i; done
```

Notice that a semicolon (or a newline) is required before **done**. So the command:

create alpha beta

ensures that two empty files, *alpha* and *beta*, exist and are empty. You can use the notation *>file* by itself to create a new file or clear the contents of an existing file.

Control Flow — case

A multiple-choice branch is provided by the **case** notation. For example, an append command:

```
case $# in
1)cat >>$1;;
2)cat >>$2;;
3)echo 'usage append [ from to';;
esac
```

When called with one argument, $# is the string *1*, and the standard input is appended onto the end of *file* using the **cat** command. When called with two arguments, the contents of the first argument (file) are appended onto the second argument (file). If the number of arguments to append is other than 1 or 2, a message indicating proper usage displays.

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

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

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

NOTE

No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second asterisk are never executed since the first asterisk executes everything it receives.

```
case $# in
   *)...;;
   *)...;;
esac
```

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

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

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by the character ¦. For example:

```
case $i in
-x ¦ -y)...
esac
```

is equivalent to:

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

The usual quoting conventions apply, so that this entry matches the character **?**:

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

Here Documents

The shell procedure **tel** described earlier uses the file */usr/lib/telnos* to supply the data for **grep**. An alternative is to include this data within the shell procedure as a *here* document. For example:

```
for i
do
     grep $i<<!
     ...
     fred mh0123
     bert mh0789
     ...
!
done
```

In this example, the shell takes the lines between << ! and ! as the standard input for **grep.**

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

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

This call:

edg *string1 string2 filename*

is then equivalent to the command:

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

This command changes all occurrences of *string1* in *filename* to *string2*. You can prevent substitution using a backslash (\) to quote the special character dollar sign. For example:

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

This version of **edg** is equivalent to the first except that **ed** prints a question mark if there are no occurrences of the string $1.

Substitution within a *here* document may be prevented entirely by quoting the terminating string. For example

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

The here document is not parameter substituted before presentation to **grep**. If parameter substitution is not required in a *here* document, this latter form, using the pound sign (#) instead of << and <<, is more efficient.

The notation <<- tells the shell to strip leading tabs.

Shell Variables

The shell provides **string-valued variables**. Variable names begin with a letter and consist of letters, digits, and underscores. You can assign a value to a variable like this:

user=fred box=m000 acct=mh0000

This assigns values to the variables *user, box*, and *acct*. You can set a variable to the null string by entering:

null=

You can substitute the value of a variable by preceding its name with a dollar sign. For example:

echo $user

This command echoes *fred*.

You can use variables interactively to abbreviate frequently–used strings. For example,

```
b=/usr/fred/bin
mv marv $b
```

This command moves the file *marv* from the current directory to the directory */usr/fred/bin*.

A more general notation is available for parameter (or variable) substitution. For example:

```
echo ${user}
```

This command is equivalent to:

```
echo $user
```

It is used when the parameter name is followed by a letter or digit. For example:

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

This directs the output of **ps** to the file /tmp/psa, whereas:

```
ps a >$tmpa
```

causes the variable *tmpa* to be substituted. This substitutes the value of the variable *tmpa*.

The shell initially sets the following values. All of them, except the exit status ($?) are set initially.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of returned codes is dealt with later under the **if** and **while** commands.

$# The number of positional parameters in decimal. Used, for example, in the **append** command to check the number of parameters.

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

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

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

$- The current shell options, such as **-x** and **-v**.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since you last looked at it, the shell prints the message *You have mail* before prompting for the next command. This variable is typically set in the .profile of the user's home directory. For example:

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

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

cd /usr/fred/bin

This makes the current directory */usr/fred/bin*. The command **cd** without an argument is equivalent to:

cd $HOME

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

$PATH A list of directories containing commands. Each time the shell executes a command, it searches a list of directories for an executable file. If *$PATH* is not set, the current directory, */bin* and */usr/bin* are searched by default. Otherwise, *$PATH* consists of directory names separated by a colon. For example:

PATH=:/usr/fred/bin:/bin:/usr/bin

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

$ENV An interactive shell can read the value of ENV. ENV is set to the name of a file, and commands are read from that file and executed at login. This variable has no default value, so you must set it in the *.profile* file. Usually, this variable is used to define shell functions.

$PS1 The primary shell prompt string, by default a dollar sign ($).

$PS2 The shell prompt when further input is needed, by default the greater than symbol (>).

$IFS The set of characters used by *blank interpretation*. (For more on blank interpretation see the topic *Keyword Parameters*.)

Control Flow — while

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

while *command-list 1*
do
  command-list 2
done

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list 1* is executed. If a zero exit status is returned, the *command-list 2* is executed; otherwise, the loop terminates. For example:

```
while test $1
do
   ...
   shift
done
```

This shell command is equivalent to:

```
for i
do
   ...
done
```

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

The **shift** command also accepts a numeric argument, so that you can shift more than one position. See *UTek Command Reference, shift(1sh)* for details. Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example:

```
until test -f file
do
   sleep 300
done
```

commands

This shell program loops until *file* exists. Each time around the loop, it waits for five minutes (300 seconds) before trying again.

Control Flow — if

A general conditional branch is also available:

if *command-list*
then
  command-list
else
  command-list
fi

This loop tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file. For example:

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

A multiple–test **if** command can be written using an extension of the **if** notation. The **elif** command is a combination of **else** and **if**. For example:

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

The **touch** command changes the last–modified time for a list of files. The command may be used in conjunction with **make** to force recompilation of a list of files.

The following is an example of the **touch** command:

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

The **–c** option is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some nonnull string if the **—c** option is encountered. The **ln** and **rm** commands make a link to the file and then remove it.

The sequence:

if *command1*
then
  command2
fi

This sequence can be written as:

command1 **&&** *command2*

Conversely, this sequence executes *command2* only if *command1* fails:

command1 ¦ ¦ *command2*

In each case, the value returned is that of the last simple command executed.

Test Command

The **test** command is extremely useful when you write shell programs. It implements a number of programming functions, such as algebraic comparison of numbers, string operators, and file status.

The **test** command is used with the **if** and **while** commands to test conditions. If the condition is true, **test** exits with a status of 0. If the condition is false, the exit status is something other than 0.

Algebraic Comparisons

A number of options to the **test** command compare quantities. One commonly used option is **-lt**, that checks to see if one quantity is less than another. The shell program in the following example tests to see if there is at least one argument given on the command line of the shell program. Recall that $# is the number of arguments given to the shell program:

```
#!/bin/sh
if test $# -lt 1
then
   echo "No arguments given."
   exit 1
fi
exit 0
```

The command **test $# -lt 1** exits with a nonzero status if the value of $# (number of arguments) is less than one, and the message *No arguments given* displays on the terminal.

Following are the options that are available for algebraic comparison:

- **-eq** equal
- **-ne** not equal
- **-gt** greater than
- **-lt** less than
- **-le** less than or equal to

String Operators

You can also use **test** to evaluate strings. The simplest use of the **test** command checks to see if an argument to **test** is a nonempty string. For example:

test foo

The **test** command evaluates to true because *foo* is a non-empty string. The **-n** option to the **test** command also checks for a non-empty string. So the following example is the same as the command **test foo**:

test -n foo

The **-z** option, followed by a string, evaluates to true if the string is zero length.

You can also compare two strings to see if they are the same or not. The two operators are equal (=) and not equal (!=). Here are three examples:

test foo = bar	evaluates to false
test foo != bar	evaluates to true
test bar = bar	evaluates to true

You can also evaluate the length of a string using the **-l** option. The expression **-l** *string* is replaced by the length of the given string. This example shows a shell program that reads data from the terminal until a string longer than 12 characters is entered:

```
#!/bin/sh
line=""
while test -l "$line" -lt 12
do
   echo "Input a string of at least 12 characters."
   read line
done
```

You can combine algebraic and string comparisons. Three logical operators help accomplish this:

! not

-o or

-a and

Following is an example of using logical operators to combine algebraic and string comparisons:

```
#!/bin/sh
if test \( "0$1" -eq 0 \) -a \( "0$1" -eq != "00" \)
then
   echo "max : First argument ($1) is not a number."
   exit 1
fi
```

This example makes sure that the first argument of the shell program is a number. The expression **\("0$1" -eq 0 \)** checks to see if the value of the first argument is 0. The expression **\("0$1" -eq** != **"00" \)** checks to see if the value of the first argument is actually 0. The reason for this second check is that if the first argument is not a number, it is converted to 0. The **-a** operator says that if the first argument is equal to zero, but not actually the string zero itself, then the first argument is not a number.

File Status

You can also use the **test** command to see if a file exists, and whether it is readable or writeable. For example, the following command tests to see if a file exists:

test -f *filename*

This command returns zero exit status if *file* exists, and non-zero exit status otherwise.

Some of the more frequently used **test** options that give you information about file status are listed below.

test -f *file*	**true if** *file* exists.
test -r *file*	**true if** *file* is readable.
test -w *file*	**true if** *file* is writable.
test -d *file*	**true if** *file* is a directory.

Three other integer functions are available that give you information about files:

-M *file*	Replaced by last modification time for the named file. If the file does not exist, the time is set to 0.
-C *file*	Replaced by the last time the status of the file changed. See the *UTek Command Reference, stat(2)* for information on file status.
-A *file*	Replaced by the last time the file was accessed.

The which Command

In addition to verification using the **test** command, you frequently need to know what version of a command a shell program executes. Within a shell program you can use the **which** command in this form:

which *command1 command2*

Each command is described in terms of how it would be executed. Messages display to tell you if a command is built into the shell, if it's a function, or the pathname it executes.

Command Grouping

You can group commands in two ways:

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

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

(cd x;rm junk)

This command executes **rm junk** in the directory *x* without changing the current directory of the invoking shell.

Debugging Shell Procedures

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

set –v

This command causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering:

sh –v proc ...

In this command *proc* is the name of the shell procedure. This option may be used in conjunction with the **–n** option, which prevents execution of subsequent commands. (Note that entering **set –n** at a terminal leaves the terminal useless until you type an end–of–file.)

This command produces an execution trace:

set –x

Following parameter substitution, each command is printed as it is executed.

You can turn off both the **–x** and the **–n** options by entering:

set —

The current setting of the shell options is available in a procedure as $—.

Keyword Parameters

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

user=fred *command*

This executes *command* with *user* set to *fred*. The **—k** option causes arguments of the form *name*=*value* to be interpreted in this way anywhere in the argument list.

Such *names* are sometimes called *keyword parameters*. If any arguments remain, they are available as positional parameters $1, $2, etc.

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

set —*

This command sets $1 to the first file name in the current directory, $2 to the next, and so on. A leading hyphen (-) turns the option on, and an addition sign turns the option off. Note that the first argument, —, ensures correct treatment when the first filename begins with a —. The **—a** option to **set** marks variables that are exported to the environment.

Parameter Transmission

When you invoke a shell procedure, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported, using the **export** command. For example:

export user box

This command marks the variables *user* and *box* for export. When a shell procedure is invoked, copies are made of all exportable variables so that you can use them. If you modify exported variables within the procedure it does not affect the values in the invoking **shell**. Generally, a **shell** procedure cannot modify the state of its caller without an explicit request on the part of the caller.

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

readonly *name*

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution

If a shell parameter is not set, then the null string is substituted for it. For example:

echo $d

This command echos nothing. You can give a default string that will echo the value of the variable *d* if it is set, and something else otherwise. For example:

echo ${d-$1}

This command echoes the value of *d* if it is set, and the value (if any) of $1 if it is not set.

You can assign a variable a default value. For example:

echo ${d=.}

This command sets *d* to the string . (a period) if it was not previously set. (The notation ${*variable* = *value*} is not available for positional parameters.)

If there is no default, the command:

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

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

:${user?}${acct?}${bin?}

The colon (:) is a command built into the shell that does nothing once its arguments have been evaluated. If any of the variables are not set, execution of the shell procedure ends.

Command Substitution

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

d = `pwd`

The entire string between single quotes is taken as the command to be executed, and is replaced with the output from the command. The command is written using the usual quoting conventions, except that a single quote must be escaped using a backslash (\).

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

basename main.c .c

This command prints the string *main*. Its use is illustrated by the following fragment from the **cc** command:

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

This sets *B* to the part of *$A* with the suffix *.c* removed.

Evaluation and Quoting

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

Initially commands are parsed according to the grammar given in Table 2B-2 at the end of this section.. Only one evaluation occurs. So if you set variable *x* to the string $y, the command **echo $x** echoes the string $y. Before the shell executes a command, it makes the following substitutions in this order:

- parameter substitution, for example **$user**
- command substitution, for example `` `pwd` ``
- blank interpretation
- filename generation

The first two items in the list were discussed earlier. After parameter and command substitution take place, the resulting characters are broken into nonblank words. For this purpose, blanks are the characters defined in the string $IFS in your *.profile* or *.login* file. By default, this string consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted.

In filename generation each word is scanned for the special characters *, ?, and [ ]. An alphabetical list of filenames is generated to replace the word. Each such filename then becomes a separate argument.

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

As well as the quoting mechanisms described earlier (using backslash and single quotes) a third quoting mechanism using double quotes is provided. Within double quotes, parameter and command substitution occurs; but filename generation and the interpretation of blanks does not.

The following characters have a special meaning with double quotes. You can quote them using a backslash.

$ parameter substitution

` command substitution

" ends the quoted string

\ quotes the special characters $ ` " \

For example, this command passes the positional parameters as a single argument:

```
echo "$*"
```

It is equivalent to:

```
echo "$1, $2, ..."
```

The notation $@ is the same as $*, except when it is quoted. For example:

```
echo "$@"
```

This passes the positional parameters, unevaluated, to **echo**. It is equivalent to:

```
echo "$1" "$2" ...
```

The following table gives the shell metacharacters that are evaluated for each quoting mechanism.

Table 2B–1
QUOTING AND EVALUATION OF SHELL METACHARACTERS

	metacharacter					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t = terminator
y = interpreted
n = not interpreted

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable *X* has the value $y, and if *y* has the value pqr, then this command echoes the string *pqr*.

```
eval echo $X
```

In general, the **eval** command evaluates its arguments and treats the result as input to the shell. The input is read, and the resulting commands are executed. For example:

```
wg='eval who ¦ grep'
$wg fred
```

This is equivalent to:

```
who ¦ grep fred
```

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

Error Handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal. A shell invoked with the —i option is also interactive.

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

- Input/output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a bus error or memory fault signal.
- The command terminates normally but returns a nonzero exit status.

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

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

The shell option **—e** causes the shell to terminate if any error is detected. The following is a list of the UTek operating system signals.

1 hangup
2 interrupt
3* quit
4* illegal instruction
5* trace trap
6* IOT instruction
7* EMT instruction
8* floating point exception
9 kill (cannot be caught or ignored)
10* bus error
11* segmentation violation
12* bad argument to system call
13 write on a pipe with no one to read it
14 alarm clock
15 software termination (from **kill**)

The UTek operating system signals marked with an asterisk (*) produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list used by shell programs are 1, 2, 3, 14, and 15.

Fault Handling

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

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

This sets a trap for signal 2 (terminal interrupt). If this signal is received, it executes the following commands:

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

Signal 11 (segmentation fault) cannot be trapped.

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

UTek operating system signals can be handled in one of three ways.

- They can be ignored, so the signal is never sent to the process
- They can be caught, so the process must decide what action to take when the signal is received
- They can be left to cause termination of the process without taking any further action.

If a signal is being ignored on entry to the shell procedure, **trap** commands are ignored.

The use of **trap** is shown in this modified version of the **touch** command:

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

The cleanup action is to remove the file *junk$$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the UTek operating system, 0 is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure can ignore signals by specifying the null string as the argument to **trap**. For example:

trap '' 1 2 3 15

This command is a fragment from the **nohup** command. **Nohup** ignores the operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals. These signals are ignored by both the procedure and the invoked commands.

You can reset traps by entering:

trap 2,3

This resets the traps for signals 2 and 3 to their default values. You can obtain a list of the current values of traps by entering:

trap

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

```
d=`pwd`
for i in *
do
   if test -d $d/$i
   then
      cd $d/$i
      while echo "$i:" && ftrap exit 2 && read x
      do
         trap : 2
         eval $x
      done
   fi
done
```

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

Shell Functions

Shell functions provide a convenient way of abbreviating commands for entry on a command line or within shell programs. A shell function is similar to a C–Shell alias, except that a function has the easier syntax of a shell program.

Executing Functions

Normally, when the shell executes a command it looks for the command in three places. First, it looks to see if the command is built into the shell itself, such as **cd** and **test**. Second, the shell reads user–defined functions. After looking for the command in these two places, the shell looks for executable files defined in the PATH variable. Because functions are read after built–in commands, you can never give a function the same name as a built–in command.

The general form to define a function is:

function-name () *command-list*

For *command-list*, substitute one or more valid UTek commands. In this command, *function-name* is an abbreviated name you choose for the commands in *command-list*. However, *function-name* cannot be the same as the name of a built–in shell command. If you are not sure whether a command is a built–in command, use the **which** command to find out.

Following are two function definitions:

```
cl () clear
todo () { date; cat $HOME/.todo; }
```

The first example shortens the **clear** command to the function-name **cl**. Now each time the shell reads **cl**, it clears the screen. When you enter the command **todo**, the second example displays the date, followed by a file that presumably contains a list of things you need to do today.

Another way to write these two functions involves entering each portion of the definition on separate lines. The $ is the primary shell prompt (defined by $PS1) and the > is the secondary shell prompt (defined by $PS2). This example shows the same two functions defined previously:

```
$ cl ()
> clear
$

$ todo ()
> {
> date
> cat $HOME/.todo
> }
$
```

To find out what functions are set within the current shell, enter **list**. The **list** command without arguments lists all shell variables and functions.

You could execute functions at a shell prompt, or in a shell program. But you must redefine the functions within each shell. The next topic, *Passing Functions to Subshells*, discusses how to pass the capabilities of a function to another shell.

Passing Functions to Subshells

Although functions are listed with shell variables, they cannot be passed into your environment using the **export** command. So when you invoke a new shell, no functions are defined.

To define new functions for an interactive shell, you can set the shell variable $ENV to the name of a file that contains the definitions for shell functions. Then each time you invoke a new shell, this file is read for commands. Because $ENV is not defined by default when you log in, you must explicitly set it and export it in your *.profile* file. For example:

```
ENV=$HOME
export ENV
```

Non–interactive shells, such as those executed by shell programs, do not read the variable $ENV. To execute your shell functions within a shell program you must use the commands:

```
if [ "$ENV" ]
then
   $ENV
fi
```

Function Arguments

When you execute a shell function, $@ contains the arguments given to the function. As with a shell program, arguments are placed in $1, $2, and so on. So you can convert many shell programs to internal functions without changing them.

But when a shell program uses functions, the function redefines the value of $@ with different arguments. All arguments in the shell program that you need later must be saved prior to executing a function.

Following is an example of a shell function with arguments. The function, called **chdir**, changes to the directory you give as the first argument. Without an argument, **chdir** changes to $HOME:

```
chdir()
{
   if [ "$1" ]
   then
      cd "$1"
   else
      cd
   fi
}
```

Exiting from Functions

In older versions of the Bourne Shell, the **exit** command only exited from shell scripts. Now, the **exit** command unconditionally causes exit from the current shell.

Because you don't want to delete the current shell to exit from a function (recall that the function does not invoke a separate shell), a special command to exit is provided. The command to exit from a shell function has the following form:

return *n*

In this command *n* is set to the return value of the last command. When the shell executes the **return** command, it sets the shell variable $? (exit code of last command executed) to the value of *n*.

Command Execution

To run a command (other than a built–in shell command), the shell first creates a new process using the system call **fork**. The execution environment for the command includes input, output and the states of signals. The environment is established in the child process before the command is executed. The built–in command **exec** is used in rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the **nohup** command looks like this:

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

The **trap** turns off the signals specified so that they are ignored by subsequently–created commands, and **exec** replaces the shell by the command specified.

Most forms of input/output redirection have already been described. In the following list *word* is only subject to parameter and command substitution. No filename generation or blank interpretation takes place. Input/output specifications are evaluated left to right as they appear in the command. Following are some input/output specifications:

>*word*	The standard output (file descriptor 1) is sent to the file *word*, which is created if it does not already exist.
>>*word*	The standard output is sent to file *word*. If the file exists, then output is appended; otherwise, the file is created.
<*word*	The standard input (file descriptor 0) is taken from the file *word*.
<<*word*	The standard input is taken from the lines of shell input that follow, up to but not including a line consisting only of *word*. If *word* is quoted, parameter and command substitution occur and backslash is used to quote the characters $, ', and the first character of *word*. In the latter case, \<RETURN> is ignored.
>& digit	The file descriptor *digit* is duplicated using the system call **dup**, and the result is used as the standard output.
<& digit	The standard input is duplicated from the file descriptor *digit*.
<	The standard input is closed.
>	The standard output is closed.

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

...2>*filename*

This runs a command with message output (file descriptor 2) directed to *file*. Another example:

...2>&1

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

The environment for a command run in the background is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the shell and the command) that run in parallel from trying to read the same input.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This modification allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UTek operating system convention for a signal is that if it is set to 1 (ignored), then it is never changed even for a short time. Note that the shell command **trap** has no effect for an ignored signal.

Invoking the Shell

The following options are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, commands are read from the file *.profile*. These are the most commonly-used options. For a complete list see *UTek Command Reference, sh(1)*.

—c*string* If the **—c** option is present, then commands are read from *string*.

—s If the **—s** option is present, or if no arguments remain, commands are read from the standard input. Shell output is written to file descriptor 2.

—i If the **—i** option is present, or if the shell input and output are attached to a terminal (as determined by **getty**), the shell is interactive. In this case, TERMINATE is ignored so that kill 0 does not kill an interactive shell. INTERRUPT is caught and ignored so that **wait** is interruptable. In all cases, QUIT is ignored by the shell.

Table 2B–2
SHELL GRAMMAR

item	word input–output name = value
simple-command	item simple–command item
command	simple–command (command–list) {command–list}
	for *name* **do** *command-list* **done**
	while *command-list* **do** *command-list* **done**
	until *command-list* **do** *command-list* **done**
	case *word* **in** *case-part* ... **esac**
	if *command-list* **then** *command-list* *else-part* **fi**
pipeline:	*command* *pipeline* ¦ *command*
and/or:	*pipeline* *and/or* & & *pipeline* *and/or* ¦¦ *pipeline*
command-list	*and/or* *command-list;* *command-list&* *command-list; and/or* *command-list* & *and/or*
input-output;	> *word* < *word* >> *word* << *word*
file	*word* & *digit* & -
case-part:	*pattern*) *command-list* ;;
pattern:	*word* *pattern* ¦ *word*

else-part:	**elif** *command-list* **then** *command-list* *else-part* **else** *command-list*
empty:	*empty*
word:	*sequence of non-blank characters*
name	*sequence of letters, digits, or underscores starting with a letter*
digit	*0 1 2 3 4 5 6 7 8 9*

TABLE 2B-3
METACHARACTERS AND RESERVED WORDS

(a)*syntactic:*

|	pipe symbol
&&	andf symbol
||	orf symbol
;	command separator
;;	case delimiter
&	background commands
()	command grouping'
<	input redirection
<<	input from a here document
>	output creation
>>	output append

(b)patterns:

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

(c) *substitution:*

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

(d) *quoting*:

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

(e) *reserved words*:

if then else elif fi
case in esac
for while until do done
{ } [ ] test
echo
type
which
pwd

Introduction to the C-Shell

Introduction

The C-Shell is a command interpreter for the UTek operating system that allows you to write shell programs using constructs similar to the C programming language. The other shell available on the UTek system, the Bourne shell, is described in section 2B. The C-Shell provides several features not available in the Bourne shell: job control to track the progress of a job easily and manipulate its priority, the ability to abbreviate, or *alias*, commands for easy access, and a history list of past commands that lets you quickly execute all or part of a previous command.

Most features of the Bourne shell are also included in the C-Shell. Its features include control-flow primitives, parameter passing, variables, and string substitution. Programming constructs such as **while, if then else, case,** and **for** are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically filenames or options, can be passed to a command. Commands set a return value that you can use to determine control-flow. Standard output from a command can also be used as shell input.

The shell can modify the environment in which commands run. It redirects input and output to files, and invokes processes that communicate through *pipes*. The shell finds commands by searching directories in the file system in a sequence that is defined by the user. The shell can read commands from the terminal or from a file that stores command procedures.

Simple Commands

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

```
who
```

This command prints the names of users logged into the system. The command:

```
ls -l
```

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

Input/Output Redirection

Most commands produce output (called *standard output*) to the terminal. You can redirect this output to a file using the notation >. For example:

ls –l *>filename*

The notation *>filename* is interpreted by the shell, instead of being passed as an argument to **ls**. If the *filename* does not exist, the shell creates the file; otherwise, the original contents of the file are replaced with the output from **ls**.

You can append output to the end of a file using the notation >>. For example:

ls –l *>>filename*

In this case, the output of the **ls** command is appended to the end of the file. If the file does not exist, it is created.

The *standard input* of a command can be taken from a file instead of the terminal using the < notation. For example:

wc *<filename*

The **wc** command reads the standard input (in this case redirected from the file *filename*) and prints the number of characters, words, and lines found. If only the number of lines is required, then you can use the **–l** option:

wc –l *<filename*

The C–Shell can also redirect the standard error independent of the standard output. You can redirect error output to a file using the following form:

>& *filename*

or you can append the error output to a file using the form:

>>& *filename*

You can also pipe output from other commands to the standard error using:

¦ &

If you have the C-Shell *noclobber* variable set, redirection of input and output cannot write to a file that exists, except for special files like terminals or /dev/null. But if *noclobber* is set, you can suppress this check for an existing file by using an exclamation mark after the redirection symbol. For example:

>! >>! >&! >>&!

Pipelines and Filters

The standard output of one command can be connected to the standard input of another by using a *pipe* operator, indicated by ¦ between the commands. Two or more commands connected in this way constitute a *pipeline*. For example:

ls –l ¦ wc

Its overall effect is the same as the following command, except that no intermediate file is used.

ls –l >*file*;**wc** <*file*

When they are connected with a pipes, the two processes run in parallel. Pipes are unidirectional, and the two processes are synchronized by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**, selects from its input lines that contain a specified string. For example:

ls ¦ grep old

This command prints lines from the output of **ls** that contain the string *old*. Another useful filter is **sort**. For example:

who ¦ sort

This command prints an alphabetically sorted list of all the users who are logged onto the system. A pipeline may consist of more than two commands. For example:

ls ¦ grep old ¦ wc –l

This command prints only the number of filenames in the current directory that contain the string *old*.

Filename Matching

Many commands accept arguments that are filenames. For example:

ls –l main.c

This command prints only information relating to the file *main.c*. The **ls –l** command by itself prints the same information about all files in the current directory.

The shell provides a mechanism for generating a list of filenames that match a pattern. For example:

ls –l *.c

This command generates as arguments to **ls** all filenames that end in *.c* in the current directory. The asterisk (*) pattern matches any string, including the null string. In general, patterns are specified as follows:

~	Matches the home directory of a specified user, or if no user is specified it expands to the home directory of the current user.
*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a hyphen matches any single character between the pair.
{...}	Matches each one of the characters enclosed with a preceding and/or trailing string.

For example:

[a–z]*

This command matches all names in the current directory beginning with one of the letters *a* through *z*.

The input:

/usr/fred/test/?

matches all names in the directory */usr/fred/test* that consist of a single character. If no filename is found that matches the question mark, the shell looks for a file named ?.

This mechanism is useful both to save typing and to select names according to some pattern. You can also use it to find files. For example:

echo /usr/fred/*/core

This command finds and prints the names of all *core* files in subdirectories of */usr/fred*. This last process can take a long time, because it requires a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. A period at the start of a filename must be explicitly matched.

This input echoes all filenames that begin with a period:

echo .*

This command avoids inadvertent matching of the names "." and "..", which stand for the current directory and the parent directory. The notation **a{b,c,d}e** is a shorthand for **abe ace ade**. This is particularly useful for matching several files whose pathnames have the same beginning or trailing components. For example:

/source/{oldls,ls}.c

This notation expands to the filenames */source/oldls.c* and */source/ls.c*.

Quoting

Characters that have a special meaning to the shell are called *metacharacters*. Here are some examples:

< > * ? ¦ &

Any metacharacter preceded by a backslash (\) is *quoted*, and loses its special meaning. In this discussion, the word *quote* is unrelated to quotation marks. It means "remove the special meaning."

When you use a backslash to quote a character, the backslash does not appear in the output. For example, this command echoes a single dollar sign:

echo \$

And this command echos a single backslash.

**echo **

To allow long strings to be continued over more than one line, the sequence \<RETURN> is ignored.

The backslash is convenient for quoting single characters. When you want to quote more than one character, using a backslash is clumsy and difficult. You can quote a string of characters by enclosing the string between single quotes. For example:

echo xx'**'xx**

This command echoes the output:

xx****xx

The quoted string cannot contain a single quote, but it can contain the newline character. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism, using double quotation marks, is also available. It prevents interpretation of some, but not all, metacharacters.

Starting and Terminating the C-Shell

When you log in to the system, you are in your home directory. The system starts the shell automatically. As you log in, the shell reads commands from a file in your home directory called *.cshrc*. Each time you invoke a new C-Shell, the commands in this file are read. Read further in this topic for more information on what commands to put in the *.cshrc* file.

Each time you log in, but not each time you invoke a new shell, a file in your home directory called *.login* is read for commands. After reading the commands from *.cshrc*, the login shell reads commands from *.login*. Read further in this topic for more information on what commands to put in the *.login* file.

If you want to change from the Bourne shell to the C-Shell, change your entry in the system password file so that UTek invokes **csh** when you log in. To do this you need to use the **chsh** (change shell) command. To make the C-Shell your login shell, type:

chsh *login-name* **/bin/csh**

A Sample .cshrc File

The *.cshrc* file contains C-Shell-specific commands that you want to execute every time you create a new C-Shell.

This file sets up environment variables, the pathname searched by the C-Shell to execute commands, and any C-Shell variables that you want to set. It also abbreviates (aliases) frequently-used commands.

Each time that you invoke a new C-Shell, *.cshrc* executes. Because the *.cshrc* file sets so many parameters, it can take a long time to execute. If it takes too long to invoke a new C-Shell, remove some of the aliases that you use infrequently.

Example 2C–1 shows a sample *.cshrc* file.

```
 1 set path=( . ~/.bin /bin /usr/bin )
 2 if ($?prompt) then
 3         set history=29
 4         set mail=(60 /usr/spool/mail/$USER)
 5         setenv EDIT vi
 6         setenv MORE 'page -u -f'
 7         setenv SEDIT vi
 8 endif
 9 alias more 'more  -u -f *'
10 alias hi history
11 alias who 'who * ¦ sort ¦ more'
```

Example 2C–1. A Sample .cshrc File.

The path Variable (line 1)

The C–Shell uses the path variable instead of the PATH environment variable used by the Bourne shell. Line 1 of Example 2C–1 shows how to use the **set** command to assign a value to the *path* variable.

The first directory in **path** is the current working directory, represented by the period (.). The second directory is named *.bin* and is a subdirectory of your home directory. In C–Shell, a tilde (~) represents your home directory. The last two directories in path, */bin* and */usr/bin*, contain the UTek commands.

When you create a C–Shell (by logging in, executing a C–Shell file, or starting up a subshell), *path* is given a default value of:

. /bin /usr/bin

The if Statement (lines 2, 8)

Line 2 of Example 2C-1 is the C-Shell **if** statement. If the expression in the parentheses is true, all the commands down to the **endif** (line 8) are executed.

The expression $?*prompt* is true if the *prompt* variable (the string that the C-Shell uses for its prompt) is set. This is one way of testing for an interactive C-Shell.

The history Variable (line 3)

The C-Shell can store commands you type in a *history list*, so you can reenter them later. To make the C-Shell create a history list, you must set the *history* variable to tell the C-Shell how many previously-executed commands to save in this list.

Line 3 of Example 2C-1 tells the C-Shell to remember the last 29 commands entered.

To reenter a command from the history list type an exclamation mark (!) followed by the number in the history list of the command you want to execute.

The mail Variable (line 4)

C-Shell uses the *mail* variable to store the name of the file that receives your mail. Line 4 tells the C-Shell to check the directory */usr/spool/mail/$USER* (your *system mailbox*) every 60 seconds for new mail.

Setting Environment Variables (lines 5-7)

To assign a value to an environment variable use the **setenv** command. Lines 5-7 set the EDIT, MORE, and SEDIT environment variables. These variables have the same function under the C-Shell as under the Bourne Shell.

Creating Aliases (lines 9-11)

The **alias** command lets you abbreviate commands

Line 9 of Example 2C–1 changes the **more** command so that each time you enter **more**, **more –u –f** is executed.

The exclamation mark followed by the asterisk (!*) is replaced by any other command line arguments you enter to **more**. (The exclamation mark must be preceded by a backslash (\) to quote its special meaning to the C–Shell.)

Line 10 lets you enter the **history** command by typing **hi**. Line 11 passes the output of **who** through **sort** and **more**.

A Sample .login File

The *.login* file contains commands to execute when you first log into UTek. Example 2C–2 shows a sample *.login* file.

```
1  set noglob
2  eval `tset -s -Q -m :?display -m dialup:?vt100 -m network:?display`
3  stty crt susp ^Z dsusp ^Y rprnt ^R flush ^O werase ^W \
4          lnext ^V intr ^? stop ^s start ^q
5  switch($TERM)
6  case aaa:
7          set history=35
8          ${HOME}/.bin/setup.aaa
9          breaksw
10 case vt100:
11          set history=30
12          ${HOME}/.bin/setup.aaa
13          breaksw
14 default:
15          set history=23
16 endsw
17 set prompt="`hostname` \!"
```

Example 2C–2. A Sample .login File.

The noglob Variable (line 1)

Line 1 of Example 2C–2 sets the Boolean C–Shell variable *noglob*. If *noglob* is set, the C–Shell doesn't try to expand special characters into filenames when you enter them as arguments to a command.

Setting Up your Terminal (lines 2-4)

Use the **eval** and **tset** commands to set up your terminal. The **tset** command generates commands that set up your terminal, and **eval** executes them. See the *UTek Command Reference, tset(1)* for details on setting up your terminal.

Lines 3 and 4 call **stty** to set options on your terminal. The first argument, *crt*, tells UTek to set options for a CRT (video display terminal). The rest of the arguments define the functions of certain control characters. See the *UTek Command Reference, stty(1)* and *tty(4)* for details.

The switch Command (lines 5-16)

The C–Shell **switch** statement is similar to the Bourne Shell **case** command. Line 5 of Example 2C–2 shows the **switch** command. The string in parentheses, in this case the value of the $TERM environment variable, is successively matched against the strings in the **case** statements (lines 6 and 10). If a match is found, the commands between the **case** command and the **breaksw** command (lines 9 and 13) are executed. If no match is found, the commands between the default label (line 14) and its **breaksw** command are executed.

In this example, the length of the history list is altered, depending on the type of terminal used (lines 7, 11, 15). This ensures that the entire history list fits on your terminal screen.

Line 8 of Example 2C–2 calls a program that sets up an Ann Arbor Ambassador (abbreviated *aaa*) terminal, if you are on that type of terminal. Line 12 sets up a Digital Equipment Corporation VT100 (abbreviated *vt100*), if you are on that type of terminal. Note these calls are to hypothetical programs in your *.bin* directory; you would need to create these programs.

The prompt Variable (line 17)

The *prompt* variable contains the string that the C–Shell uses for its prompt. The default value of *prompt* is a percent sign (%).

Line 17 sets the prompt to the output of the **hostname** command, followed by the number of this command in the history list, which is substituted for the exclamation mark (!). The exclamation mark must be preceded by a backslash (\) to quote its special meaning to the C–Shell. See the topic *History List* for more information on history event numbers.

Logging Off

Each time you log off the system, the C–Shell reads the *.logout* file for commands to be executed as you log out. These commands may include cleaning up temporary files, clearing your screen, or other administrative details.

Invoking the C–Shell

The first line of a C–Shell program contains the following:

```
#!/bin/csh
```

This invokes a C–Shell to run the program.

On this same line you can also give several options to the shell. Following are the most frequently used options. For a complete list see *UTek Command Reference, csh(1).*

—e The shell exits if any command terminates abnormally or returns a non–zero exit status.

—s Command input is taken from the standard input.

—v Sets the *verbose* variable so that command input is echoed after history substitution.

—x Sets the *echo* variable so that commands are echoed immediately before execution.

—V Sets the *verbose* variable, even before the *.cshrc* file is executed.

—X Sets the *echo* variable, even before the *.cshrc* is executed.

After the shell processes these options, and if the **—c**, **—i**, **—s** or **—t** were not given, the first argument is taken as the name of a file from which commands are executed.

Shell Procedures

You can use the shell to read and execute commands contained in a file. For example, this command calls the shell to read commands from the file *filename*.

csh *filename [arguments]*

Such a file is called a *shell program*, a *command procedure*, or a *shell procedure*. Arguments may be supplied with the command. In the file, arguments are referenced using the *positional parameters* $1, $2, and so on. For example, if the file *wg* contains:

```
who ¦ grep $1
```

then the call:

```
csh wg fred
```

is equivalent to the command:

```
who ¦ grep fred
```

When a UTek system file has the execute (x) attribute, you can execute a file directly, without invoking a special shell on the command line. To make a file executable, you can use the **chmod** command. For example, to make the file *wg* executable enter:

```
chmod +x wg
```

This command is equivalent to entering:

csh wg fred

This feature lets you use shell procedures and programs interchangeably. In either case, the shell creates a new process to execute the command.

As well as providing names for the positional parameters, the number of positional parameters when you invoke the shell is available as $#. The name of the file being executed is available as $0.

A special shell parameter $* is used to substitute for all positional parameters except $0. A typical use of this is to provide some default arguments.

Control Flow— for

A frequent use of shell procedures is to loop through the arguments ($1, $2, ...) executing commands once for each argument. An example of such a procedure is **tel**, which searches the file */usr/lib/telnos*. The file *telnos* contains lines of the form:

fred mh0123
bert mh0789

The text of **tel** is:

```
for i
do
   grep $i /usr/lib/telnos
done
```

The following command prints those lines in */usr/lib/telnos* that contain the string *fred*:

tel fred

The following command prints those lines containing *fred*, followed by those lines containing *bert*.

tel fred bert

The **for** loop notation is recognized by the shell. It has the general form:

for *name* **in** *word1 word2*
do
  command-list
done

A *command-list* is a sequence of one or more simple commands, separated or terminated by a newline character or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. A *name* is a shell variable that is set to the words *word1 word2 word3*, etc., each time the *command-list* following **do** is executed. If you eliminate **in** *word1 word2*, then the loop is executed once for each positional parameter; that is, **in** $* is assumed.

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

for i do >$i; done

Notice that a semicolon (or a newline) is required before **done**. So the command:

create alpha beta

ensures that two empty files, *alpha* and *beta*, are created, or emptied if they already exist. When you redirect the output of a file, the **create** command is not necessary. You can use the notation *>filename* on its own to create or clear the contents of a file.

Control Flow — if

A general conditional branch is also available:

if (*expressions*) **then**
  command-list
endif

This loop tests the value returned by the last simple command following **if**.

You can also follow this form of the **if** command with **else–if** pairs:

if (*expression*) **then**
  command-list
else if (*expression*) **then**
  command-list
else
  command-list
endif

There exists a second form of the **if** command that is much more restricted, but simpler to use:

if (*expression*) *command*

In this case, if the *expression* is true, the command is executed. The command must be a single command, not a pipeline, command list, or parenthesized command list.

NOTE

The command redirects input or output even if expression is false.

Following is an example program that illustrates the use of both kinds of **if** statements. This program copies C programs in the specified list to the directory *~/backup* if they differ from those already in *~/backup*.

```
set noglob
foreach i ($argv)

   if ($i ~ *.c) continue

   if ( -r ~/backup/$i:t) then
      echo $i:t not in backup... not cp\'ed
      continue
   endif

   cmp -s $i ~/backup/$i:t

   if ($status !=0) then
      echo new backup of $i
      cp $i ~/backup/$i:t
   endif
end
```

This program uses the **foreach** command, which executes the commands between **foreach** and **end** for the successive values of *i*. After each **foreach** loop, *i* has the value of the last iteration of the loop.

The variable *noglob* is set in this program to prevent filename expansion of the members of *argv*. This is a good idea since the arguments to this shell script contain metacharacters that can be used for filename expansion.

Control Flow - while and switch

The C–Shell provides two control structures, **while** and **switch**, similar to those of the C programming language.

The while command has the following form:

while (*expression*)
    commands
end

The **while** command is a loop that repeats once per line. If *expression* is true, each of the *commands* of the loop is executed. Then the *expression* is tested again, and if it is still true *commands* are executed. When the *expression* becomes false, the loop ends.

The **switch** command lets you structure multiple branches into a shell program without using the **if... else** form. It evaluates a word and tests whether the value matches one of a number of **cases**. It has the general form:

```
switch (word)
case string1:
    commands
    breaksw
...
switch (word)
case string2:
    commands
    breaksw
default:
    commands
    breaksw
endsw
```

The **switch** command evaluates the *word* in parentheses, and compares its value to all the cases. If a case matches the value of *word*, execution starts at that case. The command **breaksw** prevents the command from moving to the next case. Without it, commands associated with every **case** following the first match would be executed. The **default** case is executed if none of the other cases is satisfied. If no cases match, and the **default** case is not present, no commands are executed. Each of the **cases** that you specify must be different, but the **case** and **default** commands can occur in any order.

Although there are frequently better ways to structure a shell program, the C–Shell provides a **goto** command. It has the general form:

```
loop1
    commands
    goto loop2
```

After *loop1* and *commands* execute, control of the program execution moves to *loop2*.

Other C–Shell Commands

The C–Shell has many other commands built into the shell. Many of the most important commands are described in this section, but for a complete list see the *UTek Command Reference, csh(1).*

Supplying Input to the Shell

Unlike other UNIX shells, commands run within a C–Shell program receive their input from the shell that is running the script. Notation that redirects input and output and uses pipelines can be used to give input to the shell, but to avoid putting command input into a file, you need the special 'EOF' notation.

Consider the following example. This shell program uses the **ed** text editor to delete leading blanks from the lines in each file:

```
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation << 'EOF' begins the input for the **ed** command, and the notation 'EOF' on a line by itself terminates that input. The EOF terminator is placed in quotes so that the shell does not perform variable substitution on intervening lines.

Note that the Bourne shell notation <<–'EOF' is not available in the C–Shell.

Command Substitution

Another way you can supply input to the shell is called *command substitution.* This lets you take the output of a UTek command, and read it as shell input. To do this you enclose the appropriate UTek command in left single quotes. For example:

'pwd'

When the shell reads this, it substitutes the pathname of the current working directory for the string 'pwd'.

The Bourne shell lets you do nested command substitution, but the C–Shell does not. For example, in the Bourne shell you can set a variable *foo* like this:

foo = " ` grep foo \ ` cat list \ "

This example of nested substitution does not work in the C–Shell.

You also cannot perform command substitution on commands that are aliases.

Reading from the Terminal

The C–Shell provides a special symbol that signals the shell to receive input from the terminal. The symbol is $< For example:

```
%foo=$<
hello
%echo $foo
hello
```

In this example, you entered the first *hello* to set the value of the variable *foo*, then displayed that value on the screen by echoing the value of *foo*.

Catching Interrupts

When you have a shell program that creates temporary files, and the shell receives an interrupt signal, you frequently want to remove the temporary files that were created.

To catch the interruption of a shell script, enter the following line at the beginning:

onintr *label*

In this command, *label* is a label somewhere in your program that contains commands to delete the necessary files, ending with the **exit** command. Then when your shell program receives an interrupt, execution goes to the label you defined at the beginning to remove temporary files.

The History List

The shell saves a numbered list of the previous commands you executed. You can then use this numbered list to quickly execute the same command again, or you can repeat arguments or a previous command. The number of commands you choose to save is set in the history variable. You can set the history variable in the *.cshrc* file, as discussed earlier in this section. To find out what is stored in your history list, simply type **history**.

History substitutions begin with an exclamation mark (!). For example, to execute the command numbered 11 on the history list enter:

!11

The terminal displays the contents of the command before it is executed, so that you can interrupt the command if it is not the one you want.

The following example shows a history list obtained using the **history** command. It shows various ways of referring to a command:

```
 9 write michael
10 ex write.c
11 cat oldwrite.c
12 diff*write.c
```

In this history list, each number is associated with a command. The *current event number* is 13, because that is the next number on the list. Suppose that you want to execute command 11 again; there are several ways of using the **history** mechanism to reexecute the command. You can enter:

```
!11
```

Or you can address the command relative to the current event number of 13:

```
!-2
```

You can also identify a history event by matching a prefix of the word, or by matching a string contained in the word. For example, both these commands would match event 11:

```
!ca
!?oldwr?
```

The reference !! also refers to the previous command. So in the case of the above history list, **!!** executes event 12.

You can also select particular words from an event in the history list. To do this, enter:

event-number:*word-designator*

The first word of a command line is numbered 0, the second is numbered 1, and so on. For example:

ls -a *filename*

In this example, **ls** is the first word (numbered 0), **-a** is the second word (numbered 1), and *filename* is the third word (numbered 2). Optional word designators include:

0	first word (command)
n	argument *n*
^	first argument (*n* = 1)
$	last argument
%	word immediately preceding ?*s*? search
x-y	range of words
-y	words 0–*y*
*	first to last argument; nothing if only 1 word in event
*x**	word *x* to last argument
x-	word *x* to next–to–last argument

You can also give a history reference without referring to the event number. In this case, it is assumed that the reference is to a previous command. For example, the following command uses the last argument of the previous command:

!$

If the word designator begins with ^, $, *, —, or %, you can omit the colon that separates the event specification from the word designator.

After the word designator, you can also enter another colon and a modifier. Following are the available modifiers:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing filename suffix *.xxx*, leaving the root name.
e	Remove all the pathname components except a trailing filename suffix.
s/*x*/*y*	Substitute *x* for *y*.
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Repeat the previous substitution globally.
p	Print the new command, but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs, and newlines.

The **s** substitution modifier accepts another delimiter in place of a forward slash. Precede the new delimiter with a backslash. An ampersand character (&) in the right hand side of the substitution is replaced by the text from the left. If you do not enter anything for the left side of the substitution, the C–Shell uses a string from a previous substitution, or from the last forward or backward editor scan.

A special abbreviation, ^, is available to substitute something for the first argument of the previous command. The character ^ is the equivalent to !:s^. For example, suppose that you misspell the first argument of a command as "lb" instead of "lib." You can correct it by entering:

^lb^lib<RETURN>

This command executes the previous one, except that the first argument is changed from *lb* to *lib*. So if you misspell the first argument of a command you can easily correct it.

Following is an example of word designator and modifier use. Let's say that the history list event number 12 is the command **cc –O /usr/prog/program.c**. The following command changes history event number 12, by choosing all it's arguments (word 0 – last argument), but then modifying the last argument so that only it's trailing pathname component (program.c) is executed:

!12:0–$:t

Effectively, this command does the same thing as history event 12, but the **cc** command doesn't have to look beyond the current directory for program.c.

Alias Substitution

Following any history substitutions, the shell does *alias* substitutions. An alias is a way of simplifying the commands you need to type. An alias can be a shorter notation for a command, a command and its arguments, or several commands put together. For example, you can alias the **clear** command to **c** by entering:

```
alias clear c
```

Another example is to have **ls** always show the size of files using the **–s** option. Enter:

```
alias ls ls –s
```

Now when you enter **ls**, the command **ls –s** is executed.

To use an alias during a particular login session, you can enter the **alias** command at your C–Shell prompt. But to alias a command permanently, you need to enter it into your *.cshrc* file. To put an alias in your *.cshrc* file, simply enter it as shown above. To alias more than one command, put each **alias** command on a different line. For an example of aliasing in a *.cshrc* file, see the earlier topic *A Sample .Cshrc File.*

Frequently, to alias more complicated commands, you need to use quoting mechanisms. Consider the following example:

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

In this example, the entire alias definition is quoted in single quotes. This prevents most substitutions within this string and the recognition of the semicolon as a metacharacter. In this case the semicolon is meant to separate two different UTek commands. So that the ! is not interpreted as a history specification when you type the command in, it is quoted with a backslash (\).

NOTE

Because the shell reads the .cshrc file each time you invoke a new shell, a large number of alias commands can make each invocation very slow. Limit your alias commands to a number that you can easily remember, and if invocation of a new shell seems unreasonably slow experiment with reducing the number of aliases in your .cshrc file.

Job Control

Each time that you enter a command, a job is created. For example, when you enter more than one command on a line, separated by semicolons or in a pipeline sequence, the shell creates a single job that consists of these commands put together.

Normally when you enter a command, you wait for the prompt to return indicating the command is done. But many commands, such as text formatting, require more processing time. In this case you can run a command in the *background.* The shell does not wait for a job in the background to complete execution, but immediately returns the prompt so that you can continue with other work. To place a job in the background, type an ampersand (&) at the end of the commands. You can place one or more jobs in the background, but they always run at lower priority than jobs that are not in the background. The more jobs you put in the background, the more slowly they run. If there is no space available on the system for more processes, the message *No more processes* displays.

Although many jobs can run in the background, only one job can be in the *foreground.* A job in the *foreground* is one where you enter the commands and wait for the prompt to return. The shell reads and executes foreground jobs one at a time until each one is finished, but these jobs run at a higher priority than background jobs.

When you put a job in the background, the shell assigns it a *job number.* After you enter the commands, followed by an ampersand, the job number and the process id display. For example:

```
nroff -mm file1  > file1.out
[1] 503
```

This means that the above nroff job has job number 1, and a process id of 503. When a background job is done, a message displays. Using the previous example:

```
[1] - Done    nroff -mm file1>file1.out
```

If a background job terminates normally, a message saying "Done" displays. Otherwise, it says something like *Killed.* This job termination message displays only when you return to the shell prompt. If you want to be notified immediately when a background job terminates, set the *notify* variable in your *.cshrc* file.

In addition to being in the foreground or background, a job can be suspended by typing <CTRL–Z>. You can suspend a job, then manipulate it between the background and the foreground. You can start a job in the foreground, type <CTRL–Z> to suspend it, then use the **bg** command to put it in the background. For example:

```
%nroff -mm file1
<CTRL–Z>
Stopped
%bg
[1] nroff –mm file1
```

The shell displays the message that says *Stopped*, and returns the prompt. Then you enter **bg** to put the command in the background, just as you would use an ampersand to put a job in the background. It is very convenient to suspend foreground jobs when you need to execute other commands in the middle of a job.

To suspend a job that is running in the background, use the **stop** command. (<CTRL–Z> does not suspend background jobs). For example:

```
%sort file1&
[1] 2435
%stop %1
[1] Stopped (signal)   sort file1
%
```

In this example the **sort** command is placed in the background, then suspended using the **stop** command. The **stop** command takes references to job numbers as its arguments, and the percent sign (%) refers to job numbers. So the stop command is stopping job number 1, and displays a message informing you that the job is stopped. Once the job is stopped, you can use the **fg** command to bring it into the foreground. For example:

```
%fg
sort file1
```

The **fg** command brings the job into the foreground, displays its commands, and waits for the prompt to return.

There are many ways of referring to job numbers. The percent sign introduces a job number, for example %1 is job number 1. Entering only a job number brings it into the foreground, so the command **%1** is equivalent to **fg %1**. Similarly, **%1 &** puts job number 1 in the background. So long as they are unique, you can also refer to jobs by prefixes of their first string, or by identifying a string in the middle of the job. For example, **%ex** puts in the foreground a previous job beginning with **ex**. To bring a job containing *string* into the foreground, enter **%?***string*.

The shell keeps track of what job is current and what is a previous job. In output pertaining to jobs, the current job is marked with an addition sign (+), and the previous job is marked with a subtraction sign (—). So the abbreviation %+ refers to the current job, and the abbreviation %— refers to the previous job. The notation %% is also a reference to the current job.

To check on the status of all the jobs you are running, enter **jobs**. This displays the job number, what jobs are current and previous, and the commands they contain. For example:

```
%jobs
[1] - Running     cat temp>foo
[2]   Running     ls -s ¦ sort -n > myfile
[3] + Stopped     comp
```

With the **-l** option the **jobs** command also prints the process id numbers of each job.

To stop a job that is suspended or running in the background, use the **kill** command. For example:

kill %2

This command terminates job number 2. The **kill** command accepts multiple arguments, so you can kill several jobs at once. If you find out the process numbers associated with particular jobs, you can use the command **kill** *process-number* to terminate the job.

The **notify** command (not the *notify* variable) informs you immediately when a job terminates, instead of waiting for the next shell prompt. Enter:

notify %2

This command tells you immediately when job number 2 terminates, instead of waiting for a shell prompt. Without arguments, the **notify** command refers to the current job.

When a job running in the background tries to read input from the terminal, it is automatically stopped. When the job stops, you can bring it into the foreground using the **fg** command and enter commands from the terminal.

When you try to leave the shell while jobs are stopped, the message *You have stopped jobs* displays. You can use the **jobs** commands to see what they are. If you try to exit the shell again, no warning displays and the suspended jobs are killed.

Job Control Using Remote Login

The local area network provided with your 6000 Series workstation lets you access remote host machines, using the **rlogin** command. If you type **rlogin** *hostname*, and your system administrator has set up the network correctly, you can access a remote host computer. For more information on setting up a local network see your *System Administration* manual.

When you remotely log in to another host, you can begin running a job on the remote host, and then pause the remote **rlogin** job and return to your home machine. The actual job that you began on the remote machine continues to run; only the **rlogin** job is paused.

The following example illustrates the process of remotely logging in to another machine, starting an **nroff** job there, and returning to the original machine. The home (original) machine is called **boris**, and the remote machine is called **natasha**. To make it easier to identify which machine you are on, we have changed the default C–Shell prompt to *machine-name*>. So the prompt **boris**> means that you are on the machine named **boris**. Notice the command that stops the **rlogin** job, ~<CTRL–Z> <RETURN>:

```
boris>rlogin natasha
(system messages display)
natasha>nroff -mm file1 <RETURN>
(no prompt displays)
~<CTRL-Z> <RETURN>
boris>
```

Now that you have returned to the home machine, **boris**, you can use the **jobs** command to see what jobs are running. The **nroff** command is still running on **natasha**, and the **rlogin** job called is actually the job that is paused:

```
boris>jobs
[1] +   rlogin  Stopped
```

You can pause a job on a host machine using ~<CTRL–Z> <RETURN>, return to the home machine to do work there, then return to the paused job. Several commands return you to the job paused on the remote machine:

fg %*job-number*
%rlogin
%*job-number*

Using this ability to remotely log in to another machine and process jobs there, while still accomplishing work on your home machine, lets you run more jobs at one time and distribute your use of computer resources in a more balanced way.

C–Shell Variables and Variable Substitution

The C–Shell maintains a list of variables that you can use in shell programming, and that the shell refers to whenever it is invoked. Some variables are set by the shell permanently, and you can change the value of other variables the shell always recognizes. In addition, you can define your own variables for use in shell programs. To set the value of a shell variable, use the **set** command. The general form of the set command is:

set *name = value*

The C–Shell has a variable called *path*, and you can set the value of path yourself. The variable defines a default set or directories where the C–Shell looks to find executable commands. For example:

```
path=(. /usr/bin /bin /usr/local)
```

This command sets the *path* variable so that the C–Shell searches the directories */usr/bin*, */bin*, and */usr/local* for commands to execute. To display the values of all the variables you have set in the current C–Shell, enter **set** without any arguments.

Shell variables are always strings of characters, but some are defined differently. Some variables have specific values that you define, or that the shell defines, while others are *toggled*, or switched from one state to another. If a variable is toggled, the shell does not care what its value is, only whether it is set or not. An example of a toggled variable is the *noclobber* variable that does not allow you to overwrite a file that already exists. You can use the **set** command to set this variable:

```
set noclobber
```

If the variable is already set and you want to unset it enter:

```
unset noclobber
```

Other operations treat variables numerically. The @ command preceding a variable lets you do numeric calculations. Although variable values consist of zero or more strings, variables with numeric values assign a value of zero to the null string and ignore any strings beyond the first.

Variable substitution in the shell program occurs after the history and alias substitutions, and after the input line is broken up by the shell. The C–Shell also recognizes redirections of input and output before it does variable substitution, so any variable involved in the redirection of input and output is expanded before other variables.

A dollar sign ($) preceding the variable name tells the C-Shell to substitute the value of the variable for its name. You can prevent variable substitution by preceding the dollar sign with a backslash (\), except within double quotes ('') where variable substitution *always* occurs, and within right single quotes (') where variable substitution *never* occurs.

Following variable substitution, the resulting value is subject to command and filename substitution. See the previous topic *Command Substitution*.

Following is a list of the various ways you can introduce variables into the shell input:

$*name* ${*name*}	The value of the variable *name* in words, each separated by a blank. Braces insulate *name* from characters that follow it and would otherwise be part of *name*. If *name* is not a shell variable, but set in the environment, its value in the environment is returned.
$*name*[*selector*] ${*name*[*selector*]}	Select only some words from the value of *name*. *Selector* is a single number or a range of numbers separated by a hyphen (–) that represents the variable word(s) you want to select. The first word of a variable value is numbered 1, so if you omit the first number of a range it defaults to 1. If the last member of a range is omitted, it defaults to the number of words in the variable. To select all words, use an asterisk (*) as *selector*.
$#*name* ${#*name*}	The number of words in the variable. It is useful to later substitute this number in a *selector*.
$0	The name of the file where the shell reads command input.
$*number* ${*number*}	Selects a certain number of the arguments given to the shell. It is equivalent to $argv[*number*], where argv contains all the arguments given to the shell.
$*	Selects all arguments given to the shell. It is equivalent to $argv[*].

Some of the same modifiers that you use to select particular words from a history event can be used to select words from a variable substitution. You can enter the following modifiers following the introduction of a variable:

:h head of a pathname

:t trailing component of a pathname

:r root of a filename

:q quote the substituted words

:x quote words, but break them at blanks, tabs, newlines

You can also combine the :h, :t, and :r modifiers with the global modifier g(make all the substitutions on a line), as :gh, :gt, and :gr. If you use braces in the command you must place the modifiers before the closing brace.

You cannot apply modifiers to the following variable substitutions:

$?*name*
${?*name*} If *name* is set, substitutes the string *1*. Otherwise, substitutes the string *0*.

$?0 If the current input filename is known, substitutes the string *1*. Otherwise, substitutes *0*.

$$ The decimal process number of the parent shell.

$< A line from the standard input. Use to read keyboard input in a shell program.

Predefined Shell Variables

The following variables have a special, predefined meaning to the shell. Avoid redefining variables of the same name in shell procedures that you write. Of these variables, *argv, cwd, home, path, prompt, shell,* and *status* are always set by the shell; you can change the settings for others.

Some things that are defined as environment variables in the regular shell (see Section 2B) become C–Shell variables. The C–Shell copies the environment variable USER into the variable *user*, TERM into *term*, PATH into *path*, and HOME into *home*. For information on setting these variables in your *.cshrc* file, see the earlier topic *A Sample .Cshrc File.*

To incorporate Bourne shell environment variables into the C–Shell, use the **setenv** command. This command has the form **setenv** *name value*, where *name* is the Bourne shell environment variable, and *value* gives the variable a new value. For example, you can set the Bourne shell environment variable EDIT to **vi** by entering **setenv EDIT vi**.

argv Arguments to the shell. Positional parameters are substituted according to this variable

cdpath	A list of alternate directories searched to find subdirectories in **chdir** commands.
complete	Lets you enter only enough characters to make a filename or a command unique. Then you press <ESC>, and the C-Shell automatically fills in the rest of the filename or command name. You then press <RETURN> to enter the command.
cwd	The full pathname of the current directory.
echo	Set when you enter the **-x** option on the command line. Echoes each command and its arguments before executing it.
hardpaths	Instead of creating a path using relative pathnames, this variable substitutes the absolute pathname.
histchars	Changes the characters used in history substitution. It is a string value, and the first character of its string replaces the default character !. The second character of its string replaces the character ^ in quick substitutions.
history	Controls the size of a history list with a numeric value.
home	The home directory of the user that invoked the shell, initialized from the *environment* variable.
ignoreeof	The shell ignores an end-of-file from terminal input. This prevents the shell from accidentally being killed by <CTRL-D>.
list	Lets you enter partial filenames or directory names as command arguments, then press <CTRL-D> to list all the files and directories that match the characters you have entered thus far.
listpathnum	Used in conjunction with the *list* variable. When you press <CTRL-D> to list paths for commands, this also includes the *path element number*, that is, the first, second, etc. element in your *path* variable.
mail	Defines the files where the shell checks for mail. If the first word is numeric, it specifies, in seconds, how frequently the shell checks to see if you have mail.
noclobber	Checks to see that redirected output does not write over an existing file, and that files to which you append output using >> exist.
noglob	Inhibits filename expansion.
nonomatch	When this variable is set, no error occurs if a filename expansion does not match existing files. Instead, the primitive pattern is returned.
notify	Notifies you immediately that a job is complete, instead of waiting for a prompt to display a message.

path	Specifies the directories where the shell looks for executable commands.
prompt	The string that displays before reading a command from the terminal. By default, this string is %. If a ! is in the string, it is replaced by the current history event number.
savehist	Controls the number of entries in the history list that are saved in your *.history* file from one login to the next. It has a numeric value.
shell	The file where the shell resides. In shells that use a forking process this interprets files that have execute bits set, but are not executable by the system.
status	The status returned by the last command. If it terminated abnormally, 0200 is added to the status. Built-in commands that fail return exit status 1; all other built-in commands set the status to 0.
time	Controls the automatic timing of commands.
verbose	This variable is set by the **-v** command line option. After each history substitution, the words of each command display.

Expressions

To write useful shell scripts, you need to evaluate expressions in the shell based on the values of variables. The C-Shell provides all the arithmetic operations of the C programming language, with the same precedence. For example, the operators = = and =! compare strings, while && and ¦ ¦ implement the Boolean and/or operations. The special operators =~ and !~ are similar to = = and ! =, except that the string on the right side can have pattern-matching characters like * and ?.

The C-Shell also provides operators that let you see whether a file exists, if it is readable and writable, and so on. These operators take the form:

-c **filename**

In this operation *c* can be one of the following:

r read permission

w write permission

x execute permission

e existence

o ownership

z zero-length

f regular file

d directory

You can also use braces to test whether a command terminates normally. This form returns 1 if the command executes normally, or zero if the command terminates abnormally. For example:

{*command*}

These integer values are the opposite of exit status, which is zero for a normal execution and nonzero otherwise.

New C-Shell Features

The UTek C-Shell contains all the features of the Berkeley 4.2 BSD C-Shell, as well as the extensions described in the following paragraphs.

File Name Completion

When you type a command, you can use abbreviations for file names. First, set the **complete** variable by typing:

set complete

When you enter a filename as an argument to a command, type as many characters as you need to make the filename unique. Then press the <ESC> key. The C-Shell automatically fills in the rest of the filename, and you can press <RETURN> to enter the command.

Following is an example of filename completion.

```
% ls
DSC.OLD   bin       cmd      lib    memos
DSC.NEW   chaosnet  cmtest   mail   netnews
bench     class     dev      mbox   news
% ls ch<ESC>
% ls chaosnet          (The cursor remains at the end of this line)
```

Press <RETURN> to enter the command.

File and Directory List

When you enter a command, you may want to know what filenames match what you have typed so far. First, set the **list** variable by typing:

```
set list
```

Then, when you enter a filename or directory name as an argument to a command, press <CTRL-D> to list all matching files and directories. Below is an example of the file and directory list.

```
% ls
DSC.OLD   bin        cmd      lib    memos
DSC.NEW   chaosnet   cmtest   mail   netnews
bench     class      dev      mbox   news
% ls c<CTRL-D>
chaosnet    class     cmd     cmtest
% ls c          (The cursor remains at the end of this line)
```

After you press <CTRL-D>, the filenames that match **c** display. The shell prompt and the fragment of a command that you entered earlier display on the next line, and the cursor remains there. This gives you the opportunity to complete the unique filename before pressing <RETURN> to enter the command.

Command Name Recognition

You can use the completion and list features when entering command names, as well as filenames and directory names. Below are examples of command completion:

```
% pass<ESC>
% passwd          (The cursor remains at the end of this line)
```

and command listing:

```
% pas<CTRL-D>
passwd    paste
% pas          (The cursor remains at the end of this line)
```

Automatic Logout

With this feature, the C–Shell logs you out if your terminal is idle for a specified period of time. You can set the *autologout* variable; for example:

```
% set autologout=60
```

This variable waits 60 minutes before logging you off. You can turn this feature off by typing:

```
unset autologout
```

When you log in, this feature is always unset.

Terminal Checking

If your terminal is left in *raw, cbreak*, or *noecho* mode (in other words, it is unusable) by a command, the C–Shell automatically restores it to a usable state.

Saving Your History List

The C–Shell can store your history list between login sessions. The list is stored in a file named *.history* when you log out and is restored the next time you log in. To set this feature, specify the number of commands you want the C–Shell to restore. For example:

```
set savehist=30
```

causes the C–Shell to store the last 30 commands you entered.

The UTek System Implementation

Introduction

The facilities available to a UTek user process are divided into two parts: kernel facilities directly implemented by code running in the UTek operating system, and system facilities implemented by the system or in cooperation with a server process. The first part of this section deals with kernel facilities, while the second part deals with system facilities.

The kernel facilities define the *UTek virtual machine*, in which each process runs. Like real machines, this virtual machine has memory management, an interrupt facility, timers, and counters.

The second part of this section deals with system facilities. The UTek virtual machine allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, parts of the descriptor capability are built into the operating system, while other parts are implemented as server processes on other machines.

Throughout this section reference is made to the *privileged user* or *superuser*. This person logs in as root, and can move throughout the system with special access to all its facilities. Avoid logging in as the superuser because mistakes you make can have drastic effects on the system.

Kernel Facilities

Processes and Protection

Host and Process Identifiers

Each workstation has a 32-bit host id associated with it, and a host name of up to 255 characters. These are set by a privileged user, and returned by the system calls:

```
sethostid(hostid)
int hostid;

hostid = gethostid();
result int hostid;

sethostname(name, namelen)
char *name; int namelen;

gethostname(name, namelen)
result char *name; result int namelen;
```

Each host runs a set of *processes*. Each process is largely independent of other processes, with its own protection, address space, timers, and an independent set of references to system- or user-implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result long pid;
```

On each workstation this identifier is unique. So if you have several workstations in a local area network, hostid/process id pairs are unique.

Process Creation and Termination

A new process is created by making a duplicate of itself:

```
pid = fork();
result int pid;
```

The **fork** call returns twice, once in the original parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. Every process in the system runs in this kinds of hierarchy.

A process terminates using an **exit** call:

```
exit(status)
int status;
```

This call returns 8 bits of information on exit status to its parent.

Because the **fork** system call returns only the process id, the **wait** system call is necessary to keep the parent process awaiting the return of a child process. When a child process exits or terminates abnormally, the parent process receives information about any event that caused termination of the process. This information is returned as *status* in the **wait** system call:

```
#include <sys/wait.h>

pid = wait(status);
result int pid; result union wait *status;
```

A second system call, **wait3** provides an alternative interface for programs that must not block when collecting the status of child processes:

```
pid = wait3(status, options, usage);
result int pid; result union waitstatus *status;
int options; result struct rusage *rusage;
```

A process can overlay itself with the memory image of another process, passing the newly–created process a set of parameters, using the call:

```
execve(path, argv, envp)
char *path, *argv[], *envp[];
```

The specified *path* must be a file, either a binary executable file or a file of data for an interpreter.

User and Group Identification

Two user identifications are associated with each process in the system: a *real user id* and an *effective user id*. Each process also has a *real accounting group id* and an *effective accounting group id*, as well as a set of *access group id's*. Each process can be in several different access groups. The maximum concurrent number of access groups is defined by the constant NGROUPS in the file <sys/param.h>, whose value is guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by the following system calls. The first returns the real user id, and the second returns the effective user id:

```
uid = getuid();
result int uid;

euid = geteuid();
result int euid;
```

The real and effective accounting group ids are returned by the following calls. The first returns the real id, and the second the effective id:

```
gid = getgid();
result int gid;

egid = getegid();
result int egid;
```

The access group id is returned by the **getgroups** call:

```
#include <sys/param.h>

getgroups(ngroups, gidset)
result int *ngroupd, *gidset;
```

The system assigns the user ID and the group ID at login time, using the calls **setreuid, setregid,** and **setgroups**:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(ngroups,gidset);
int ngroups; *gidset;
```

The **setreuid** call sets both the real and effective user ID, while the **setregid** call sets both the read and effective group ID. Unless the caller is the superuser, the effective user ID and the effective group ID can only be changed to the current real user ID or real group ID. The **setgroups** call is restricted to the superuser.

Process Groups

Each process in the system is normally associated with a *process group*. The group of processes in a process group is referred to as a *job*, and can be manipulated by the shell and other high–level system software. The process group of a process is returned by the **getpgrp** call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

When a process is part of a process group, software interrupts that affect the single process also affect all the processes in a group. For example, a terminal has a process group, and to read from the terminal a process must be part of that process group. This terminal process group lets the terminal divide its resources among several different jobs.

The **setpgrp** call changes the process group of a process:

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

When a process is created, it is assigned the same process group as its parent process. It is also assigned a process id distinct from all processes and process groups. A normal (unprivileged) process can set its process group equal to its process id. A privileged process can set the process group of any process to any value.

Memory Management

This information to be supplied later.

Signals

Overview

The system defines a set of *signals* that can be delivered to a process. When a signal is delivered to a process, the current process context is saved, a new context is created, and any further occurrence of the same signal is blocked. A process can specify the *handler*, which determines what happens when the process receives a signal. See the later topic, *Signal Handlers*. A process can also specify that a signal is to be *blocked* or *ignored*, or a *default* action to take when the signal occurs.

Some signals cause a process to exit if they are not caught. This may be accompanied by the creation of a *core* image file that contains the current memory image of the process. You can use this image to debug the process.

When a signal is delivered to a process, you can manipulate the way it will be executed by its handler. You can specify a special signal stack, instead of the normal one, so that you can manipulate the signal without disturbing the normal stack.

All signals have the same priority. If multiple signals are pending for the same process, the order in which they are delivered to the process depends on how they are implemented. The *signal routines* execute the appropriate action when the process receives a signal. When a signal routine executes, the signal that caused them to be invoked is blocked, but other signals can occur. If you want to protect a piece of code from other signals, it can be protected against specific signals.

Signal Types

The system defines five signal types, in the file <signal.h>.

- hardware conditions
- software conditions
- input/output notification
- process control
- resource control

Hardware signals are derived from exceptional conditions that can occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently–assigned area of memory, and SIGBUS for accesses that violate memory protection constraints.

Software signals reflect interrupts initiated by the user: SIGINT for a normal interrupt, SIGQUIT for the quit signal (normally generates a core image), SIGHUP and SIGTERM that terminate processes gracefully due to a hang up or by user or program request, and SIGKILL, a termination signal that a process cannot catch or ignore. Other software signals indicate the expiration of interval timers (SIGALRM, SIGVTALRM, SIGPROF).

The SIGIO signal informs a process when input or output is possible, or when a *non-blocking* operation completes. A *non-blocking* operation does not tell the process when a descriptor can be accessed. A process can also request a SIGURG signal when an urgent condition arises.

Several signals can stop a process when sent to the process itself or a member of its process group. SITSTOP is a powerful stop signal that cannot be caught. Other stop signals, SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped.

A SIGCONT signal is sent to a process when it is continued from a stopped state. When a child process changes state, either by stopping or terminating, processes can receive notification from a SIGCHLD signal.

If you exceed resource limits, signals like SIGXCPU for the CPU time limit and SIGXFSZ for file size warn you that the limit has been reached.

Signal Handlers

The signal handler routine determines what happens when a signal is delivered. The signal handler can choose to ignore a signal, give the signal a default action (usually process termination), or run an interrupt routine that affects the process. The **sigvec** system call assigns handler addresses that specify an interrupt routine, a default action, or that a signal is ignored:

```
#include <signal.h>

sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;

struct sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_onstack;
};
```

If *vec* is nonzero, it specifies a handler routine *sv_handler* and mask *sv_mask* to be used when delivering the signal. The constants SIG_IGN and SIG_DEF as values for *sv_handler* cause a signal condition to be ignored or default. If *sv_onstack* is 1, the system delivers the signal to the process on a signal stack, instead of the normal run–time stack. If *ovec* is non–zero, the previous handling information for the signal is returned to the user.

When a signal condition arises for a process, the signal is added to a set of signals pending for that process. If the signal is not blocked by the process, it is delivered. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described in the next paragraph), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process resumes execution in the context before the delivery of the signal. To resume in a different context, the process must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the signal handler, or until you change it using the **sigblock**(2) or **sigsetmask**(2) system calls. This new signal mask is formed by taking the current signal mask, adding the signal to be delivered, and **or**'ing in the signal mask associated with the handler that is invoked.

The handler can be declared as follows:

```
handler(sig, code, scp);
int sig, code;
struct sigcontext *scp;
```

The parameter *sig* is the signal number, while *scp* is a pointer to the structure *sigcontext*. The *sigcontext* structure is defined in <signal.h>.

Sending Signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, sig)
int pid, sig;

kill pg(pgrp, sig)
int pgrp, sig;
```

The process sending the signal and receiving the signal must have the same effective user id, unless the process sending the signal is privileged.

Protecting Code from Signals

To protect a section of code against one or more signals, you can use a **sigblock** call to add a set of signals to the existing mask:

```
omask = sigblock(mask);
result int omask; int mask;
```

You can then restore the old mask with the **sigsetmask** call:

```
omask = sigsetmask(mask);
result int omask; int mask;
```

You can then use the **sigblock** call to read the current mask by specifying an empty *mask* parameter.

You can check conditions with some signals blocked, then pause to wait for a signal and restore the mask, using the **sigpause** call.

```
sigpause(sig);
int sigmask;
```

Signal Stacks

Programs that maintain complex or fixed–size stacks can use the **sigstack** call to provide a special signal stack:

```
sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;

struct sigstack {
  caddr_t  ss_sp;
  int      ss_onstack;
};
```

The stack is based at *ss_sp* for signal delivery, and the value *ss_onstack* indicates whether the process is currently on the signal stack.

When the system wants to deliver a signal to a process, ikt checks whether the process is on a signal stack. If not, the process is switched to the signal stack for delivery, and when the signal returns, the previous stack is restored.

If a process wants to take a non–local exit from the signal routine, or run code from the signal stack that uses a different stack, use the **sigstack** call to reset the signal stack.

Timers

Real Time

The system uses the calls **gettimeofday** and **settimeofday** to set and return the current Greenwich time and time zone:

```
#include <sys/time.h>

settimeofday(tp, tzp);
struct timeval *tp;
structg timezone *tzp;

gettimeofday(tp,tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

The timeval and timezone structures are defined in <sys/time.h.>:

```
struct timeval {
  long          tv_sec;          seconds since Jan.1, 1970
  long          tv_used;  microseconds since Jan. 1, 1970
};

struct timezone {
  int           tz_minuteswest;          minutes west of Greenwich
  int           tz_dsttime;              type of dst correction to apply
};
```

Interval Time

Each process is provided with three interval timers, defined in <sys/time.h>:

```
ITIMER_REAL     0
ITIMER_VIRTUAL  1
ITIMER_PROF     2
```

The ITIMER_REAL timer decrements in real time. It can be used by a library routine to maintain a wakeup service queue. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. When it expires, a SIGVTALRM signal is delivered.

THE ITIMER_PROF timer decrements both in process virtual time and system virtual time. It is designed to profile the execution of a process. A SIGPROF signal is delivered when it expires.

A timer value is defined by the itimerval structure:

```
struct itimerval {
  struct  timeval it_interval;
  struct  timeval it_value;
};
```

A timer is set or read by the **setitimer** and **getitimer** calls:

```
getitimer(which, value);
int which; result struct itimerval *value;

setitimer(which, value, ovalue);
int which; struct itimerval *value; result struct itimerval *ovalue;
```

The *ovalue* parameter specifies an optional structure to receive the previous contents of the interval timer. You can disable a timer by specifying a value of 0.

The time intervals measured by the system can only be as accurate as the resolution of the system clock. To find out the resolution of the clock load a very small value into a timer and read the timer back to see what value results.

To get profiled statistics on how much time is used by a particular process, use the **profil** system call:

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset scale;
```

Resource Controls

Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This favors interactive processes and processes that execute only for short periods. You can determine the priority assigned to a process, process group, or the processes of a particular user using the **getpriority** system call:

```
prio = getpriority(which, who)
result int prio; int which, who;

PRIO_PROCESS  0
PRIO_PGRP     1
PRIO_USER     2
```

To alter priority of a process, process group, or a particular user's processes use the **setpriority** call. Only the superuser can lower priorities.

```
setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range —20 to 20. The default priority is 0, and lower values cause more favorable execution. The **getpriority** call returns the highest priority (lowest numerical value) of any of the specified processes.

Descriptors

This information to be supplied later.

System Facilities

This topic deals with the system facilities that are not considered part of the kernel. The system abstractions described here include:

- directory contexts
- files
- communications domains
- sockets

Directory Contexts and Files

Certain operations are common to all system abstractions. These include the **read**, **write**, and **ioctl** calls. Also included in these operations are mechanisms where normally synchronous operations can happen in a non–blocking or asynchronous fashion. In non–blocking operations, a process can have no more than one input/output request outstanding.

Read and Write

The **read** and **write** system calls can be applied to communications channels, files, terminals, and devices:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; char *buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; char *buf; int nbytes;
```

The **read** call transfers as much data as possible from the object *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*. This value is —1 if a return occurred before any data was transferred, due to an error or the use of non–blocking operations. The **read** returns 0 at an end–of–file.

The **write** call transfers data from the buffer to object *fd*. Depending on the type of *fd*, the **write** call accepts some portion of the number of bytes returned; you can resubmit the other bytes in a later request.

With the **readv** and **writev** calls, you can scatter data on input, or gather it for output, using an array of input/output vector descriptors.

```
cc = readv(fd, iov, iovcnt);
result int cc; fd; struct iovec *iov; int iovcnt;

cc = writev(fd, iov, iovcnt);
result int cc; fd; struct iovec *iov; int iovcnt;
```

The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t         iov_msg;        base of a component
        int             iov_len;   length of a component
};
```

Iovec specifies the base address and length of the memory area where data should be placed. **Readv** scatters the input data into *iovcnt* buffers specified by the members of the *iovec* array. This call is not supported for raw devices or for files on remote hosts. *Iovlen* is the count of elements in the *iov* array. The **writev** call performs the same action as **write**, except that it gathers the data for output from *iovcnt* buffers specified by the members of the *iovec* array.

Input/Output Control

The **ioctl** call performs control operations on an object (socket or file descriptor):

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

The specified *request* is performed on the object *fd*. The *request* parameter specifies whether the argument buffer is read, written, read and written, or unnecessary. It also specifies the size of the buffer and the request. Different descriptor types and subtypes can use different **ioctl** requests. For example, you could use different **ioctl** requests for operations on terminals that control the flushing of input and output queues versus operations on disks that control formatting. The names of basic control operations are defined in <sys/ioctl.h>.

Non-blocking and Asynchronous Operations

You can use the **fcntl** call to perform non-blocking operations on a descriptor:

```
#include <fcntl.h>

result = fnctl(fd, cmd, arg);
int result;
int fd, cmd, arg;
```

The **fcntl** call controls operations on open descriptors, where *fd* is an open descriptor. You can set a descriptor in non-blocking mode using the value F_SETFL for *cmd*. The value of *arg* depends on the value of *cmd*. See the *UTek Command Reference, fnctl(2)* for details. When you set a descriptor to non-blocking mode the operation on that descriptor either completes immediately or returns an EWOULDBLOCK error if there is no data to be **read**. The process can use the **select** call on the descriptor to determine when a **read** is possible.

When a descriptor can accept less output than is requested, either it accepts some of the data provided, or returns an error indicating that the operation would block. The system can perform more output as soon as a **select** call indicates that the object is writeable.

You can perform operations other than data input and output on a descriptor in non-blocking fashion. If they cannot return immediately, these operations return an error. You can then use the **select** call to find out when the descriptor is readable or writeable.

File System

Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories, as well as files and references to other objects, such as devices and inter–process communication sockets.

Each file is organized as a linear array of bytes. No record boundaries or system–related information is included in a file. You can read or write files in a random–access fashion. You can read the data in a directory as though it were an ordinary file, but only the system can write into directories. The file system stores only a small amount of ownership, protection, and usage information with a file.

Naming

The file system calls take *pathname* arguments. These consist of zero or more component *filenames* separated by slashes (*/*), where *filename* is up to 255 characters, excluding null and /.

Each process has three naming contexts: one for the root directory of the file system, one for the current directory, and one for the network. The system uses these in the filename translation process. If a pathname begins with a slash, it is called an *absolute* or *full* pathname, and is interpreted relative to the root directory. If the pathname does not begin with a slash, it is a *relative* pathname and is interpreted relative to the current directory.

The system limits the total length of a pathname to 1024 characters.

The filename .. in each directory refers to the parent directory of that directory. The parent directory of a file system is always the root directory.

The **chdir** and **chroot** calls change the current working directory and the root directory contest of a process. Only the superuser can change the root directory context of a process:

```
chdir(path);
char *path;

chroot(path);
char *path;
```

Creation and Removal

The file system lets you create and remove directories, files, and special devices from the file system.

Directory Creation and Removal

The **mkdir** call creates a directory:

```
mkdir(path, mode);
char *path; int mode;
```

The **rmdir** system call removes a directory: To delete a directory it must be empty.

```
rmdir(path);
char *path;
```

File Creation

Files are created using the **open** system call:

```
#include <sys/file.h>

fd = open(path, flags, mode)
int fd; char *path; int flags, mode;
```

This opens the file named *path* as specified by the *flags* argument. It returns a descriptor for that file in *fd*. If you specify O_CREAT as a flag (create a new file), the file is created with access mode *mode*, with values for *mode* as described in *UTek Command Reference chmod(2)*. The *path* parameter is a null-terminated pathname, while *flags* is constructed by **or**'ing the following values defined in <sys/file.h>:

O_READONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_NDELAY	Do not block on open.
O_APPEND	Append on each write.
O_CREAT	Create file if it does not exist.
O_TRUNC	Truncate size to 0.
O_EXCL	Error if O_CREAT is set and file exists.

When the **open** completes the value *fd* is returned, and the file pointer used to mark the current position within the file is set to the beginning of the file. Before a file can be opened, the user must have access to the file.

You should specify one only of O_READONLY, O_WRONLY, or O_RDWR. If the **open** specifies to create the file with O_EXCL and the file already exists, the **open** fails without affecting the existing file.

Creating References to Devices

The file system allows entries that reference peripheral devices. Peripherals are distinguished as *block* or *character* devices, according to their ability to support block-oriented operations. Devices are identified by their *major* and *minor* device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind.

Structured devices perform all operations internally in block quantities, while unstructured devices have a number of special **ioctl** operations and can perform input and output in large units.

Removing Files and Devices

You can remove a reference to a file or special device using the **unlink** call:

```
unlink(path);
char *path;
```

The caller must have write access to the directory where the file is located.

Do not unlink a device. This can cause severe problems and possible loss of data.

Reading and Modifying File Attributes

You can obtain detailed information about the attributes of a file using the **stat** and **fstat** calls:

```
#include <sys/types.h>
#include <sys/stat.h>

stat(path, buf);
char *path; result struct stat *buf;

fstat(fd, buf);
int fd; result struct stat *buf;
```

The **stat** call is on a pathname, while **fstat** is on an open file descriptor. The structure *stat* includes the file type, protection, ownership, access times, size, and a count of hard links.

If the file is a symbolic link, you can use the **lstat** call to find the status of the link itself (rather than the file referenced by the link).

```
lstat(path, buf);
char *path; result struct stat *buf;
```

When a new file is created, it is assigned the user ID of the process that created it and the group ID of the directory where it was created. You can change the ownership of a file using the **chown** and **fchown** calls:

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

The first call changes the ownership of the file referenced by *path*, while the second changes the ownership of the file referenced by the descriptor *fd*.

In addition to ownership, each file has three levels of access protection associated with it. These levels include access for the owner, the group, and global (all users and groups). Each level of access has separate indicates for read permission, write permission, and execute permission. You can set the protection associate with a file with the **chmod** and **fchmod** calls:

```
chmod(path, mode);
char *path; int mode;
```

```
fchmod(fd, mode);
int fd, mode;
```

As with **chown**, **chmod** references a file by its pathname and **fchmod** references it by its descriptor. The value *mode* represents the new protection for the file. The file mode is a three-digit octal number. Each digit encodes read access as 4, write access as 2, and execute access as 1, **or**'ed together. The 0700 bits describe owner access, 070 group access, and 07 access rights for other processes.
You can set the access and modify times for a file using the **utimes** call:

```
utimes(path, tvp)
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, so that you can preserve relationships between the times the file was modified.

Links and Renaming

Links provide multiple names for the same file. The link exists separately from the file it references.

Two types of links exist, *hard links* and *symbolic links*. A hard link is a reference counting mechanism that lets a file have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process. For more information on symbolic and hard links see Section 2E, *The UTek Fast File System*.

Hard links ensure that the target file is always accessible, even after its original directory entry is removed. No such guarantee exists for a symbolic link, but it can span file system boundaries. The following calls create a hard link, or a symbolic link, respectively:

```
link(path1, path2);
char *path1, *path2;

symlink (path1,path2);
char *path1, *path2;
```

These calls create a new link, named *path2* to *path1*.

The **unlink** call removes either type of link. Only the superuser can unlink a directory:

```
unlink(path)
char *path;
```

If a file is a symbolic link, the value of the link can be read with the **readlink** call:

```
cc = readlink(path, buf, bufsize);
result int cc; result char *path, *buf; int bufsize;
```

This call places the contents of the symbolic link *path* in buffer *buf* with buffer size *bufsize*.

You can rename the links using the **rename** call:

```
rename(from, to);
char *from, *to;
```

This causes the link *from* to be renamed to *to*. Both *from* and *to* must be in the same file system and of the same type (file or directory). If *to* exists, it is first removed.

Extension and Truncation

Files are created with zero length and extended by writing or appending to them. While a file is open the system maintains a pointer into the file, indicating the current location in the file associated with the descriptor. You can move this pointer about in random–access fashion. To set the current offset into a file, use the **lseek** system call:

```
#include <sys/file.h>
#include <sys/types.h>

pos = lseek(fd, offset, whence);
result int pos; int fd; off_t offset; int whence;
```

The **lseek** calls sets the file pointer of the file referenced by *fd* and returns the value in *pos*. The value of *whence* is one of the following:

L_SET	0	set absolute file offset
L_INCR	1	set file offset relative to current position
L_XTND	2	set offset relative to end–of–file

Files can have "holes" in them. Holes are empty areas in the linear extent of the file where data has never been written that take up no disk space. You can create them by seeking to a location past the current end–of–file and writing there. The system treats the holes as zero–valued bytes.

You can truncate a file with the calls **truncate** and **ftruncate**:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd; int length;
```

The **truncate** call references a file by its pathname, while **ftruncate** references it by its descriptor. Both calls reduce the size of the files to **length** bytes.

Checking Accessibility

A process running with different real and effective user IDs can check the accessibility of the file to the real user using the **access** call:

```
access(path, mode);
char *path, int mode;
```

The value of *mode* is constructed by **or**'ing the following bits defined in <sys/file.h>:

F_OK	0	file exists
X_OK	1	file is executable
W_OK	2	file is writeable
R_OK	4	file is readable

The presence or absence of advisory locks does not affect the result of **access**.

Locking

The file system lets processes synchronize their access to shared files. An *advisory lock* is applied to a file only when a program requests it, so it is effective only when all the programs accessing a file use the same locking scheme. A process can set an advisory **read** or **write** lock on a file, to presever its exclusive access to the file. See Section 2E, the *Utek Fast File System* for more information on file locking.

Locking is performed after an **open** call, using **flock**:

```
flock(fd, operation);
int fd, operation;
```

The operation parameter is formed from bits defined in <sys/file.h>:

LOCK_SH	1	shared lock
LOCK_EX	2	exclusive lock
LOCK_NB	4	don't block when locking
LOCK_UN	8	unlock

You can use successive lock calls to increase or decrease the level of locking. If an object is currently locked by another process when the **flock** call is made, the caller is blocked until the current owner releases the lock. You can avoid this by including LOCK_NB in the *operation* parameter. Advisory locks held by a process are automatically deleted when the process terminates.

Interprocess Communications

Communication Domains

The system provides access to an extendible set of communication domains. A communication domain is identified by a constant defined in <sys/socket.h>. The most important domains supported by the system are the UNIX domain for communication within the system, and the INET domain for internetwork communication.

Socket Types and Protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type that describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplicate messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

SOCK_DGRAM	datagram
SOCK_STREAM	virtual circuit
SOCK_RAW	raw socket
SOCK_RDM	reliably delivered message
SOCK_SEQPACKET	sequenced packets

The SOCK_DGRAM type models the semantics of in network communication; messages can be lost or duplicated, or arrive out of order. The SOCK_RDM type models the semantics of reliable ; messages arrive unduplicated and in order, and the sender is notified if messages are lost. The **send** and **receive** operations generate reliable or unreliable The SOCK_STREAM type models connection-based virtual circuits; two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or sent out of order. You can use the last two abstractions for out-of-band transmission to send out-of-band data.

The SOCK_RAW type is used for unprocessed access to internal network layers and interfaces. It has no set semantics.

Each socket must have a concrete *protocol* associated with it. The protocol is used within the domain to provide the semantics required by the socket type. For example, within the INTERNET domain, the SOCK_DGRAM type can be implemented by the UDP user datagram protocol, and the SOCK_STREAM type can be implemented by the TCP transmission control protocol.

Socket Creation and Naming

Sockets can be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the **socket** call:

```
#include <sys/types.h>
#include <sys/socket.h>

s = socket(af, type, protocol);
result int s; int af, type, protocol;
```

An unconnected socket descriptor can yield a connected socket descriptor by actively connecting to another socket, or by associating itself with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. This is done with the **bind** system call:

```
bind(s, name, namelen);
ints; struct sockaddr *name; int namelen;
```

You can retrieve a socket's bound name with **getsockname**:

```
getsockname(s, name, namelen);
int s; struct sockaddr *name; result int *namelen;
```

You can retrieve the peer's name with the **getpeername** call:

```
getpeername(s, name, namelen);
int s; result struct sockaddr *name; result int *namelen:
```

Accepting Connections

Once a binding is made, it is possible to **listen** for connections:

```
listen (s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

The **accept** call returns a descriptor for a new, connected socket from the queue of pending connections on *s*.

Making Connections

The **connect** call makes an active connection to a named socket:

```
connect(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

You can also create connected pairs of sockets without using the domain's space to rendezvous. Do this using the **socketpair** call:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with **accept** and **connect**.

The **pipe** call creates a pair of SOCK_STREAM sockets in the UNIX domain, with fd[0] only writeable and fd[1] only readable:

```
pipe(fd);
result int fd[2];
```

Sending and Receiving Data

You can use **sendto** to send messages from a socket if it is not connected:

```
cc = sendto(s, msg, len, flags, to, tolen);
result int cc; int s; char *msg; int len, flags;
struct sockaddr *to; int
```

You can use **send** if the socket is connected:

```
cc = send(s, msg, len, flags);
result int cc; int s; char *msg; int len, flags;
```

The corresponding receive primitives are **recvfrom** and **recv**:

```
cc = recvfrom(s, buf, len, flags, from, fromlen)
result int cc; ints; result char *buf; int len, flags;
struct sockaddr *from; result int *fromlen;

cc = recv(s, buf, len, flags);
result int cc; int s; char *buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlen* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high priority out–of–band messages:

MSG_OOB	0x1	process out–of–band data
MSG_PEEK	0x2	peek at incoming message

Scatter/Gather and Exchanging Access Rights

It is possible to scatter and gather data, and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call is large. So the system defines a message header structure that contains the parameters to the calls:

```
struct msghdr {
        caddr_t         msg_name;       optional address
        int             msg_namelen;    size of address
        struct          iovec *msg_iov; scatter/gather array
        int             msg_iovlen      # elements in msg_iov
        caddr_t         msg_accrights   access rights sent/received
        int             msg_accrightslen size of msg_accrights
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* can be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the UNIX domain, these are an array of integer descriptors, taken from the sending process and duplicated in the received.

This structure is used in the operations **sendmsg** and **recvmsg**:

```
cc = sendmsg(s, msg, flags);
result int cc; int s; result struct msghdr msg[]; int flags;

cc = recvmsg(s, msg, flags);
result int cc; int s; result struct msghdr msg[]; int flags;
```

Read and Write with Sockets

You can use the normal **read** and **write** calls with connected sockets, and translate them into **send** or **receive** calls. A process can operate on a virtual circuit socket, a terminal, or a file with blocking input/output operations, without distinguishing the descriptor type.

Shutting down Halves of a Full Duplex Connection

A process that has a full–duplex socket such as a virtual circuit, and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, how);
int s, how;
```

The parameter *how* is 0 to not read further, 1 to not write further, and 2 to completely shut the connection down.

Socket and Protocol Options

Sockets, and their underlying communication protocols, can support *options*. These options can be used to manipulate implementation–specific or non–standard facilities. The **getsockopt** and **setsockopt** calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname; result char *optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname; char * optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for a socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified level. The level SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol to act on the option request; these values are normally interpreted as a protocol number.

UNIX Communications Domain

Types of Sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe–like facilities, while SOCK_DGRAM provides reliable message–style communications.

Access Rights Transmission

In the UNIX domain, you can use **sendmsg** to pass file descriptors within the system. This means that user processes can be used to build system facilities.

Internet Communications Domain

Socket Types and Protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM is supported by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue.

Socket Naming

Sockets in the INTERNET domain have names composed of the 32–bit internet address, and a 16–bit port number. You can use options to provides source routing for the address, security options, or additional address for subnets of INTERNET where 32–bit addresses are insufficient.

Raw Access

The INTERNET domain allows the superuser access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

The UTek Fast File System

Introduction

This section discusses the UTek Fast File System that is derived from the Fast File System developed for 4.2bsd UNIX. To introduce the Fast File System this introduction compares major features of the original UNIX file system to the UTek Fast File System.

Originally, UNIX file systems transferred data in blocks of 512 bytes. But this rate of data transfer is too slow to offer the kind of performance that you need from an engineering workstation.

In the original UNIX system the *super block* defined the parameters of the file system. These parameters include:

- number of data blocks in the file system
- count of the maximum number of files
- a pointer to the list of free blocks

Every file has a descriptor associated with it called an *inode*. The inode describes ownership of the file, when it was last modified, and the location of data blocks that comprise the file. The inode information is stored separately from the data blocks. So to access a file the disk must perform a long seek between the inode and the file itself. Because the original file system transfers only 512 bytes at a time, the data blocks for the same file are frequently far apart. So inefficient layout of data blocks, the small block size, and the many seeks performed by the disk limit the speed of the original UNIX file system.

The UTek File System Organization

Like the original UNIX file system, the Berkeley file system, and thus the UTek file system, has a superblock that describes the file system. When you create the file system, duplicate copies of the superblock are created.

The minimum size of the file system blocks is 4096 bytes, although the size of a block can be any power of two greater than or equal to 4096 bytes. The superblock maintains the block size of the file system, so you can have file systems of different block sizes accessible on the same system. You must decide on the block size when you create the file system; you must rebuild the file system to change the block size.

As a three-dimensional object, the disk is divided in several different ways. The *tracks* of a disk are like the tracks of a record album, except that they have a third dimension introduced by the multiple surfaces of the disk. A *cylinder* of a file system consists of tracks of the same number on all the surfaces of the disk. So that if you look down from the top of the disk you see concentric cylinders. The UTek file system partitions the disk into *cylinder groups*. The cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group has associated accounting information including a redundant copy of the superblock, space for inodes, a bit-map describing available blocks in the cylinder group, and a summary of data block usage within the cylinder group. When the file system is created, one inode is allocated for each 2048 bytes of disk space.

If you placed the accounting information at the beginning of each cylinder group, it would always be on the top surface of the disk. So if a single hardware failure destroyed that portion of the disk, all copies of the superblock would be destroyed. To avoid this the cylinder group accounting information begins at a floating offset from the beginning of the cylinder group. The offset for each cylinder group is about one track further than the beginning of the group. So copies of the superblock spiral down into the disk, and you can lose any single track, cylinder, or surface without losing all the copies of the superblock.

Organization of Data Blocks

In the UTek file system, data is arranged so that larger blocks can be transferred in a single disk transfer. A block in the new file system is at least 4096 bytes, and by increasing the block size the disk can transfer more information in each transaction. If you have files larger than 4096 bytes, several blocks of 4096 bytes can are allocated from the same cylinder so that even larger data transfers are possible.

One potential problem with uniformly large blocks is that they waste disk space. To avoid this the UTek file system divides a single block into one or more *fragments,* so that you can store files smaller than 4096 bytes without wasting space. The fragment size of a file system is specified when you create the file system; each block can be broken into two, four, or eight addressable fragments. The smallest fragment size is 512 bytes, the disk sector size of your workstation. To keep track of space on the level of fragments, a block map is associated with each cylinder group. The availability of an entire block is determined by the availability for all its fragments. Figure 2E-1 shows a block map for a system with 4096 byte blocks and 1024 byte fragments:
1024 byte fragments:

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0–3	4–7	8–11	12–15
Block numbers	0	1	2	3

Figure 2E-1. Layout of Blocks and Fragments in 4096/1024 File System.

Each bit in the map records the status of a fragment; X is a fragment in use and O is an available fragment. In this example fragments 0–5, 10, and 11 are in use, while fragments 6–9 and 12–15 are free. You cannot use fragments of adjoining blocks as a block, even if they are large enough. In this example, fragments 6–9 cannot be put together into a block; only fragments 12–15 are available for allocation as a block.

So when the file system allocates space for a file, it uses a combination of blocks and fragments. The principle that governs the allocation is to use the smallest possible numbers of blocks and fragments that accommodate the file. For example, to allocate an 11000 byte file on a 4096/1024 file system, two blocks and a 3072 byte fragment are used. If no 3072 byte fragments are available, a block is split into a 3072 byte fragment that is allocated to the file, and an unused 1024 byte fragment.

The *write* system call is the cornerstone of allocating space for a file. Each time data is written to a file, the system checks to see if the size of the file has increased. If the file needs to hold more new data one of three conditions apply:

1. The file is completely new, so no space has been allocated for it. A combination of full blocks and/or fragments that can accommodate the amount of new data is allocated.

2. There is enough space left in an already allocated block to hold the new data, and it is written into the available space in the block.
3. Only a fragment, instead of a block, has already been allocated for the file. If the new data and the data already in the fragment exceed 4096 bytes, a new block is allocated. The data in the fragment is copied to the beginning of the block, and the rest of the block is filled with the new data. Any remaining data is allocated to full blocks and/or fragments as described earlier.

One potential problem with laying data out in fragments is potentially recopying a one-fragment file up to three times as it grows. This frequent reallocation can be avoided if the user program writes a full block at a time. Because file systems with different block sizes can exist on the same system, the file system interface determines the optimal size for a read or write. For files, the optimal size for a read is the block size of the file system where it exists. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. The Standard Input/Output Library, the package used by most user programs, and certain system utilities such as archivers and loaders, determines the optimal size for reads and writes.

This scheme of laying data out in blocks and fragments does not require any more space in the file system than the original 512 byte blocks of the UNIX file system. The new file system uses less space for the data itself because it requires less indexing information for large files. This savings is offset by the space required to keep track of available blocks. The net result is use of the same amount of disk space when the new file system fragment size is 512 bytes.

The effectiveness of laying data out in blocks and fragments depends on having at least 10% free blocks on the disk. If the number of free blocks falls below this level only the system administrator can continue to allocate blocks. You can also change the minimum amount of free space, even when the file system is mounted and active. But if you set the number of reserved blocks to 0, the throughput of the file system is cut in half because the file system cannot put all the blocks for one file in one location. Files created during periods when the disk has little free space have a slow access speed, but you can restore a normal access speed by recreating the files when enough space is available.

File System Parameters

The UTek Fast File System uses the physical characteristics of your workstation to optimize the performance of the file system. These parameters include:

- processor speed
- disk characteristics
- hardware support for disk transfers

The file system tries to allocate new blocks on the same cylinder as other blocks in the same file. To speed access time the file system allocates consecutive blocks based on the rotational speed of the disk. So writing two "consecutive" blocks can mean skipping physically consecutive blocks so that the next block is coming into position under the disk head at the right time. The allocation routines in the file system calculate the number of blocks to skip so that the next block is ready for reading or writing.

Another factor that affects allocation is how fast the processor channel can transfer information and how much information can be read or written at one time. The amount of time it takes to skip to the next rotationally optimal block includes the time it takes for a disk transfer operation.

Once the file system determines how to find a rotationally optimal block, it must be free. The cylinder group summary information includes the availability of blocks at eight different rotational positions.

Layout of Inodes and Data Blocks

File system allocation routines are divided into two distinct groups. *Global routines* decide the placement of new directories and files. They also calculate rotationally optimal block layouts and decide when to move information to a new cylinder group because there are not enough blocks left in the current cylinder group to lay out the data blocks efficiently.

Local routines are called by the global routines. So once a global routine has decided where information should be placed, the local routine allocates the requested block if it is free. If the requested block is not free, local routines calculate the block rotationally closest to the one requested and allocate it.

Inodes are used to describe both files and directories. Because files in the same directory are frequently accessed together, the global routines try to place all the files in a directory in the same cylinder group. But directories are placed in a cylinder group that has a greater than average number of free inodes and the fewest number of directories in it already. This leaves space for all the new file that will be created under a directory so that files can be clustered together. Within a cylinder group the inodes themselves are allocated randomly, on a next free basis. This means that you can access all the inodes for a cylinder group with fewer disk transfers.

The global routines also try to place all the data blocks for a file in the same cylinder group, if possible rotationally optimal on the same cylinder. The only problem with this layout of data blocks is that large files quickly use up available space in the cylinder group and spill over into other areas. Large files also means that enlargements for other files in the same cylinder group spill over to another cylinder group. To correct this problem a file that exceeds 32 kilobytes is redirected to a newly chosen cylinder group that has a higher than average number of free blocks.

The global routines determine what block is preferable, and then call local routines to allocate the requested block. If the block is not free the local routine uses this strategy for allocation:

1. Use the available block rotationally closest to the requested block on the same cylinder.
2. Use a block within the same cylinder group.
3. Quadratically rehash among the cylinder groups looking for a free block.
4. If a rehash fails, apply an exhaustive search.

The last two steps of this process typically occur on a file system that has less than 10% free space, so rehashing is used to make them run quickly.

Other File System Enhancements

Other than speed, the following functional enhancements were added to the UTek file system.

Long File Names

Filenames can be of nearly arbitrary length, up to 255 characters. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems a set of directory access routines has been introduced to provide a uniform interface to directories.

Directories are allocated in units of 512 bytes. Each allocation unit contains variable-length directory entries, with each entry contained within an allocation unit. The first three fields of a directory entry contain an inode number, the length of the entry, and the length of the name contained in the entry. If the inode number is set to 0, that entry is unallocated. Following this information is the null-terminated name, padded to a 4-byte boundary.

Free space in a directory is held by directory entries whose record length exceeds the space required by the directory itself. All the bytes in an allocation unit are claimed by directory entries. Normally this results in a large last entry. When you delete an entry from a directory, the freed space increases the length of the previous entry.

File Locking

The original UNIX file system had no provision for locking files.. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. The UTek file system provides both *hard locks* and *advisory locks*. Hard locks are enforced whenever a program tries to access a file; an advisory lock is applied to a file only when a program requests it. So advisory locks are effective only when all programs accessing a file use the same locking scheme. Typically, advisory locks are used to lock files being run by the system administrator because programs with system administrator privilege can override any protection scheme.

Advisory locks can be *shared* or *exclusive*. Only one process can have an exclusive lock on a file, while you can have several shared locks on a file. If you request a lock when another process has an exclusive lock on a file, or if you request an exclusive lock when another process has any lock, the attempt to open the file blocks until the file is free. You can override this block by specifying that the locking request return with an error if it cannot obtain the lock. Because shared and exclusive locks are only advisory, another process can override the lock by opening the same file without a lock.

You can apply or remove locks on open files, so you can manipulate the locks without needing to close and reopen the file. This is useful, for example, when a process wants to open a file with a shared lock to read some information that determines whether an update is required. It can then get an exclusive lock to read, modify, and write to the file.

A process can deadlock itself by requesting locks on two separate file descriptors for the same object. The file system only prevents a second lock of the same type on a file descriptor.

For more specific information on how to implement file locking see *UTek Command Reference, flock (1)* and *flock(2)*.

Symbolic Links

The original UNIX file system allows *hard links*, or multiple directory entries in the same file system to reference a single file. Files do not live in directories, but exist separately and are referenced by links. This does not allow references to a single file across physical file systems or between machines.

The UTek file system uses a *symbolic link,* or a file that contains a pathname. When the system interprets a pathname that has a symbolic link as one of its components, the contents of the symbolic link are prepended to the rest of the pathname. If the symbolic link contains an absolute pathname it is used; otherwise, the symbolic link is relative to its position in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname they are using. However, some system utilities can detect and manipulate symbolic links. See *UTek Command Reference, ln(1)* and *symlink(2)*.

Renaming Files

The original UNIX file system required three calls to the system to rename a file. The UTek file system implements the **rename** system call that performs the rename in one operation and guarantee the existence of the original name.

In addition, **rename** lets you move directories around in the directory tree hierarchy. It also checks to see that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. This kind of corruption occurs if a parent directory is moved into one of its descendants.

The Distributed File System

The distributed file system lets you use any of the UTek commands with files that are on other machines on the network. To access a file on a remote machine you enter a UTek command as you normally would, but you specify a file on another machine with the following syntax:

//machine/pathname

where *machine* is the name of the workstation the file is on and *pathname* is the *full* pathname of the file.

The following example copies the file */usr/joe/datafile* from the workstation named *engr1* to a file named *temp* in your current working directory on the current workstation:

cp //engr1/usr/joe/datafile temp

The following example places a user in the directory */usr/joe* on the workstation *engr1*.

cd //engr1/usr/joe

The following example uses the **vi** editor to edit a file named */usr/joe/datafile* that is on the workstation *engr1*.

vi //engr1/usr/joe/datafile

When you execute a command that resides on another workstation, the command is copied to your workstation and executed on your workstation. For example, if you typed:

//engr1/bin/who

the **who** command is copied from *engr1* to your workstation and is executed on your workstation.

DFS Protection

The system administrators of each workstation on the network decide who can access their workstations with the distributed file system. In order to access another workstation on the network with the distributed file system, you must have:

- Distributed file system access to the other workstation. This access is permitted by entries in the */etc/hosts.dfs.access* file on the other workstation.
- Access to the file you are trying to read or write. This access is permitted by the settings of the protection bits of the file. This type of access control is identical to the access control used for files on your workstation.

When you enter a command that accesses a file on a remote machine, the remote machine checks to see if your workstation is listed in the */etc/hosts.dfs.access* file. If your machine is listed in this file then you and all other users on your workstation can access the remote machine, if you have accounts on the remote machine.

For more on the */etc/hosts.dfs.access* file, see your *System Administration* manual and *hosts.dfs.access(5)* in the *UTek Command Reference* manual.

MDQS — The Multidevice Queueing System

The UTek Multi–Device Queueing System (MDQS) provides a flexible means of sending printing or batch requests to devices that you define. MDQS arranges the order of tasks and sends them to a variety of logical devices. The most common use for MDQS is to send printing jobs to a printer. The **batch** queue orders files that contain UTek commands, and sends them to the shell for processing.

The queue in MDQS is a separate entity from the device; the queue sets up the priority of tasks, then the tasks are mapped to a particular device. So you can hav both multiple queues and multiple devices, with more than one queue mapped to a device, or more than one device mapped to a queue. Unlike a traditional printer spooler, MDQS accommodates different job priorities for multiple devices, and lets you change job priority very easily.

Most of your contact with MDQS configuration is through the **sysadmin** program, also called the system administration interface. This program displays a series of menus that let you configure the MDQS system and do other system administratio tasks. See your *System Administration* manual to find out how to use the **sysadmi** program to configure MDQS. This section presents the concepts of the MDQS system as a whole, and gives you more details about how MDQS works. Use the **sysadmin** command to do the routine configurations of MDQS, and refer to this section if you have a configuration that cannot be accomplished using **sysadmin**, to find out more about the internal workings of MDQS.

Creating a Queue Entry

To create a queue entry, you submit a request to the queue. You can submit a request to the printer queue using the **lpr** command. Or you can submit batch requests to run many commands at once using the **batch** command. See the *UTe Command Reference* for details on these commands.

Mapping a Queue to a Device

The procedure that takes a job in the queue and sends that job to a particular device is the key to the flexibility of MDQS. Figure 2G–1 shows a mapping of multiple queues and devices. The first queue sends jobs to the first device, the second queue sends jobs to either device, and the third queue sends jobs to the second device.

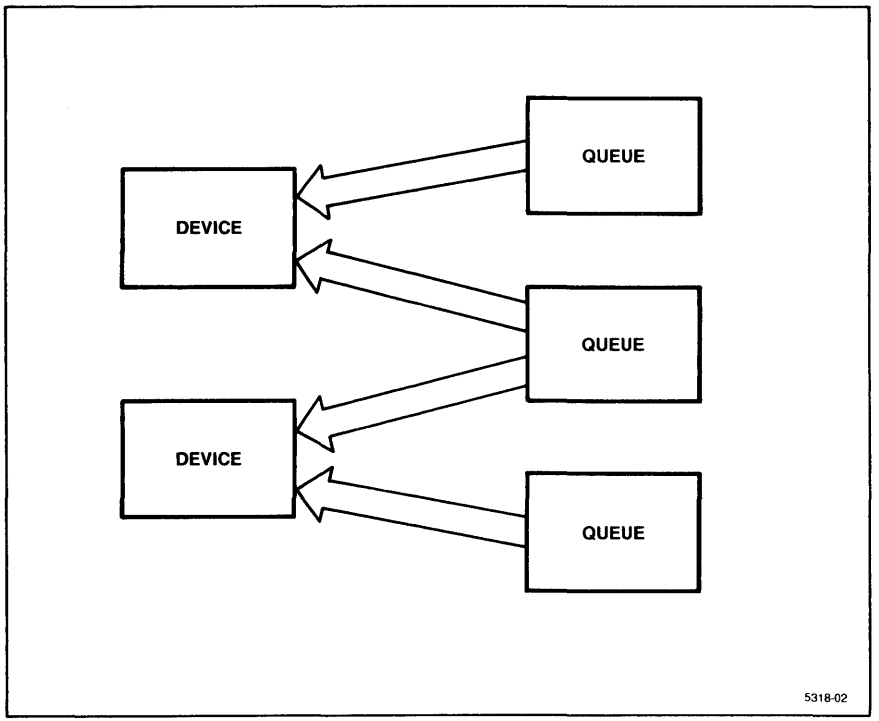


Figure 2G–1. Multiple Queues and Devices.

To get a job into a queue, MDQS builds a *request*. The request contains all the information necessary to process a job — what files to print, or what commands to run as a **batch** process, and what queue is appropriate for the job. After MDQS builds the request, it sends the request to the appropriate queue. From the queue the request is mapped to a particular device, and as we said earlier more than one queue can be mapped to a device. The *server process* is the actual program that executes a request. So the information that maps a queue to a device includes the name of the queue, the name of the device, and the name of a server process for each mapping of a queue to a device.

When all the information for a mapping of queue to device (queue-name, device-name, and server process) is in place, the MDQS *scheduler daemon* determines what server processes to execute first. A daemon is a process that runs all the time and automatically takes care of system procedures like networking, printing, and mail operations. Figure 2G-2 shows how the scheduler daemon examines the devices until it finds an empty device. An empty device does not have a request attached to it.

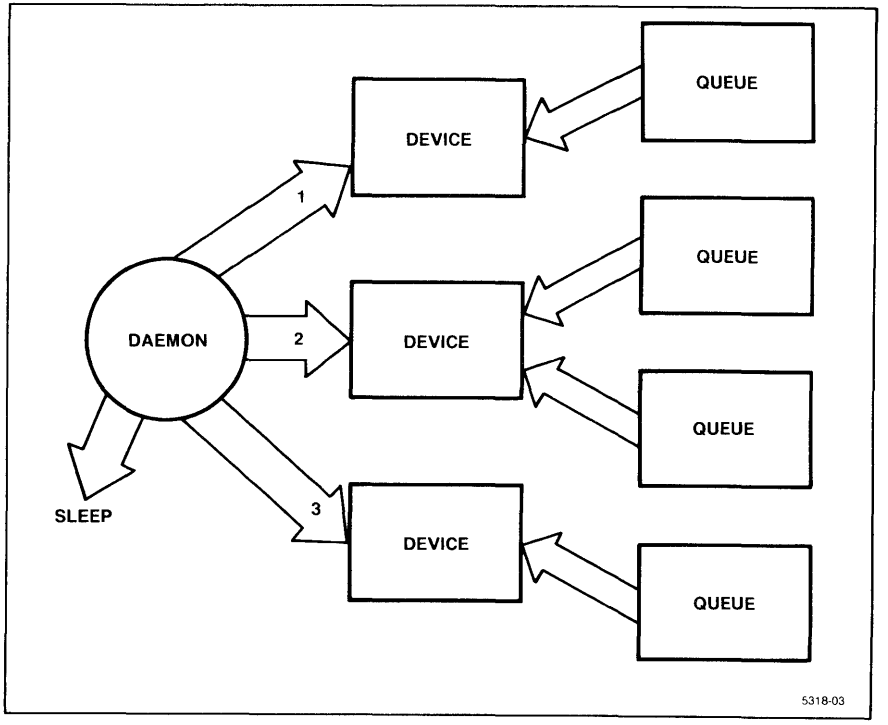


Figure 2G-2. Daemon Scans for Empty Devices.

When the daemon finds an empty device, it takes the first request in the queue mapped to that device, and tells the server process to execute that request. If there is more than one queue mapped to a device, by default the first queue is emptied before requests in the second queue can be processed. However, you can change the order in which queues are emptied. See the section *The Qconf File* to change emptying the first queue by default.

After the daemon finds the first eligible request, it moves on to the second empty device, finds the queue mapped to that device, and pulls a request from the second queue to attach it to a device. When the queue is attached to the device, the appropriate server process executes the request. Figure 2G–3 shows how the daemon moves from one device to another. The order of the devices is set up in the *qconf* file.

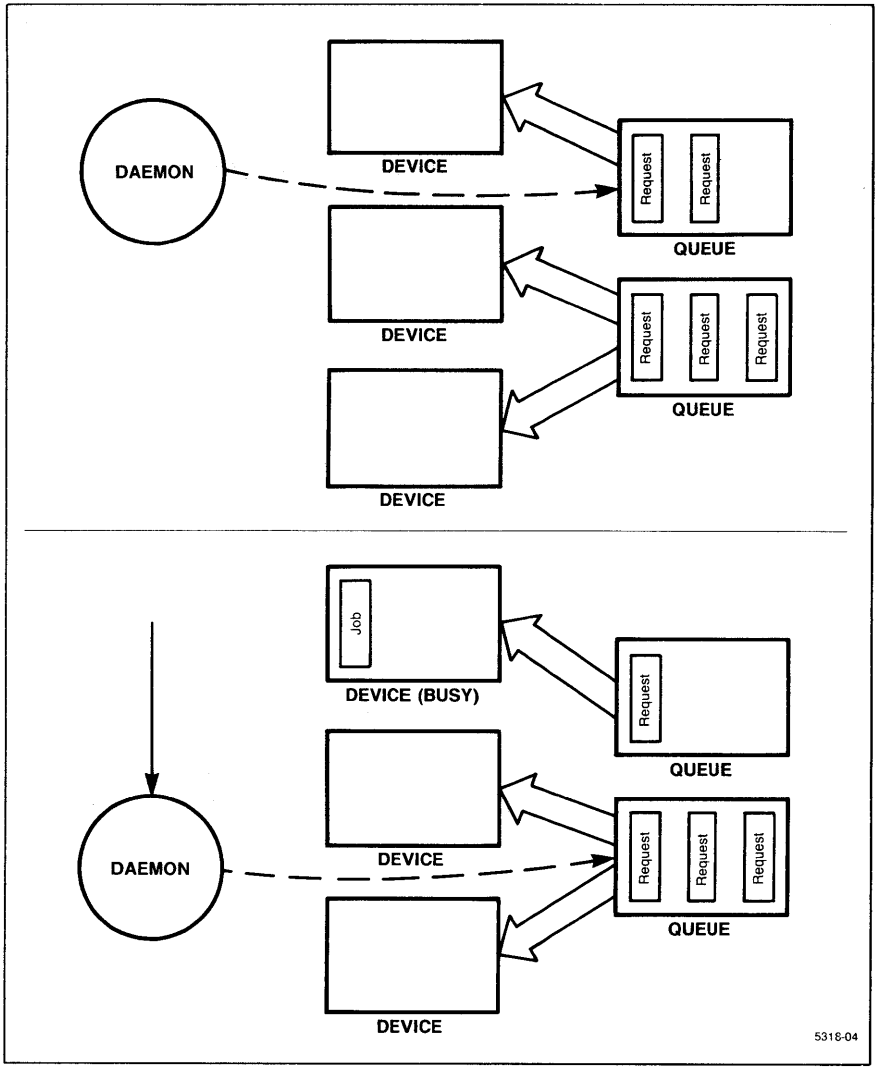


Figure 2G–3. Daemon Moves from First to Second Device.

The scheduler daemon is a program called **mdqsd** that resides in the directory */etc*. Normally the daemon is started by the **init** program when you boot the system, or by the system administrator interface. But you can also start the daemon by executing **mdqsd** if you are logged in as *root*.

When no printing requests exist, the daemon stops checking for empty devices and remains idle. But a parameter that you set in the *qconf* file, called **scanwait**, activates the daemon at regular intervals to check whether delayed requests should be transferred to their appropriate queues. See the later discussion *The Qconf File* for information on setting the **scanwait** parameter and the following discussion on *Delayed Queues*.

Delayed Queues

You can use the **—a** option of **lpr** or **batch** to send a request to a *delayed queue*. A delayed queue simply waits a certain amount of time before it puts the request in the regular queue. You specify a time, after which the request can be moved from the delayed queue to the regular queue. For example:

lpr –a 10:00pm *request*

This command sends *request* to the delayed queue, where it stays until after 10:00 pm. The request is moved to the regular lineprinter queue the first time after 10:00 pm that it is examined by the scheduler daemon. The scheduler daemon examines delayed queues every *x* seconds, where *x* is determined by the **scanwait** parameter in the *qconf* file. See the discussion *The Qconf File* for more information on the **scanwait** parameter. See your *UTek Command Reference, getdate(5),* for information on date formats accepted by the **--a** option.

The MDQS Directory Structure

The directory structure of MDQS is designed to provide maximum security, while reducing linear searches of directories by the daemon. A request to the MDQS system (for example the **lpr** command) starts out as a nonprivileged process, but as it moves deeper into the hierarchy of the MDQS directories, it takes on the status of a privileged process. First a request moves into a temporary directory. Then, to protect the process, it moves into the *lock* directory, which can only be accessed by a privileged process.

Following is a diagram of the directory hierarchy:

```
/usr/spool/q/
        qtmp/
        lock/
                        home/
                                cntrl/
                                data/
                                new/
                                mod/
                                adm/
                                hold/
```

Many of the MDQS processes run in the *cntrl* directory, for example the **qmod** and **qdev** programs. These programs are the processes that modify or delete a request, and modify or display the status of a local device. The *cntrl* directory also contains one control file for every printing request.

The *data* directory is closely related to the *cntrl* directory. It provides the raw data (the file you entered) to the *cntrl* directory, so that a control file can be produced for each request.

The *new* directory is normally empty, but when it contains something it notifies the daemon that a new request is ready to be processed. The daemon then enters the request in an internal queue and unlinks the file from the *new* directory. This file containing the request is actually a link to the same file in the *cntrl* directory, but by putting the link to the request in the *new* directory, it is easier for the daemon to recognize what requests are new.

The *mod* directory works in a fashion similar to the *new* directory, except that it indicates whether the user has modified or deleted a request using the **qmod** program. Instead of maintaining a link to the *cntrl* directory, files in the *mod* directory contain all the old and new request information so that the daemon can easily find the original request in its internal queues.

The *adm* directory contains queue status files and the lock files for the daemon, as well as the device status files for the server process. The **qdev** program uses the device status files to modify the status of a local device. For example, you could use the **qdev** program to change printing forms from wide to narrow for a printing device. The daemon also uses the *adm* directory to communicate information about the device that is currently active, or about a device failure, back to the **qdev** program.

The daemon uses the *hold* directory to save the copies of request files that cause severe errors in the daemon or the server processes.

The Qconf File

The *qconf* file sets up the queues, sets up devices, and maps queues to devices. It also sets many changeable parameters, such as forms and priority.

The daemon checks the run-time configuration file, */etc/qconf*, for any modifications. When you have modified the *qconf* file, restart the daemon using the **daemon** command. For more information on this command see your *UTek Command Reference, daemon(8)*. You can also change the *qconf* file using the **sysadmin** program. As portions of the *qconf* file are discussed, this section also gives you the name of the corresponding menu in the **sysadmin** program.

Example 2G-1 shows the *qconf* file for a computer called **hammer**. Other computers on the same local area network are called **shark, tekecs, orca,** and **mako**.

```
#       MDQS configuration file (for hammer)
#
#       Parameters
#
console             /usr/spool/q/qtmp/mdqs.log
openwait 10
scanwait 60
maxfailures         10
sysmgr              steveh@hammer
#
print-queue         lp
print-forms         narrow
print-prior         64
print-hdr /usr/lib/mdqs/lphdr
#
batch-queue         batch
batch-forms         Shell
batch-prior         64
-------------------------------
# Device Descriptions
#
# <dname> <device>          <status>
lp0       /dev/lp0
lp1       /dev/lp1
batch0    /dev/null  anyform
net       /dev/null  anyform,skipmsg
-------------------------------
# Queue Descriptions
#
# <qname>
lp
Lp
sharklp
tekecslp
orcalp
makolp
batch
```

Example 2G-1. Example Qconf File.

```
# Queue—>Device Mappings
#
# <qname>          <dname> <server>
batch              batch0  /usr/lib/mdqs/shserver
lp                 lp0     /usr/lib/mdqs/plpserver
lp                 lp1     /usr/lib/mdqs/plpserver
Lp                 net     /usr/lib/mdqs/netsend shark Lp
sharklp            net     /usr/lib/mdqs/netsend shark lp
tekecslp  net      /usr/lib/mdqs/netsend tekecs lp
orcalp             net     /usr/lib/mdqs/netsend orca lp
makolp             net     /usr/lib/mdqs/netsend mako lp
```

Example 2G–1 (cont.). Example Qconf File.

As you can see, the *qconf* file has four major parts, each separated by a line of hyphens. This section discusses each part of the example file, so refer back to this example as you read the description of it. This example should make it clear exactly how the principles of MDQS discussed earlier are implemented.

The Qconf File — Parameters

The first portion of the *qconf* file has parameters that you can change, such as how long the daemon waits between checking for delayed queues, the filename of the *console* file, and the queue names and priorities for printing and batch jobs.

As you can see, the **Parameters** portion of this file is divided into three subportions. Each of these portions corresponds to a **sysadmin** menu, and are discussed separately.

The first part of **Parameters** defines all of the changeable parameters defined in the **sysadmin** Control Parameter menu. These parameters include:

console	Redirect the standard error output to this file.
openwait	If a daemon cannot open a device, it waits this many seconds before trying to open it again.
scanwait	Sets the amount of time a daemon is idle after a check of all the devices. On a system with no new or finished requests, this affects how frequently the daemon checks the delayed requests queue.
maxfailures	Defines the maximum number of times a server process can fail before the device it services is marked as "failed." If this happens you can restart the device using **qdev**.

sysmgr — The name of the MDQS system manager, which defaults to "mdqs." The daemon mails messages about orphaned notices to this address. It must be a valid login name.

The second portion of **Parameters** controls the default behavior of a printing job. It corresponds to the **sysadmin** Print Parameter menu. It has four components: the queue, the forms, the priority, and the print header:

print–queue — The name of the default print–queue. This name must be specified later in the **Queue Descriptions** portion of the file, or in the **sysadmin** MDQS Configuration Maintenance menu.

print–forms — Defines a name for the printing forms. For example, the default for the lp queue is "narrow." This is used primarily to distinguish between two parts of a queue, for example a part that goes to a narrow printing device or a part that goes to a wide printing device.

print–prior — Sets the priority for printing jobs. The values range from 1 to 10, with 5 the default value. The value 1 is the highest priority, and 10 the lowest.

print–hdr — The file that contains the line printer header. Normally, this is a null file.

The third part of **Parameters** sets the sets the value for parameters in the batch queue. This is equivalent to the Batch Parameter menu in the **sysadmin** program. It has three components: the queue, the forms, and the priority.

batch–queue — The name of the default batch–queue. This name must be specified later in the **Queue Descriptions** portion of the file, or in the **sysadmin** MDQS Configuration Maintenance menu.

batch–forms — Defines a name for the batch–forms. As with the print–form parameter, you can enter any value. This example uses Shell.

batch–prior — Sets the priority for **batch** jobs. The values range from 1 to 10, with 5 being the default value. The value 1 is the highest priority, and 10 the lowest.

The Qconf File — Device Descriptions

The second part of the *qconf* file contains device descriptions. You can change this portion of the *qconf* file using the MDQS Configuration Maintenance menu in **sysadmin**. The field <dname> contains the name of the logical device, and the field <device> contains the name of the real device. The logical device name provides an easier way of referring to a real device. On UTek, the real device is a file, but the logical device name that corresponds to that file can follow any naming convention that you choose.

The only case where a one–to–one correspondence of real to logical device name does not exist is for pseudodevices that service a network or run the **batch** command. As you can see from the example *qconf* file, the real device that corresponds to the batch and network logical devices is the special file */dev/null*, which discards data written to it.

The third field in **Device Descriptions** is <status>. This field contains options that control the behavior of a device. Following are the status options:

anyform	Indicates that this device can accept requests regardless of what forms were specified for the request. This is used for the network or batch devices.
roundrobin	Causes the daemon to use a round–robin algorithm to choose requests for a particular device. When two queues are mapped to the same device, this option causes the daemon to process a request from the first queue, followed by a request from the second queue. This is different from the default value that empties the first queue before accepting requests from the second queue.
skipmsg	Disables the sending of a message when a device completes a request. This is useful for the network device, because you want to see the completion message from the remote system device, instead of that from the network pseudo device.

The Qconf File — Queue Descriptions

The third portion of the *qconf* file, called **Queue Descriptions**, describes the logical queues where you can submit requests. You can change this portion of the *qconf* file using the MDQS Configuration Maintenance menu in **sysadmin**. The names that you designate for queues are not important, so long as you use a consistent scheme. In this file two queues, lp and Lp are on the home computer called **hammer**, while the other queues are for the **batch** requests and for other computers on the network. The queue descriptions are used by the MDQS submit programs (**batch** and **lpr**).

The Qconf File — Queue to Device Mappings

This last portion of the *qconf* file expands on the device names and queue names that you specified earlier in the file. This portion of the *qconf* file matches a queue name with a device name, and defines a server process that actually accomplishes the request. The queue name and the device name must be defined earlier in the file. As you can see in the example file, each queue that you defined in the **Queue Descriptions** portion of the file is mapped to a device name that you specified in **Device Descriptions**. The first field contains the queue name, while the second contains the device name. In this example, multiple queues are mapped to one device.

Currently there are five server processes available to execute requests to MDQS. All of these files are in the directory */usr/lib/mdqs*. When you enter them into the <server> field, you must enter their full pathnames. These include:

rawserver	Passes input to the device without filtering. This is useful for sending input to sophisticated printers that themselves define multiple printing modes.
plpserver	Sends input to a Printronix lineprinter.
lpserver	Sends input to a printing device. Before it sends the input to the device, it expands special characters into a two-character format. For example, tabs become spaces and a backspace followed by an newline is split over two lines.
shserver	Sets up input from the **batch** command to the null device, which always executes the shell.
netsend	This server process sends a request across the network to another computer. You must enter two arguments following */usr/lib/mdqs/netsend*. The first argument is the name of the computer to which you are sending the request, and the second argument is the name of the queue on that computer where the request is placed. The name of the remote queue is defined in the *qconf* file on the remote computer.

As you can see from the example file, the batch queue is processed by the server process **shserver**. The next entries are for the queue named **lp**. This queue is mapped to two different devices that correspond to a line printer with narrow forms and one with wide forms. The server for the **lp** queues is the server process **plpserver**, so the output is sent to Printronix line printers.

The next queue, **Lp**, is sent across the network pseudodevice, to a queue on the **shark** computer called **Lp**. The next queue, **sharklp**, is sent to a queue on **shark** called **lp**.

The remainder of the entries in the *qconf* file send requests to queues called lp on various machines. To find out what device and server process complete these requests on the remote machine you need to examine the *qconf* file of the remote machine.

Changing the Status of a Queue or Device

You can run several MDQS commands that give you information not available through the **sysadmin** program. One of the most useful is the **qstat** command. This command displays the status of the MDQS queues. It tells you what devices are active, what active requests are in the queues, and what delayed requests are in the queues. For more information on **qstat** see *UTek Command Reference, qstat(1).*

The **qmod** command lets you modify or delete an MDQS request. You can delete a request entirely, change the priority of requests, change the request to a new queue, or put a request on hold. You identify a request by specifying its job number, that is the job number you obtain using the C–shell **jobs** command. For more information on **qmod**, see *UTek Command Reference, qmod(1).*

The **qdev** command displays and modifies the status of a local device. It lets you disable a device and restart the current request or completely remove a request from a device. You can also use **qdev** to change the current form for a device, for example, to move from a line printer with narrow paper to one with wide paper. For more information on **qdev** see *UTek Command Reference, qdev(8).*

3A

Introduction to Editing Documentation

Available Editing Tools

This part of the *UN*X Tools* manual discusses the editing tools that are available on your workstation.

There are three editors available on your workstation:

- ed
- ex
- vi

The first editor **ed** is a line–oriented, instead of a visually–oriented, editor. This means that you usually look at one line at a time, unless you explicitly choose to have more lines displayed. You will find **ed** most useful for performing text editing within shell programs, where you cannot use a visual editor.

The second editor, **ex**, is a line–oriented editor, with many expanded features by comparison with **ed**.

The third editor, **vi**, is a visually–oriented editor. This means that you can move easily around the screen. The **vi** editor lets you move forward or backward, using units like words, sentences, and sections as points of reference. For most text entry we recommend that you use **vi**; it is slightly more difficult to learn than **ed** and **ex**, but much more versatile.

How to Use This Documentation

If you are not familiar with the basic commands of these editors, first read the section on editors in the book *Using the UNIX System*. *Using the UNIX System* gives you a grasp of the most often used commands.

When you want to learn more advanced uses of these editors, refer to sections 3B, *Advanced Uses of Ed*, and section 3C, *Advanced Uses of Ex and Vi*. So if you already know a particular editor, use this section to improve the your efficiency with it and as a reference manual.

3B

Advanced Uses of Ed

Introduction

This section is meant to help **ed** users use the text editor more effectively. You should have a knowledge of the material on **ed** in *Introducing the UNIX System.*

This section concentrates on the features of **ed** that save you time and accomplish more with one command. These features include:

- extending commands using special characters
- line addressing
- global commands
- cut and paste

Extending Commands Using Special Characters

Using special characters within **ed** makes it a much more flexible editor. Special commands discussed in this section let you concisely describe and address portions of the text you are editing.

Print and List Commands

Ed treats text files on a line–by–line basis. Two commands print the lines being edited, the **p** command and the **l** command.

The first is the print command (**p**). The following command prints all the lines in the file:

```
1,$p
```

The command:

```
s/abc/def/p
```

changes abc to def on the current line and prints the new version.

The list command (**l**) displays more information than the **p** command. In particular, **l** displays characters that are normally invisible, such as tabs and backspaces. For example, if a line contains tabs or backspaces, **l** prints each tab as > and each backspace as <. This makes it easier to correct typing mistakes adjacent to tabs or backspaces.

The **l** command also folds long lines. Any line that exceeds 72 characters continues on the next line. Each line except the last ends with a backslash (\) to indicate that the line was continued. A dollar sign is appended to the actual end of the line.

Substitute Command

The substitute command (**s**) changes an individual line. It accepts many ways of defining the strings that make up its arguments. This lets you easily make substitutions in the existing text.

A trailing global command after a substitute command is illustrated in the next example:

s/this/that/g

If there is more than one occurrence of *this* on the current line, the trailing **g** changes all of them. The trailing **g** command can be followed by **p** or **l** to print or list the contents of the line. For example:

s/this/that/gp
s/this/that/gl

Instead of performing the substitution on only the current line, the **s** command can specify a group of lines where the substitution is performed. For example:

1,$s/mispell/misspell/

This changes the first occurrence of *mispell* to *misspell* on every line of the file. The following command changes every occurrence on every line:

1,$s/mispell/misspell/g

If you add a **p** or **l** to substitute commands that affect multiple lines, only the last line that was changed prints.

You can use any character to delimit the strings of an **s** command. There is nothing special about slashes. Consider this line that contains a lot of slashes already:

//exec//sys.fort.go//etc . . .

You could use a colon as the delimiter to avoid confusing the delimiter with the strings that you enter. To delete all the slashes, enter:

s:/::g

Undo Command

Occasionally, you accidentally execute an incorrect **ed** command. The undo command (**u**) negates the last command you entered.

Metacharacters

In **ed**, certain characters have special meanings on the left side of a substitute command or in a search for a particular line. These are called metacharacters and include:

- period
- backslash
- dollar sign
- circumflex
- asterisk
- brackets
- ampersand

Although each metacharacter is discussed separately in the following text, you can combine them. An example of combined metacharacters is given in the *Circumflex* section.

Period

The period (.) on the left side of a substitute command or in a search stands for any single character. Thus the search:

```
/x.y/
```

finds any line where "x" and "y" are separated by a single character. This command finds lines like:

```
x+y
x-y
x y
x.y
```

Since the period matches any single character, you can match the invisible characters printed by **l**. For instance, if the **l** command prints the line:

```
...th\07is...
```

and you want to get rid of the \07 (bell character), enter:

```
s/th.is/this/
```

This command removes the bell characters, because the period matches the character between the *h* and the *i*, no matter what it is.

Since the period matches any single character, the command:

```
s/./,/
```

converts the first character on the line into a comma.

The period has several meanings in **ed**, depending on its context. This line shows all three:

```
.s/./,/
```

- The first period is the line number of the line being edited.
- The second period is a metacharacter that matches any single character on that line.
- The third period is the only one that is a literal period. On the right side of a substitution. the period has no special meaning.

Backslash

Since a period matches any single character, you cannot use it to indicate a literal period. To ensure that the period is read literally, you precede it with a backslash (\). A backslash turns off any special meaning of the following character. **Ed** considers the characters \. as a single literal period.

Use the backslash when you search for lines that contain a special character. If you are searching for a line that contains:

```
.PP
```

the command:

/.PP/

does not work. This search finds a line like:

THE APPLICATION OF ..

The above command doesn't work because the period matches the letter "A." To find only the lines with .PP, enter:

/\.PP/

The backslash turns off special meanings for characters other than the period. For example, to find a line that contains a backslash enter:

/\\/

A backslash can also turn off the special meaning of the user's erase character and the line kill character. When you add text with the append (**a**), insert (**i**), or change (**c**) commands, the backslash turns off special meaning only for the erase and kill characters.

Dollar Sign

On the left side of a substitute command or in a search command, the dollar sign ($) represents the end of the current line. Use the dollar sign on the left side of a search command to provide context. Consider the following phrase:

Now is the

You can add the word *time* to the end of this phrase by entering:

s/$/ time/

The second comma in the following line can be replaced with a period, without altering the first comma.

Now is the time, for all good men,

To replace the second comma with a period enter:

s/,$/./

The $ provides context to indicate which comma to replace. Without it, the **s** command replaces the first comma.

Like other metacharacters, the $ has multiple meanings depending on context. In the line:

```
$s/$/$
```

- The first $ refers to the last line of the file.
- The second $ refers to the end of the line.
- The third $ is a literal dollar sign to be added to the line.

Circumflex

The circumflex character (^), also called caret, represents the beginning of the line. For example, to search for a line that begins with *the*, enter:

```
/^the/
```

The circumflex narrows the context, because **ed** selects only lines that begin with *the*.

You can use the circumflex to insert something at the beginning of a line. For example:

```
s/^/<SPACE>
```

inserts a space at the beginning of the line. In this example, the symbol <SPACE> represents a blank space you enter using the space bar.

You can combine metacharacters. For example, to search for a line that begins with only the characters

```
.PP
```

Enter:

```
/^\.PP$/
```

Asterisk

You can use the asterisk (*) to match all spaces between two parts of a line with a single space. Consider, for example:

text x y *text*

where *text* stands for text, and there are any number of spaces between x and y. The asterisk in **ed** matches all the spaces between x and y.

A regular expression followed by an asterisk stands for an infinite number of consecutive occurrences of that regular expression. So you can match spaces, as the example above illustrates, or you can match an infinite number of occurrences of a regular expression.

To refer to all the spaces at once, use the command:

s/x<SPACE>*y/x y

The construction **<SPACE>*** means *as many spaces as possible.* So **x<SPACE>*y** means *an x, followed by as many spaces as possible, then a y.*

Since a period matches any single character, **.*** matches as many single characters as possible. Unless you are careful when you use the asterisk in conjunction with the period, the command can eat up more of the line than expected. Consider the line:

test x *text* x . . . y *text* y *text*

If you enter:

s/x.*y/x<SPACE>y

In this command the asterisk replaces everything from the first *x* to the last *y*. But you can turn off the special meaning of period with backslash. To change the above example enter:

s/x\.*y/x<SPACE>y/

Now the example works since **.*** represents multiple periods instead of any single character.

Be aware of some additional pitfalls associated with asterisk. Multiple characters means zero or more. Zero as a possible number of characters in a substitute command can produce strange results. For example, if the line contained:

text xy *text* x y *text*

and you enter the command:

s/x *y/x y/

the first *xy* matches this pattern, because it consists of an *x*, zero spaces, then a *y*. The substitution acts on the first *xy* of the example text and does not touch the later one, that actually contains intervening spaces.

To match spaces use a pattern like:

```
/x  *y/
```

Because this pattern has two spaces between the *x* and the *, it matches an *x*, a space, as many more spaces as possible, and then a *y*. In other words, it defines one or more spaces between x and y.

Zero is also a legitimate number of occurrences of an expression followed by an asterisk. The command:

```
s/x*/y/g
```

applied to the line:

```
abcdef
```

produces:

```
yaybycydyeyfy
```

Zero is a legal number of matches. Although there is no *x* at the beginning of the line, or between any two characters, the asterisk matches a "zero" occurrence of *x*.

Brackets

Frequently you can match any one of a group of characters. Just as in the shell, **ed** uses the [bracket] metacharacters to define a character class. The brackets mean that any single character inside the brackets can match the expression.

For example, in a substitute command, this pattern:

```
[0123456789]
```

matches any single digit. The pattern [0123456789]* matches zero or more digits. So the command:

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

You can put any characters into a character class; within the brackets there are no special characters. Even the backslash does not have a special meaning. This command searches for special characters within the brackets:

/[.\[]/

In this example, the left bracket ([) is not special.

To get a right bracket (]) into a character class, make it the first character inside the brackets. For example, enter:

/[].\$[]/

You can specify a range of digits within brackets as [0–9]. Similarly, [a–z] stands for the lowercase letters and [A–Z] stands for the uppercase letters.

You can also specify a character class that means *none of the following characters*. To do this, begin the class with a circumflex. For example:

[^0–9]

matches any character except a digit.

Enter the following to find the first line that does not begin with a tab or space:

/^[^<SPACE><TAB>]/

Ampersand

The ampersand command (**&**) matches strings that you have already entered. For example, if the original line contains:

Now is the time.

And it should contain:

Now is the best time.

Enter the command:

s/the/& best/

On the right side of the substitute command, the ampersand stands for whatever was just matched, so in this case it matches the "the" on the left side. This saves time if the text you are matching is long or if you are repeating a metacharacter that matches a lot of text. Using the ampersand also decreases the possibility of making a typing error in the replacement text.

The ampersand can occur more than once on the right side of a substitution. Using the example above, enter:

s/the/& best and & worst/

This command changes the original line to:

Now is the best and the worst time.

To get a literal ampersand, use the backslash to turn off its special meaning. For example:

s/ampersand/\&/

changes the word *ampersand* into the symbol.

Operating on Lines

Substituting Newline Characters

Ed lets you split a single line into two or more lines. You can substitute a newline character into the middle of the line, so that if a line is long you can divide it. Consider the line:

text xy *text*

You can break this line between the *x* and the *y* by inserting a backslash as the newline character. Enter:

s/xy/x\<RETURN>
y/

Although it is typed on two lines, this is actually a single command. It tells **ed** to split the line between *x* and *y*. Because a backslash turns off special meanings, the <RETURN> character is no longer special.

You can make a single line into several lines by inserting several backslashes.

When you create a new line, *dot* points to the last line you created. The name *dot* refers to the current line. So if you split one line into several lines, dot points to the last line.

Joining Lines

You can join lines together with the **j** command. Consider:

```
Now is
 the time
```

If dot is set to the first line, the **j** command joins the two lines together. The **j** command does not automatically insert a space between the newly–joined lines so a space is shown at the beginning of the second line.

By itself, a **j** command joins the current line to the line that follows it. But you can specify a group of lines to be joined by entering their starting and ending line numbers. For example:

```
1,$j
```

joins all the lines of the file into one long line.

Rearranging Lines

The ampersand (&) stands for whatever was matched by the left side of a substitute command. To reserve a part of the string on the left side you enclose it between \(and \). The enclosed portion is remembered and available for use on the right side of the command. On the right side, **\1** refers to whatever matched the first enclosed pair, **\2** refers to whatever matched the second enclosed pair, and so on.

As an example, consider this list:

```
Smith, A.B.
Jones, C.
```

Suppose that you want to place the initials in front of the last names. Enter:

```
1,$s/^\([^,]*\), *\(.*\)/\2 \1/
```

The first string enclosed by \(and \) matches the last name, which is any string up to the comma. The second string enclosed by \(and \) matches whatever follows the comma and any spaces. On the right side these two strings are referenced by \1 and \2.

Line Addressing

Line addresses in **ed** specify the lines on which **ed** commands operate. For example:

```
1,$s/x/y/
```

This command changes all lines in the file. The construction:

/string/

finds a line that contains *string*. Similarly,

?string?

searches backwards for *string*.

Unlike the substitute command, a search requires the slash and question mark as delimiters.

Address Arithmetic

When you have learned to address one line, you can use + and — to combine line numbers. So the command:

$—1p

prints the next–to–last line of the current file. The command:

$—5,$p

prints the last six lines of the current file. As another example:

.—3,+3p

prints from three lines before the current line to three lines after the current line.

You can save typing by entering — and + as line numbers by themselves. For example:

-

moves up one line in the file. You can also string several minus signs together to move up several lines.

You can use + and — in combination with searches that use /. . . /, ?. . . ?, and $. The entry:

/string/— —

finds the line containing *string* and moves dot two lines before it.

Repeated Searches

Suppose that you enter:

/string/

But *string* turns out not to be the occurrence of *string* you want. You can repeat the search again, without retyping it. The entry:

//

represents the string that was previously searched for.

This repetition of the search command also applies to the backward search. The entry:

??

searches for the same string, but in the reverse direction. It searches in the opposite direction of the last forward search or the last backward search.

You can also use the // command on the left side of a substitute command to represent the most recent pattern. For example, you can search forward or backward for a string, find it, then use // to represent that string in a substitute command. To substitute the word *good* for the last thing you searched for, enter:

s//good/p

Default Line Numbers

Two ways to make editing faster are to know what lines are affected by a command without an address, and to know where dot will be when a command finishes.

Using **ed** without specifying line numbers saves a lot of typing. No address is required with the following commands:

- **s** to make a substitution on a line
- **p** to print the line
- **l** to list the line
- **d** to delete the line
- **a** to append text after the line
- **c** to change the line
- **i** to insert text before the line.

Following are some indications of where to find dot after executing a command.

If there is no occurrence of *string*, dot stays where it was before the search began. This is also true if dot is on the only occurrence of *string* when you issue the command.

The delete command (**d**) leaves dot at the line following the last deleted line. However, if you delete the last line, dot points to the new last line.

Line–changing commands **a, c,** and **i** affect the current line if you do not specify a line number. Dot points to the last line you entered.

The read command (**r**) reads a file into the text being edited, either at the end of the file if no address is given, or after the specified line if an address is given. In either case, dot points to the last line that was read in. The **0r** command reads in a file at the beginning of the text, and the **0a** or **1i** commands let you add text at the beginning of the file.

The write command (**w**) writes out the edited file from the buffer to the original file on disk. Preceding the command by two line numbers causes a range of lines to be written. The **w** command does not change dot. This is true even if you enter a command that involves a context search. Since the **w** command is easy to use, regularly save the text you are editing in case the system crashes.

After the substitute (**s**) command, dot remains at the last line you changed. If there were no changes, then dot is unchanged.

Semicolon

Searches with /. . . / and ?. . . ? start at the current line and move forward or backward, until they find the pattern or return to the current line. Sometimes this is not what you want. Suppose, for example, that the buffer contains lines like:

```
.
.
ab
.
.
bc
.
.
```

With dot at line 1, enter the command:

```
/a/,/b/p
```

This command searches for *a* and *b* from the same point, so they both find the line that contains *ab*. Normally the comma separator ensures that *a* and *b* are on different lines, as it does for line numbers. But in a search command, **ed** does not reset dot after each address is processed. Each search starts from the same place.

In **ed**, you can use the semicolon (;) just like the comma, except that the semicolon forces dot to be set at that point when line numbers are evaluated. The semicolon moves dot. Following the example above, enter:

/a/;/b/p

This prints the range of lines that contain *ab* to *bc* because after the command finds *a*, dot is set to that line. Then **ed** searches for *b* from that address.

You can also use the semicolon separator to search for the second occurrence of a string. For example:

/string/;//

This command finds the first occurrence of *string* and sets dot there. Then it finds the second occurrence and prints only that line.

To search for the first occurrence of a string in a file, with dot positioned at an arbitrary location, enter:

1;*/string/*

Note, however, that this search fails if *string* occurs on line 1. You can enter:

0;*/string/*

to start the search at line 1.

Interrupting the Editor

If you interrupt **ed** by pressing <BREAK> or <INTERRUPT> while it is executing a command, the file is put back together again. The file is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable. If the file is being read from or written to, substitutions are being made, or lines are being deleted, these operations are stopped in a stable, but unpredictable state. Dot may or may not be changed.

If you interrupt **ed** while the **p** command is executing, dot does not remain on the last printed line. Dot returns to where it was when the **p** command started.

Global Commands

Basic Global Commands

The global commands **g** and **v** work on all lines of a file. The **g** command acts on those lines that contain a specified string. For example, the command:

```
g/THIS/p
```

prints all lines that contain the string *THIS*. The same rules and limitations apply to the strings of the global command that apply to the strings of the substitute command.

The **v** command is identical to **g**, except that it operates on those lines that do not contain an occurrence of the string. So the command:

```
v/^\.p
```

prints all the lines that do not begin with a period.

Almost any command can follow **g** or **v**. For example:

```
g/^\./d
```

This command deletes all lines that begin with a period. The command:

```
g/^$/d
```

deletes all blank lines.

The substitute command often follows the global command, since you frequently want to change all occurrences of a string. For example, to change *This* to *THIS* everywhere in a file, enter:

```
g/This/s//THIS/g
```

Global commands make two passes over the file. On the first pass all lines that match the pattern are marked. On the second pass, each marked line is examined separately. Dot is set to that line, and the command is executed. The command that follows a **g** or **v** can address a location in the file. For example:

```
g/^\.PP/+
```

This command prints the line that follows each .PP command. The command:

g/topic/?^.SH?1

searches for each line that contains the string *topic*, scans backwards until it finds a line that begins with .SH, and prints the line that follows.

You can also precede the **g** and **v** commands by line numbers. In this case, **g** and **v** search only the lines you specify.

Multiline Commands

You can do more than one command in conjunction with a global command. As an example, suppose the task is to change x to y and change a to b on all lines that contain *string*. Enter:

```
g/string/s/x/y/\
s/a/b/
```

The backslash at the end of the first line says that the commands continue on the next line.

When you use the **g** command in conjunction with more than one other substitute command, you cannot use // to signify the previous pattern. Because **g** works sequentially through a file, the previous pattern executed could be either of the substitutions that you requested.

You can use the **a, c** and **i** commands with a global command. As with other multiline constructions, add a backslash at the end of each line except the last. For example, to add a **.nf** and **.sp** command before each **.EQ** line, enter:

```
g/^\/.EQ/i\
.nf\
.sp
```

It is good practice to check that the global command did only what you wanted. If the global command makes changes that you do not want, you can undo them using the **u** command.

Cut and Paste

Often cut–and–paste operations are necessary to bring various pieces of text together into a meaningful document. You can use the file–manipulation utilities of both UTek and **ed** to manipulate text.

The UTek operating system lets you:

- change the name of a file
- make a copy of a file elsewhere
- combine files
- remove a file

The **ed** text editor lets you:

- insert one file into another
- split a file into pieces
- move lines within a file
- copy lines

Moving large blocks of text around can be confusing. But defining the task you want to accomplish and checking each command before you enter it make cut–and–paste operations as easy as using other commands.

UTek Commands

UTek file–manipulation commands make it easy to get access to portions of text scattered throughout different files. They also make it easier to manipulate very large blocks of text.

Changing Filenames

If there is a file that needs to be renamed, use the move command (**mv**). It moves the file from one name to another. For example:

mv *file1 file2*

NOTE

If there is already a file with the new name, its contents are overwritten with information from the file you moved.

Copying Files

Sometimes you need a copy of a file because the original version serves another purpose, or because an extra copy gives you a backup. To make an extra copy of a file, use the **cp** command. For example:

cp *file1 file2*

This command creates two identical copies of *file1*. If *file2* previously contained information, it is overwritten.

In summary, **mv** renames a file; **cp** makes a duplicate copy. Both commands overwrite any existing version of that filename with the newly–created version.

Combining Files

You can collect two or more files into one larger file using the **cat** command.

To combine two files into one larger file, **cat** the two files together and redirect the output to a new file. Enter:

cat *file1 file2> newfile*

Sometimes you need to append a file to the end of another file. For example, enter:

cat *file1 >>file2*

This command puts *file1* at the end of *file2*.

Removing Files

If you no longer need a file as part of your editing or writing project, remove it using the **rm** command:

rm *file*

Ed Commands

Ed manipulates pieces of files, individual lines, or groups of lines.

Reading and Writing Files

It is important to know the **ed** commands for reading and writing files.

Within **ed**, the **e** command edits a new file without leaving the text editor. For example:

e *file*

This command discards whatever is being worked on and starts over on a new file.

When you enter **ed** with the command:

ed *file*

ed remembers the name of the file, and any subsequent **e, r,** or **w** commands that do not contain a filename refer to this remembered file.

You can find the current filename simply by typing **f**. You can also change the name of a remembered file with **f** *file*. For example:

```
ed precious
f junk
```

This command obtains a copy of the file *precious* and guarantees that subsequent **w** command without the filename writes to *junk*, instead of to the original file.

Inserting One File Into Another

To insert one file into another, use the **r** command. For example, to insert the file *table* just after a reference to Table 1, enter:

```
/Table1/
Table 1 shows that ...
.r table
```

The important line is the last one. The **.r** command reads a file in after dot. An **r** command without any address adds lines to the end of the file.

Writing Part of a File

You can also write part of the document that you are editing to another file. For example, you could remove a table to format or test it separately. To remove a table and write it to a file called *table*, enter:

```
/^\.TS/
.TS
.,/^\.TE/w table
```

You can use all of the various ways that **ed** addresses lines to select what lines are written to another file.

Moving Groups of Lines

There are several ways to move groups of lines from one location in a file to another. You can write the group of lines to a temporary file, read it in where it is needed, and delete the original location. An easier way is to use the move (**m**) command. Like many other **ed** commands, the **m** command accepts up to two line numbers in front of the command to specify the lines on which it operates. You can also follow **m** with a line number that tells where to move the lines. The command:

line1,*line2***m** *line3*

moves all the lines from *line1* through *line2* after *line3*.

You can also specify lines on which **m** operates by searching for patterns.

When you use the **m** command, it is important to verify each step. If you incorrectly specify a line or move the wrong lines, the mistake can be difficult to undo. Before doing complicated **m** commands use the **w** command to write the file. This lets you recover a previous version of the file, if necessary.

Copying Lines

The **ed** program provides a transfer command (**t**) that copies one or more existing lines. The **t** command is identical to the **m** command, except that instead of moving lines it duplicates them at the address you specify. For example:

1,t

This command duplicates all the lines of a file and places them at the end of a file.

Marks

The **ed** program can mark a line with a particular name. Later you can reference the line by its name, regardless of its line number. This feature is useful for moving lines and for keeping track of them as they move. To mark a line, use the **k** mark command. When you give the mark a name, it is a single lowercase letter. So the command:

k*x*

marks the current line with the name *x*. You can then reference the mark line with the address:

'x

Marks are most useful for moving blocks of text. For example, mark the first line of the block you want to move with **ka**. Then mark the last line of the block you want to move with **kb**. Then move dot to the line just above where you want the block to go. Enter:

'a,'bm

NOTE

A particular mark name can be associated with only one line at a time.

Temporary Shell Invocation

Sometimes you want to temporarily invoke the UTek shell from **ed** and execute a UTek command, without leaving the editor. The escape command (**!**) provides a way to do this. Enter:

!*UTek command*

The current editing is suspended, and the command executes. When the command finishes, **ed** prints another **!** and you can resume editing.

By using **!** in **ed**, you can execute any UTek command, including another invocation of **ed**. If you escape to another invocation of **ed**, you can repeat the **!** command.

3C

Advanced Uses of Ex and Vi

Introduction

Ex is a line-oriented editor, much like **ed**, but with extended commands and features. **Vi** is the visually-oriented UTek editor, and this section focuses primarily on **vi**, since it is a more versatile editor. **Vi** is versatile because you can execute any **ed** or **ex** command while you are running **vi**. So a fundamental knowledge of **ex** and **ed** is helpful to use **vi** effectively. For more information on **ed** and **ex**, see *Introducing the UNIX System.*

Any of the commands in this section that begin with a colon are commands you can use in **ex**, as well as in **vi**. If you execute these commands from **ex**, the colon is actually the prompt, so do not enter the colon. If you execute these commands from **vi**, you must enter the colon.

Vi runs in what is called visual or open mode, while **ex** runs in a line-oriented mode. From **ex** you can invoke the visual mode of **vi** by entering **vi**. To exit the visual mode and revert to the line-oriented mode of **ex**, enter **:Q**.

Invoking Vi

To invoke the **vi** command, use the general form:

vi *filename*

This command invokes the **vi** editor to let you edit the file you specified. The cursor begins at the first line of the file.

There are variations on this command that let you invoke **vi** in special ways. These variations put you in a certain place in the file, set the window, or edit the file without making changes to it. You can also edit a file by specifying one of the tagnames it contains.

To begin editing the file on the last line, instead of the first line, enter:

vi + *filename*

Similarly, to start editing a file at a particular line number enter: There is no space between the addition sign and the line number.

vi +*linenumber*

Another way to start editing at a specified point in the file is to search for a particular string of characters and begin editing there. To begin editing at a particular string enter:

vi *+/string filename*

As you invoke **vi** you can also set the number of lines in the window. Enter:

vi -w*n filename*

In this command, *n* is the number of lines you want in the window. If you set the window for more lines than your terminal can display at once, the terminal cannot show all the lines in the window. But the larger window size is recognized by the commands that scroll a window or a half-window at a time.

You can also set the number of lines in the window using the special window option. See the *Special Options* section.

The system prompts you for the key used when the file was originally encrypted. Enter the key, then you can use **vi** to edit the file.

Instead of using the **cat** or **more** commands to examine a file, you can use **vi**. With **vi** you can search for particular parts of the file. To examine a file without writing changes to it, enter:

view *filename*

or enter:

vi -R *filename*

Either of these invocations of **vi** allows you only to read the file. Although **vi** lets you change text when invoked with the **view** command or the **-r** option, you cannot write those changes to the file. So if you accidentally make changes to the file, **vi** only makes them in the temporary buffer.

When you are editing program source code, you can edit a file that contains a particular function, just by knowing the name of the function. Function names are stored in special *tagname* files. Tagnames are the names of functions and definition types for C, Pascal or Fortran source code. You can create a file of tagnames using **ctags**. (See *UTek Command Reference ctags(1)*). To edit a file that contains a particular function enter:

vi -t *tagname*

Text Insertion

Nearly all the commands you need to know for text insertion are covered in *Introducing the UNIX System*. This section reemphasizes some commands discussed there and introduces some new ones.

Changing back and forth between command mode and insert mode requires extra time. **Vi** provides two commands that begin in command mode, let you make quick changes to a line, and return you to command mode.

The **S** command begins in command mode and substitutes for the current line, no matter where the cursor is on the line. This command first erases the line, then goes into insert mode and lets you insert the text to replace that line. The **S** command begins inserting text at the existing indentation of the line.

Another feature that makes text insertion faster is the **vi** option called *wrapmargin*. This option automatically inserts <RETURN> at the end of a line on your screen, so that you don't have to insert one. Like all other options, it can be set in several places. (See the *Options* section.) For example, to set it within **vi** itself, from command mode enter:

:se wm=10

This command begins wrapping text onto the next line when the text you are entering is within 10 characters of the right margin. Because you do not have to enter <RETURN> at the end of every line, setting the wrapmargin option saves a great deal of editing time.

Vi also provides alternate ways of entering tab characters while in insert mode. To tab forward, enter:

<CTRL-T>

To tab backward, enter:

<CTRL-D>

These two ways of tabbing may be more convenient for you, depending on whether or not your terminal has a tab character and where it is located on the keyboard.

Inserting Control Characters

A special command in **vi** lets you insert a control character in the text. While in insert mode, enter:

<CTRL-V> *control-character*

For example, to insert a <CTRL-M> (carriage return), enter:

<CTRL-V> <CTRL-M>

Cursor Movement

This section describes some additional ways of moving the cursor around the screen that make **vi** more efficient to use. The first portion describes commands that move the cursor over characters, and the second portion describes commands that move the cursor over sentences, paragraphs, and sections.

Characters

Again, most commands for moving over characters and words, and searching for particular strings, are discussed in *Introducing the UNIX System.*

When editing text you usually search for a particular character or string, or move around relative to words. But you can also use each character on the screen as a column number, and move to a particular column. To move to the first column of a line, from command mode enter:

0 (zero)

To move to a particular column other than the first, enter that column number and the pipe character (|). For example, to move to column 40 enter:

40|

Treating characters on the screen as columns can be very useful if you are creating a table, or some other visual display. Addressing a column number can move the cursor to a white space that you cannot address with word or search commands.

To move to the beginning or ending character on the line, you can use the ^ and **$** commands. To move to the first character that is not a space, from command mode enter the caret character:

^

To move to the last character that is not a space, from command mode enter a dollar sign:

$

Sentences, Paragraphs, and Sections

Effective editing of text involves moving around quickly, in units that you can easily identify. When you are entering English text, its units of meaning are sentences, paragraphs and sections. This discussion reviews how to move over each of those units of text.

Sentences

Two commands move to a sentence, **(** and **)**. The **)** command moves forward to the next sentence, and the **(** command moves backward to the previous sentence. An easy way to distinguish these commands is to remember the direction in which the parentheses point.

The **)** and **(** commands always move the cursor to the beginning of the sentence. In the case of the **)** command, the cursor moves to the beginning of the next sentence. The following example shows how the cursor moves when you enter **)**. The cursor begins at "the" and ends at "This" (as shown by italics). The position of the cursor is shown in italics.

This is *t*he first sentence. *T*his is the second sentence.

To move to the previous sentence using the **(** command, you need to be at the beginning of the current sentence. If the cursor is not at the beginning of the sentence, and you enter **(**, the cursor moves to the beginning of the current sentence. The following example shows how the cursor moves when you enter **(**. The cursor begins at "That" and ends at "This" (as shown by italics).

*T*his is the first sentence. *T*hat is the second sentence.

Paragraphs

The commands to move forward and backward to a paragraph are } and {. Like the parentheses, } moves to the next paragraph and { moves to the previous paragraph. Both these commands work by recognizing the commonly-used text formatter codes for paragraphs. When you enter the command, the cursor moves to the line that contains the formatting code to begin the paragraph, instead of moving to the first line of the paragraph itself.

Vi recognizes the following paragraph codes:

.IP
.LP
.PP
.QP
.P
.LI
.pp
.lp
.ip
.bp

In addition to these paragraph codes, the { and } commands also recognize blank lines as paragraph delimiters.

You can make **vi** recognize other text formatting codes for paragraphs by setting the special paragraphs option. See the section *Setting Options* for information on setting the paragraphs option.

The following example shows the cursor movement when you enter }. The cursor begins at .PP and ends at .LP (as shown in italics).

.PP
The paragraph begins with this sentence. This is the second sentence.
And this is the third sentence. They comprise a paragraph.
.LP

Sections

Vi has two commands that move to sections, **[** and **]**. The **]** command moves forward to the next section, and **[** moves backward to the previous section. **Vi** recognizes sections by recognizing these commonly-used text formatter codes for sections:

.NH
.SH
.H
.HU
.nh
.sh

If none of these section codes are present in the file, the **[** and **]** commands move the cursor to the beginning or the end of the file.

The section commands can recognize other section codes. You can add these codes using the sections option. See the *Setting Options* section for information on setting the "sections" option.

The following example shows the cursor movement when you enter]. The cursor begins at .SH and ends at .NH (as shown in italics).

.SH
This is a section. It has several paragraphs.
.PP
This is the second paragraph
.NH

The following example shows the cursor movement when you enter [. The cursor begins at .NH and ends at .SH (as shown in italics).

.SH
This is a paragraph. It is part of a section.
.PP
This is another paragraph. It precedes a section command.
.NH

The Screen

Quickly moving the cursor around in the file and changing the characteristics of the screen can make your editing task much easier.

To erase the screen and redraw it, enter:

<CTRL-L>

Two of the most commonly used screen commands are <CTRL-D> and <CTRL-U>. The first command scrolls down half a screen, and the second scrolls up half a screen. The default value for scrolling is 15 lines. If you want to move the screen in smaller increments, you can reset the number of lines that scroll when you enter these commands. For example, to set scrolling to six lines, enter:

6<CTRL-D>

This command sets the scrolling for both <CTRL-D> and <CTRL-U> to to six lines, during the current editing session.

You can also set scrolling to scroll more than the current default. For example, to set scrolling to 25 lines, enter:

25<CTRL-U>

This command sets the scrolling for both <CTRL-D> and <CTRL-U> to 25 lines.

Sometimes you need to see only one more line at the top or the bottom of the screen to make a particular change. To scroll down one line enter:

<CTRL-E>

To scroll up one line enter:

<CTRL-Y>

To move to other parts of the file, you can enter a colon, then an **ed** or **ex** line address. But you can also move to particular lines within **vi**. To move to the end of the file enter:

G

To move to line 223, enter:

223G

Cut and Paste

Moving blocks of text around can be confusing. But defining the task you want to accomplish and checking each command before you enter it make cut-and-paste operations as easy as using other commands.

A common cut-and-paste operation is to copy a portion of a file, put that copy somewhere else, and modify it. To do this, use the yank command, **y** for words, and **yy** or **Y** for entire lines. The yank command does not delete text. It merely copies the text into a buffer. For example, to yank five lines, beginning at the current line, enter:

5yy

After you yank the copy of these five lines, you can put them back into the text using the **p** or **P** command. The **p** command puts the lines back after the current position of the cursor, and the **P** command puts the lines back before the current position of the cursor.

In addition to using the yank and put commands to copy existing text to another location in the file, you can use them to move a block of text from one place to another.

To move a block of text, use the yank command to copy the text into a temporary buffer, use the put command to insert it in its new location, and delete the text at its original location. The following example illustrates the movement of a block of text using the yank, put, and delete commands. As you work through the example, refer to Figure 3C–1 for a visual representation.

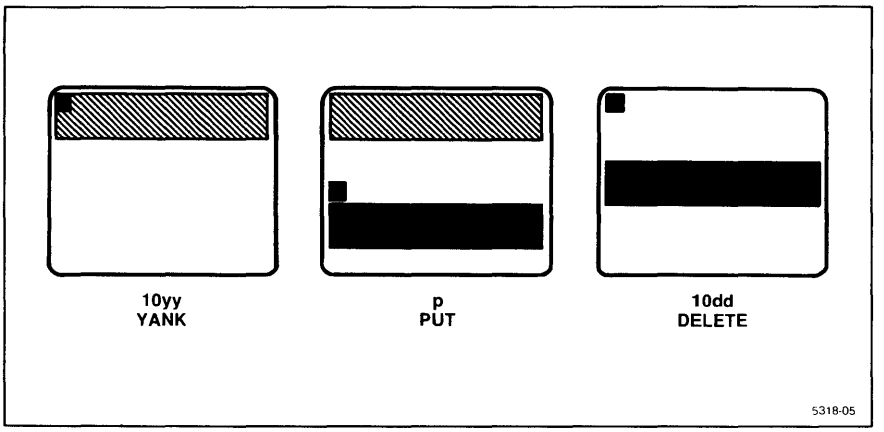


Figure 3C–1. Moving Text Using yank and put.

Suppose that you have a ten–line block of text that you want to move farther down in the file. Position the cursor at the first line of the block and enter:

10yy

This copies the next ten lines into a temporary buffer. Move the cursor to the line above the planned insertion and enter:

p

The **P** command inserts the ten lines you copied from the original location to the location just below the cursor. To delete the original block of text, move the cursor to the first line of that block and enter:

10dd

Using Marks to Address Lines and Move Text

When you are editing text, you usually think of it as a composite of logical units and sub-units. Instead of addressing these units by line number, or according to text patterns at the beginning and end of the unit, you can use the mark command (**m**) to delimit the logical units of the text. To mark a line, put the cursor there and enter:

m*z*

In this command *z* is any lowercase letter. If you mark two lines with the same letter, the first line is no longer marked.

After you have marked the line, you can use the **ex** command *'z* to refer to that mark. For example, to move to the line you have marked with the letter *a* enter:

:'a

You can use these marks in conjunction with the move (**m**) command in **ex** to move blocks of text. Let's work through an example of using the mark commands to move blocks of text. Refer to Figure 3C–2 for a visual representation of the example.

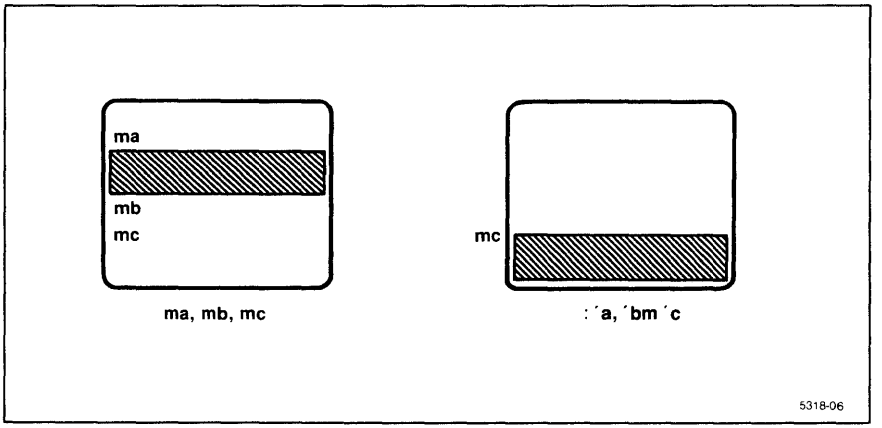


Figure 3C–2. Moving Text Using mark and move.

Suppose that you have a paragraph you want to move further down in the text. Go to the first line of the paragraph and enter:

ma

Then move the cursor to the last line of the paragraph and enter:

mb

Finally, move the cursor to the line above the planned insertion and enter:

mc

To move the paragraph just below the third mark (c) enter:

:'a,'bm'c

This command says *from mark a to mark b, move to mark c.* This method is more efficient than using the yank, put, and delete commands because the block of text is moved directly instead of being copied and then deleted.

Numbered Buffers

Whenever you delete text, it is put into *numbered buffers*, 1–9. The last text that you deleted is in buffer 1, the next–to–last text in buffer 2, and so on. So if you want to move text to another place, you can delete it and retrieve it from the numbered buffer To retrieve the deleted text, move the cursor to the line above where you want to put it and enter:

"*n*p

The quotation mark (") says that a buffer follows, and *n* is the number of the buffer. Normally, if you delete text and immediately move it elsewhere, this number is 1. The **p** command replaces the deleted text after the current line. You can also use the **P** command to put the text before the current line.

If you have already retrieved text from the first buffer, to recover text from the next buffer enter a period (.):

.

So if the original command recovered text from buffer 1, this command recovers text from buffer 2.

Named Buffers

One disadvantage of the numbered buffers is that they are cleared when you leave the editor to move from one file to another. Instead of using numbered buffers, you can use named buffers to save text. There are 26 named buffers available, and they are referenced by the lowercase letters a–z. For example, you can yank five lines into buffer *a* using the command:

"a5yy

You can then move the cursor to the line above where you want to put that text and enter:

"ap

You can retrieve text from the buffer using the **p** or **P** commands. To move text from the file where you are working into another, you could yank the text into a named buffer, enter **:e** *otherfile*, retrieve text from that buffer, and place it in the second file.

You can yank or delete pieces of text from several locations in the file and consolidate them into one buffer. You can place text into a named buffer, and then append more yanked or deleted text at the end of that buffer. To build up a buffer use the uppercase letter that corresponds to the buffer name. For example, to delete the current line of text and add it to buffer *c* enter:

"cdd

The current line of text is appended to the end of the existing text in buffer *a*, and you can continue to build up the buffer.

Macros and Abbreviations

Using macros and abbreviations reduces the number of keystrokes that you enter, so it saves a great deal of time. Using macros, you can program one keystroke to replace a sequence of **vi** commands. For example, you could program the command to exit **vi** (**ZZ**) to be equivalent to one keystroke.

You can also abbreviate frequently–used words. You define the abbreviation for the word, then enter that abbreviation as you insert text. **Vi** immediately replaces the abbreviation with the full word. Macros and abbreviations are discussed next.

Macros

To define macros, use the **:map** command. This command takes the general form:

:map *input-key commands*

The following example makes the current word plural when you type #. To define the macro, from command mode enter:

:map # eas <ESC>

Now each time you enter # from command mode, the current word becomes plural (the cursor goes to the end and appends an *s*) and you are returned to command mode. The **:map** command applies during the current editing session. If you want to set a macro permanently, enter it into your *.exrc* file or set it in your environment. See the section *Setting Options* for details.

It is important not to define a key for use with a macro if that key has another important function. Defining a key for use with a macro replaces its original meaning. Keys that **vi** does not use include: g, v, q, K, V, and Z. Many control characters and special characters are also undefined.

A more complicated example of macro definition involves replacing font commands with a single keystroke. The **troff** command to italicize a word is \fI*word*\fR. To map this sequence to the letter *y* enter:

:map y i\fI <ESC> Ea\fR <ESC>

To use this macro, from command mode enter **y** at the beginning of the word you wanted to italicize.

A more versatile way of italicizing words involves inserting the command as you are inserting the text, instead of going back later to italicize words. A variation of the **:map** command lets you invoke macros in insert mode. This variation is the **:map!** command. For example, you can define two different macro characters to italicize a word while you are in insert mode. The first macro substitutes \f2 when you press <CTRL–O>, and the second macro substitutes \f1 when you press <CTRL–K>. This example uses the <CTRL–O> and <CTRL–K> keys. Enter:

:map! <CTRL–V> <CTRL–O> \f2
:map! <CTRL–V> <CTRL–K> \f1

Recall that <CTRL–V> lets you inserts another control character when you are in insert mode. While you are in text insert mode, type <CTRL–O>, and **vi** automatically inserts the string \f2. Then enter the word you want to italicize, followed by <CTRL–K>. <CTRL–K> inserts the string \f1. Throughout the whole process you remain in insert mode, so you can continue inserting text without hitting the escape key.

When you define a macro, it is only valid in the current editing session. To have the macro available whenever you log in you must define it in your *.exrc* file or in your EXINIT environment variable. See the section *Setting Options* for details.

Abbreviations

You use macros from command mode, and they are designed to replace consecutive **vi** commands. Abbreviations are used in insert mode to replace words that you type often. Then when you are entering text, you can type the abbreviation for a word, and it expands into the complete word. You define abbreviations using the **:ab** command. Suppose, for example, that you frequently type the word *Pascal*. To define the letters *pa* as an abbreviation for Pascal, enter:

```
:ab pa Pascal
```

Each time you enter **pa** preceded by a space or a text insert command, or followed by a space, <ESC>, or <RETURN>, it expands to *Pascal*. Recognition of the spaces or other special characters ensures that *pa* is abbreviated only when it is typed by itself, and not when it is part of a word.

Setting Options

As you have seen in previous sections, **vi** provides special options that affect the editing environment. All of the options have default settings provided by **vi**, but you can reset any of these options to meet your needs. A detailed list of all the options follows. Some of the features controlled by options include: the window, automatic wrapping of the right margin, automatic indentation of text for writing programs, and turning off the special meaning of nearly all metacharacters.

There are several ways to change **vi** options from their default settings.

- the EXINIT variable in the *.login* file
- the EXINIT variable in the *.cshrc* file
- the EXINIT variable in the *.profile* file
- the *.exrc* file
- within **vi** itself

If you do not change many of the options from their default values, it is easiest to set them using the EXINIT variable. If you use C-Shell, set the options in the *.login* or *.cshrc* file. If you use the regular shell, set the options in the *.profile* file.

For example, the *wm = n* option sets the wrapmargin to *n* characters. When the cursor arrives within *n* characters of the right margin, it automatically inserts <RETURN>. If you use C-shell, to set the wrapmargin option to 5, enter:

```
setenv EXINIT "se wm=5"
```

If you use the Bourne shell, to set the wrapmargin option to 5, enter:

```
EXINIT="se wm=5"
```

If the entries for **vi** options in the *.login*, *.cshrc*, or *.profile* become too long, you can create a *.exrc* file in your home directory. The *.exrc* file is almost identical to a *.login* or *.profile*, except that the entries are not enclosed in quotes. For example:

set wm=5

Now each time you invoke **ex** or **vi**, the options in the *.exrc* file are read.

NOTE
If you create a .exrc file, be sure to delete the EXINIT line from your .login, .profile, or .cshrc file.

To set macros and abbreviations permanently, you can define them in the *.login*, *.profile*, *.cshrc*, or *.exrc* file. If you define them in a *.exrc* file, put them on a separate line from the **vi** options. For example:

set wm=5
ab tek Tektronix, Inc.
map K ZZ

If you define macros and abbreviations in a *.login*, *.profile*, or *.cshrc* file, all the information is on one quoted line. Use the vertical bar (¦) to separate the macros and abbreviations from the normal options. For example:

setenv EXINIT "se wm=5 ¦ ab tek Tektronix, Inc. ¦ map K ZZ"

The third way to set **vi** options is to set them temporarily, within **vi** itself. To do this use the **:se** command. This command has the general form:

:se *option*

In this command, the option is an abbreviation for the option name. For example, earlier we used *wm=5* to stand for *wrapmargin=5*. Although you see the full name for each option, they can be entered as abbreviations into the *.login*, *.cshrc*, *.profile*, or *.exrc* files, or when temporarily set using **:se**.

Following is a description of all the **vi** options that are available. The abbreviation for the option name and the default value for that option are also included.

autoindent Use the autoindent option to enter programs that require indentation. If you are appending text to the end of the file, autoindent automatically uses the same indent as the previous line. When you are inserting text using the append, open, insert, or change commands, it is inserted using the same indentation as the line that is the point of reference. The change command adopts the same indent as the original version of the line. To turn off the autoindent for the current line enter ^<**CTRL–D**>. The autoindent resumes at the next line.

Default: no autoindent
Abbreviations: ai, noai

autoprint When you change a line using an **ex** command (for example the substitution command) the line that you changed prints.

Default: autoprint
Abbreviations: ap, noap

autowrite The autowrite option makes **vi** write permanent changes to a file whenever you leave the editor. If you interrupt **vi** using **:stop**, changes are written before **vi** stops.

Default: noautowrite
Abbreviations: aw, noaw

beautify When you set the beautify option, no control characters are allowed in the text that you insert, except tab <CTRL–I>, newline <CTRL–L>, and form feed <CTRL–M>.

Default: nobeautify
Abbreviations: bf, nobf

directory The directory option sets the temporary buffer file where you edit a copy of the permanent file.

Default: directory = /tmp
Abbreviation: dir

errorbells If you are on a dumb terminal, the errorbells option precedes **ex** error messages with a bell. If you have an intelligent terminal, **ex** always places error messages in reverse video on the terminal.

Default: noerrorbells
Abbreviations: eb, noeb

hardtabs The hardtabs option gives the boundaries on which terminal hardware tabs are set. This option is set in your terminal variable, so the default is 0.

Default: hardtabs = 0
Abbreviation: ht

ignorecase The ignorecase option maps all uppercase characters in the text to lowercase characters for regular expression matching. All uppercase characters in regular expressions are mapped to lowercase characters, except when you specify a character class.

Default: noignorecase
Abbreviations: ic, noic

keypad The keypad option reads the TERMCAP entry for your terminal type, to determine how to program keys available on your particular keyboard. These programmed keys are invoked when you invoke **vi**. How the keys are programmed depends on the kind of terminal you have. This option causes the *ks* sequence in the TERMCAP entry to be invoked as you invoke **vi**, and the *ke* sequence in the TERMCAP entry to be invoked as you quit **vi**. If you set this option to *nokeypad*, these sequences are not read and the keys are not programmed.

Default: keypad
Abbreviations: ke, noke

lisp The lisp option must be used in conjunction with the autoindent option. Lisp changes the meaning of the autoindent, and of the **(**, **)**, **[**, **]**, {, and } commands. Instead of matching sentences, paragraphs, and sections, the commands identify units of lisp code. The lisp option always remembers the last indent of the previous line. To set up a file for editing with lisp, you can invoke it using **vi -l**. This automatically sets the autoindent and lisp options.

Default: nolisp
Abbreviations: lisp, nolisp

list The list option displays lines showing tabs as <CTRL–I> and ends of lines with $.

Default: nolist
Abbreviations: list, nolist

magic The magic option gives all the metacharacters used in regular expression matching their special meaning. If the magic option is not set, only $ and ^ retain their special meaning. If you do not set the magic option, you can restore special meaning to metacharacters by preceding them with a backslash.

Default: magic
Abbreviations: magic, nomagic

mesg The message option lets other users write to your terminal. If this option is set to *nomesg*, write permission to your terminal is turned off.

Default: mesg
Abbreviations: mesg, nomesg

modeline The modeline option lets you insert UTek commands into a file. These commands are executed the next time you invoke that file with **vi**. The syntax of the modeline entry is: **vi:***command*:. For example, you could insert the line **vi:$:** into the file to begin editing at the last line of the file. Not setting the modeline option means that **vi** can read a file you want to edit into the buffer more quickly, because it does not have to check each line for a modeline command.

Default: nomodeline
Abbreviations: mo, nomo

number The number option displays line numbers on your terminal. Visible line numbers make it easier to execute line-oriented **ed** commands. The numbers are only points of reference; they are not actually part of the file.

Default: nonumber
Abbreviations: nu, nonu

open When the open option is set, the **ex** commands **vi** and **open** are allowed. If the option is not set, you cannot execute these commands.

Default: open
Abbreviations: open, noopen

optimize The optimize option is designed for terminals that run at a slow baud rate, to make the output display on the screen more quickly. The terminal does not insert automatic carriage returns when printing more than one line of output.

Default: nooptimize
Abbreviations: opt, noopt

paragraphs The paragraphs option specifies what text-formatting commands are recognized by the { and } commands. To display the text-formatting commands this option recognizes, enter **:se para**.

Default: paragraphs = IPLPPPQPP LIppIpipbp
Abbreviation: para

prompt When you set the prompt option, the command mode of **ex** uses the : (colon) prompt. When the prompt option is not set, **ex** resembles **ed** by not displaying a prompt.

Default: prompt
Abbreviations: prompt, noprompt

readonly This option allows you only to read the file. **Vi** does not let you make permanent changes to the file. This is exactly like the **view** command. If you set this option to *noreadonly* while using **view** on a file, you can make permanent changes to the file.

Default: noreadonly
Abbreviations: readonly, noreadonly

redraw The redraw option redraws your entire screen instead of writing over characters. On a dumb terminal, this simulates the output of an intelligent terminal. Generally, this option is only used at high baud rates. Although the default value for this option is "noredraw", often it is set automatically. The redraw option is automatically set if you have a terminal that has a AL TERMCAP entry for *add line*, and a DL TERMCAP entry for *delete line*.

Default: noredraw
Abbreviations: redraw, noredraw

report The report option sets the number of lines that must be involved in a command before a report of what the command did can display on your screen. For example, if you delete 10 lines, and report = 5, report displays a message on your screen, saying that 10 lines have been deleted. Enter **report**=*n*, where n is the number of lines affected before the activity is reported to you. Enter **:se report** to see the current value of the report option.

Default: report = 5
Abbreviation: report

scroll The scroll option sets the number of lines that scroll when you enter the <CTRL–D> command in **ex**. The **z** command in **ex** scrolls double the amount of the <CTRL–D> command, so setting the number of lines that scroll also affects the **z** command.

Default: scroll = one–half the value of the window option
Abbreviation: scroll

sections The sections option defines the text–formatting commands for paragraphs that **vi** recognizes when using the **[** and **]** commands to move the cursor to the next or previous paragraph.

Default: sections = NHSHH HUnhsh
Abbreviation: sections

shell The shell option gives the pathname of the shell that is invoked when you use the **!** command to escape the editor temporarily. The default value of this shell is taken from the SHELL environment variable. To display the current shell option, enter **:se shell**.

Abbreviation: sh

shiftwidth The shiftwidth option sets the number of characters you tab forward or backward with the <CTRL–T> and <CTRL–D> tab commands. This does not affect the number of characters you tab forward with the <TAB> key.

Default: shiftwidth = 8
Abbreviation: sw, nosw

showmatch When you set the showmatch option, **vi** tries to find matching braces or square brackets. This feature is particularly useful for checking syntax when you are writing programs. This feature is separate from the syntax matching of the lisp option.

Default: noshowmatch
Abbreviations: sm, nosm

slowopen The slowopen option is designed for use on slow, dumb terminals. It limits the amount of output that can display on your terminal, so it speeds up the editing process.

Default: noslowopen
Abbreviations: slow, noslow

tabstop The tabstop option gives boundaries for the display of tabs on your terminal. To see the current value of the tabstop option enter **:se ts**.

Default: tabstop = 8
Abbreviation: ts

taglength When you identify a tagname (see *ctags(1)* in your *UTek Command Reference*) you choose a number of characters to uniquely identify it. The taglength option specifies that number. If you choose just enough characters to identify a tagname, when you invoke **vi** to edit the file containing *tagname*, it does not take long to find the tagname. The default value of 0 means that all characters are significant.

Default: taglength = 0
Abbreviation: tl

tags The tags option specifies the pathname of files used as tag files for the **tag** command.

Default: tags = /usr/lib/tags
Abbreviation: tags

term The term option sets the terminal type. Normally this comes from your environment, but you can reset the terminal type for a particular editing session. You must be in **ex** to set this temporary option. You can enter **:se term** to find you the current setting of the term option.

Default: terminal type from your environment
Abbreviation: term

terse The terse option gives shorter, more cryptic messages. This is useful if you are very familiar with **ex** and **vi**.

Default: noterse
Abbreviations: terse, noterse

timeout When you use the **:map** command to set macros, the macro you enter to call up the **vi** commands should be short. The timeout option ensures that they are short by giving you only one second to enter the macro. If you do not enter the macro within a second, it is not entered.

Default: timeout
Abbreviations: timeout, notimeout

ttytype The ttytype option is identical to the term option. Normally it is set from your environment, but you can reset it for a particular editing session. You must be in **ex** to set this temporary option. You can enter the command **:se ttytype** to find out your current ttytype.

Default: terminal type from your environment
Abbreviations: ttytype

warn The warn option tells you if you try to leave the editor without writing changes to the permanent file.

Default: warn
Abbreviations: warn, nowarn

window The window option can be set in several different ways. Normally, it comes from the TERMCAP entry for your terminal type. This entry knows how many lines constitute a full screen on your terminal type. You can also set the window option within a particular editing session by using the **:se window** command. For example, to have a window of 30 lines enter: **:se window=30**. If your terminal runs at 1200 baud or less, by default it does not display the full number of lines. If the terminal runs at 1200 baud, the default window size is 16 lines. If the terminal runs at 300 baud, the default window size is 8 lines. But you can change these default window settings according to baud rate. To do this you enter the **w***baudrate* command in the EXINIT variable. For example, to change the default setting for a terminal that is running at 1200 baud, on the EXINIT line of the .login, .cshrc, or .profile file enter:

w1200=29

Now when your terminal is running at 1200 baud, it displays a window of 29 lines on the screen.

Default: set by your baud rate or your TERMCAP entry.
Abbreviation: window

wrapmargin The wrapmargin option automatically inserts <RETURN> at the end of each line when you are in text insert mode, so that you do not have to press <RETURN> at each line. You give this option a number value that defines a new margin for wraparound of text. For example wrapmargin = 5 inserts <RETURN> when the text is within 5 characters of the right margin. This option does not break words in the middle; instead, the entire word is moved to the next line.

Default: wrapmargin = 0
Abbreviations: wm, nowm

writeany The writeany option lets you write your changes to any file where you have write permission, even if the permissions were changed during editing.

Default: nowriteany
Abbreviations: wa, nowa

Recovering From Errors

Many errors you make when using **vi** can be reversed. To make fewer errors, you can use some of the **vi** options described above, such as autowrite, readonly, report, and showmatch. But if you do make errors, there are several ways to recover from them. When you realize that you've done something wrong, stop and think about exactly what you did. This makes it much easier to recover from your errors.

If you made an error in the last command you entered, you can return the text to its previous state by entering **u**. Or if you want to undo changes on the current line only, enter **U**.

If **vi** is somehow interrupted while you are editing a file, you can recover some of the text using the **–r** option. The mail system sends you a message, saying that you have a copy of the interrupted document in a temporary buffer. To recover the temporary buffer, enter:

vi –r *filename*

If more than one instance of the file was saved, the editor gives you the newest instance each time you enter **vi –r**. So to recover an older copy, you can first recover the newer copies.

Entering simply **vi –r**, without arguments, lists all the recoverable files.

Temporarily Escaping the Editor

You can leave the editor temporarily to execute a UTek command, then return to your original position in the editor. To execute a UTek command enter:

!*command*

The shell executes the UTek command. When the command finishes, the editor prompts you to hit <RETURN> to continue. Look at the output of the command, then hit <RETURN> again; the screen is redrawn and you return to the editor.

To execute more than one UTek command, you can enter **:sh** after you see the output from the initial ! command. This invokes a new shell. When the new shell finishes executing the second command, type <CTRL–D> to return to the editor.

If you run C shell, you can stop the editor temporarily by entering:

:stop

This temporarily stops the editor and gives you the C shell prompt. To return to the editor, you can enter **fg**. If you have the autowrite option set, you can temporarily stop the editor without writing changes by entering:

:stop!

If you run C–Shell you can also temporarily stop the editor by entering:

:<CTRL–Z>

4A

Nroff/Troff Tutorial

Overview

In many ways the **troff** formatter resembles an assembly language, remarkably powerful and flexible. But some operations must be coded very specifically.

For producing general documents, macro packages define formatting rules and operations for specific styles of documents based on the **troff** formatter. In particular, the Memorandum Macros (MM) package provides most of the facilities needed for a wide range of document preparation. There are also packages for viewgraphs and other special applications. These packages are easier to use than the **troff** formatter language. They should be considered first. More information on the macro packages can be found in sections 4C, 4D, 4E, and 4F.

In the few cases where existing packages do not accomplish the job, small changes can be made to the packages that already exist. The part of the **troff** formatter described here is only a small part of the whole package; the more useful parts are introduced. Emphasis is on doing simple things and making incremental changes to what already exists.

To use the **troff** formatter, the text must include information that describes how the document is to be printed. Most commands to the **troff** formatter are placed on a line separate from the text itself, one command per line beginning with a period. For example:

some text
.ps 14
some more text

This input changes the point size of the letters being printed to 14 point (one point is 1/72 of an inch).

Occasionally, something special is needed in the middle of a line, such as an exponent. The backslash (\) is used to introduce **troff** commands and special characters within a line of text.

Tutorial Topics

Point Sizes and Line Spacing

The **.ps** request sets the *point size*. Since one point is 1/72 inch, 6–point characters are 1/12 inch high, and 36–point characters ar 1/2 inch high. Available point sizes with the **troff** formatter depend on the typesetter.

Point size is rounded to the closest valid value if the number following the **.ps** request is not a legal value. If no number follows the **.ps** request, point size reverts to the previous value. The **troff** processor begins with point size 10.

Point size can also be changed in the middle of a line or a word with a **\s** escape sequence. The **\s** sequence should be followed by a legal point size. The **\s0** sequence causes the size to revert to its previous value. The **\s1011** sequence is understood correctly as "point size 10, followed by point size 11".

Relative size changes are also legal and useful. The expression:

\s—2UNCLE\s+2

temporarily decreases the size by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. the amount of the relative change must be a single digit.

Another parameter that determines what the type looks like is spacing between lines. It is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is **.vs**. For running text, set the vertical spacing about 20 percent larger than the character size. For example, a typical combination is:

```
.ps 9
.vs 11p
```

Vertical spacing is partly a matter of taste (how text looks when squeezed into a given space) and partly a matter of traditional printing style. By default, the **troff** formatter uses a point size of 10 and a vertical spacing of 12. When **.vs** is used without arguments, vertical spacing reverts to the previous value.

The **.sp** request is used to get extra vertical space. Used alone, it gives one extra blank line (at whatever value **.vs** is set). If you wish to change this, **.sp** may be followed by a value. For instance:

INPUT	MEANING
`.sp 1.5i`	This gives a space of 1.5 inches. (**troff** usually understands decimal fractions.)
`.sp 2i`	This gives two inches of vertical space.
`.sp 2p`	This gives two points of vertical space.
`.sp 2 or .sp 2v`	This gives two vertical spaces (two of whatever **.vs** is set).

These same *scale factors* can be used after the **.vs**. Scale factors can be used after most commands that deal with physical dimensions.

All size numbers are converted internally to *machine units*, which, for the Wang C/A/T phototypesetter is 1/432 inch (1/6 point). For most purposes, this is enough resolution to provide good accuracy of representation. Vertical resolution is 1/144 inch (1/2 point). With the **troff** formatter, the resolution is typesetter dependent. The AUTOLOGIC, Inc. APS–5 typesetter has a resolution of 723 units per inch.

Fonts and Special Characters

The Wang C/A/T phototypesetter allows four different fonts at one time. Normally, three fonts (Times Roman, Times Italic, and Times Bold) and one collection of special characters are permanently mounted.

With the **troff** formatter, available fonts and names are dependent upon the typesetter. Refer to subsections three and four of this book for a more complete description of fonts and point sizes.

To change the font, the **.ft** request is used:

.ft B	switch to bold font.
.ft I	switch to italic font.
.ft R	switch to Roman font.
.ft P	return to previous font.
.ft	return to previous font.

The underline request (**.ul**) causes the next input line to print in italics. It can be followed by a number to indicate that more than one line is to be italicized.

Fonts can be changed within a line or word with the **\f** in–line sequences. For instance:

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

The **\fP** sequence can be used to reestablish the "standard" font in between font changes. For example:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Since *only the immediately previous font* is remembered, the original font must be restored after each change or it will be lost. The same is true of **.ps** and **.vs** when used without an argument.

There are other fonts available besides the standard set. The **.fp** request tells the **troff** formatter which fonts are actually mounted on the typesetter. For example, the following input says that the Helvetica font is mounted on position 3:

.fp 3 H

Appropriate **.fp** requests should appear at the beginning of a document if standard fonts are not used.

You can make a document relatively independent of the actual fonts used to print it by using font numbers instead of names. For example, both of the following mean "whatever font is mounted at position 3":

\f3

and

.ft3

Normal settings are Roman font on 1, italic on 2, and bold on 3.

You can also get synthetic bold fonts by overstriking letters with a slight offset using the **.bd** request.

Special characters have four-character input names beginning with \(and may be inserted anywhere in the text. In particular, Greek letters are all of the form \(*R, where R is an uppercase or lowercase Roman font letter reminiscent of the Greek.

Some characters are automatically translated into others; for instance, grave and acute accents become open and close single quotation marks. Similarly, a typed minus sign becomes a hyphen. The \- input will print an explicit minus sign. A **\e** entry causes a backslash to be printed.

Indents and Line Lengths

The **troff** processor starts with a line length of 6.5 inches, which is too wide for 8-1/2 inch by 11-inch paper. The **.ll** request resets the line length. For example:

.ll 6i

As with the **.sp** request, the actual length can be specified in several ways. The maximum line length provided by the C/A/T phototypesetter is 7.54 inches. Again, this may be different when using **troff** with another phototypesetter. To use the full width, the default physical left margin (page offset) must be reset. This is done with the **.po** request. The margin is normally a bit less than one inch from the left edge of the paper. The **.po 0** request sets the offest as far to the left as possible.

The indent request (**.in**) causes the left margin to be indented by some specified amount from the page offset. If the **.in** request is used to move the left margin to the right and the **.ll** request is used to move the right margin to the left, offset blocks of the text are obtained. The following example shows the input and the block of text that it creates:

.in 0.5i
.ll —0.5i
text to be set into a block
.ll +0.5i
.in —0.5i

TEXT CREATED:

```
A clergyman at Cambridge preached a sermon which one of his
auditors commended.  "Yes," said a gentleman to whom it was
mentioned, "it was a good sermon, but he stole it." This was told
to the preacher.  He resented it, and called the gentleman to
retract what he had said.  "I am not," replied the aggressor,
"very apt to retract my words, but in this instance I will.  I
said, you had stolen the sermon; I find I was wrong; for on
returning home and referring to the book whence I thought it was
taken, I found it there."
```

The use of + and — changes the previous setting by the specified amount rather than just overriding it. This distinction is quite important:

- The input **.ll +1i** makes lines one inch *longer*.
- The input **.ll 1i** makes lines one inch *long*.

With the **.in**, **.ll**, and **.po** requests, the previous value is used if no argument is specified.

The **.ti** request is used to indent a single line temporarily. The default unit for **.ti**, as for most horizontally–oriented requests (**.ll, .in, .po**), is *ems*. An em is roughly the width of the letter *m* in the current point size. Precisely, an em in size *p* is *p* points. The ems unit is used to make text that keeps its preportions regardless of point size. The ems can be specified as scaled factors directly, as in **.ti 2.5m**.

Lines can be indented negatively if the indent is already positive:

.ti —.3i

This input causes the next line to be moved back 3/10 of an inch.

A decorative initial capital that is three lines high is created by making the following changes:

- The whole paragraph is indented.
- The initial character is moved back with the **.ti** request.
- The initial character is made bigger (such as **\s36N\s0**) and moved down from its normal position.

Tabs

Tabs (the ASCII *horizontal tab* character) can be used to produce output in columns or to set the horizontal position of output. Typically, tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent but can be changed by the **.ta** request. For example, tab stops are set every inch with the following entry:

```
.ta 1i 2i 3i 4i 5i 6i
```

Tab stops are left justified (as on a typewriter). Lining up columns of right–justified numbers can therefore be a problem; if there are many numbers or if a table layout is needed, the table formatting program (**tbl**) is available.

A handful of numeric columns can be produced by preceding every number with enough blanks to make it line up when typed. For instance:

```
.nf
.ta 1i 2i 3i
\0\01Ts\0\02T\0\03
\040Ts\050T\060
700T800T900
.fi
```

The T symbol represents a tab. Each leading blank is a **\0** escape sequence. This character does not print but has the same width as a digit. When printed, the above input produces:

1	2	3
40	50	60
700	800	900

It is also possible to fill tabbed–over space with some character other than blanks by setting the tab replacement character with the **.tc** request (the *\(ru* string is the rule (_) character):

```
.ta 2i 3i
.tc \(ru \\"
NameTAgeT
```

This produces:

Name _____**Age**_____

To reset the tab replacement character to a blank, the **.tc** request (with no argument) is used. Lines can also be drawn with the **\l** escape sequence as described in the next subsection.

The **troff** processor provides a general mechanism called "fields" for setting up complicated columns. This is used by the **tbl** program.

Local Motions

The **troff** processor provides a number of escape sequences for placing characters of any size at any place. They can be used to draw special characters or to tune the output for a particular appearance. Most of these sequences are straightforward but messy to read and difficult to type correctly.

Vertical Motions

If the **eqn** program is not used, subscripts and superscripts are most easily done with the half–line local motions **\u** and **\d** sequences. To go back up the page half a point size, insert **\u** at the desired place; to go down half a point size, insert a **\d**. *The* **\u** *and* **\d** *should always be used in pairs.* Since **\u** and **\d** refer to the current point size, they should either be both inside or both outside the size changes. Otherwise, an unbalanced vertical motion will result.

Sometimes the space given by **\u** and **\d** is not the right amount. The **\v** sequence can be used to request an arbitrary amount of vertical motion. The in–line sequence **\v'***n***'** causes motion up or down the page by the amount specified in *n*. For example, to move the character *n* down, the following input would be used:

NOTE

Throughout the rest of this section, you will see text to be input displayed in two columns. Each line in the second column will begin with a comment indicator **\"**. *The commands you enter in column one will run without your comments from column two. However, these comments, when in the text, make the program easier to understand.*

```
.in +0.6i       \" indent paragraph
.ll —0.3i       \" shorten lines
.ti —0.3i       \" move n back
\v'2'\s36n\s0\v'—2'ott met Shott, Nott
shot at Shott...
```

A minus sign causes upward motion, while a plus sign or no sign causes downward motion. Thus **\v'—2'** causes an upward vertical motion of two line spaces.

There are other ways to specify the amount of motion:

```
\v'0.1i'
\v'3p'
\v'—0.5m'
```

The preceding inputs are all legal. The scale specifier **i, p,** *or* **m** goes inside the quotes. Any character can be used in place of the quotes. (This is true of all other **troff** formatter commands and sequences described in this section.)

Since the **troff** formatter does not take within-the-line vertical motions into account when figuring where it is on the page, output lines can have unexpected positions if the left and right ends are not at the same vertical position. Thus **v**, like **u** and **d**, should always be balanced with any upward vertical motion in a line compensated with the same amount in the downward direction.

Horizontal Motions

Arbitrary horizontal motions are also available. **h** is analogous to **v**, except that the default scale factor is *ems* instead of line spaces. As an example, the following input causes a backward motion of 1/10 of an inch:

```
\h'-0.1i'
```

We can see how this may be applied when working with mathematical text. When printing the mathematical symbol > >, the default spacing is too wide; **eqn** replaces this with the following:

```
\h'-0.3m'>
```

This produces >>.

Frequently, **h** is used with the *width function* (**w**) to generate motions equal to the width of some character string. The construction that follows is a number equal to the width of *thing* in machine units (1/432 inch):

```
\w'thing'
```

All **troff** formatter computations are ultimately done in these units. To move horizontally, the width of an **x** is used:

```
\h' \w'x'u'
```

Since the default scale factor for all horizontal dimensions is *m* (ems), **u** (machine units) must be used. If not, the motion produced will be too large. Nested quotes are acceptable to the **troff** formatter as long as none are omitted. An example of this kind of construction would be to print the string **.sp** by overstriking with a slight offset. The following example prints **.sp**, moves left by the width of **.sp**, moves right one unit, and prints **.sp** again:

```
.sp \h'-w' .sp'u'\h'1'.sp
```

There are several special purpose **troff** formatter sequences for local motion:

- The **0** is an unpaddable (never widened or split across a line by line justification and filling) white space the same width as a digit.
- The \<**space**> is an unpaddable character the width of a space.
- The \\| is 1/6 em wide.
- The \\^ is 1/12 em wide.

- The **\&** has zero width and is useful in entering a text line that would otherwise begin with a period (.).
- The **\o** sequence causes up to nine characters to be overstruck, and centered on the the widest of the characters. This is for accents such as:

  syst \o"\\(ga"me t\o"e\(aa"l\o"e\(aa"phonique

This produces the following:

systéme téléphonique

The accents **\(ga** and **\(aa** are just one character to the troff formatter.

Overstrikes

Overstrikes can be made with another special convention, **\z**, the zero-motion sequence. Normal horizontal motion is suppressed with the **\zx** (after printing the single character **x**), so another character can be laid on top of it. Although sizes can be changed within **\o**, characters are centered on the widest, and there can be no horizontal or vertical motions. The **\z** may be the only way to get what is needed.

A more ornate overstrike is given by the bracketing function **\b**, which piles up characters vertically, centered on the current baseline. Thus, big brackets are obtained by constructing them with piled-up smaller pieces.

Drawing Lines

A convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters is provided by the **troff** formatter. A one-inch long line is printed with a **\l'1i'** sequence. The length can be followed by the character to use if the the underline character (**_**) is not appropriate. The **\l'0.5i.'** sequence draws a 1/2 inch line of dots. Escape sequence **\L** is analogous, except that is draws a vertical instead of a horizontal line.

The **troff** formatter provides an even better facility for drawing lines using the **\D** escape sequence. This function can also be used to draw arcs, circles, and ellipses.

Strings

If your document contains a large number of occurrences of an acute accent over a letter **e**, typing **\o'e\''** for each é would be a nuisance. Fortunately, the **troff** formatter provides a way to store an arbitrary collection of text in a *string*, and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** formatter mechanisms whose judicious use permits typing a document with less effort and organizing it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the **.ds** request. The following line defines the string **e** to have the value \o"e\''':

```
.ds e  \o"e\'''
```

String names may be either one or two-characters long. They are referred to by *x for one-character names or by *(xy for two-character names (where *x* or *xy* are string names). Thus, to get

```
te'le'phone
```

given the definition of the string **e** as above,

```
t\*el\*ephone
```

is input.

If a string must begin with blanks, it is defined in the following way:

.ds *xx* " *text*

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may be several lines long. If the **troff** formatter encounters a backslash (\) at the end of any line, it is thrown away and the next line is added to the current one. A long string can be made by ending each line except the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings or even in terms of themselves.

Introduction to Macros

In its simplest form, a macro is a shorthand notation similar to a string. For instance, if every paragraph is to start in exactly the same way, with a space and a temporary indent of two ems, the following requests would perform the operation:

```
.sp
.ti +2m
```

To save typing these requests every time used, they could be collapsed into one shorthand line, such as a **troff** command **.PP**. The **.PP** is called a *macro*. The way to tell the **troff** formatter what **.PP** means is to define it with the **.de** request:

```
.de PP      \"paragraph macro
.sp
.ti +2m
..
```

The first line names the macro (**.PP** in this example). It is in uppercase so it will not conflict with any name that the **troff** formatter might already know about. The last line (. .) marks the end of the definition. In between is the text which is inserted whenever the **troff** formatter encounters the **.PP** macro call. A macro can contain any mixture of text and formatting requests.

The definition of a macro has to precede its first use; undefined macros are ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences for requests is important since it saves typing and makes later changes easier. If you decide that in producing a document the paragraph indent is too small, the vertical space is too large, and Roman font should be forced, only the definition of **.PP** needs to be changed to read:

```
.de PP    \"paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

The change takes affect everywhere **.PP** is used and is easier than changing commands throughout the whole document.

A **troff** formatter escape sequence that causes the rest of the line to be ignored, is \". It is used to add comments to the macro definition (a wise idea once definitions get complicated).

Another example of macros is this pair that start and end a block of offset, unfilled text:

```
.de OS    \'' start indented block
.sp
.nf
.in +0.5i
..
.de OE              \'' end indented block
.sp
.fi
.in -0.5i
..
```

The **.OS** and **.OE** macros could be used before and after text to provide the following effect:

```
    Copy to
John Doe
    Richard Roberts
    Janice Smith
```

In this example, the indentation used is **.in +0.5i** instead of **.in 0.5i**. This permits the nesting of the **.OS** and **.OE** macros to get blocks within blocks.

Should the amount of indentation be changed at a later date, it is necessary to change only the definitions of **.OS** and **.OE**, not individual requests throughout the whole document.

Titles, Pages, and Page Numbering

Titles, pages, and page numbering is not done automatically. This section includes many examples for you to copy.

Titles

To get a title at the top of each page, the following specifications must be provided in the **troff** formatter:

- What to do at and around the title line
- When to print the title
- What the actual title is

Pages

The **.NP** macro (new page) is defined to process titles at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

These requests are explained as follows:

- The **'bp** (begin page) request causes a skip to the top–of page.
- The **'sp 0.5i** request spaces down 1/2 inch.
- The **.tl** request prints the title.
- The **'sp 0.3i** request provides another 0.3 inch space.

The reason that the **'bp** and **'sp** requests are used instead of the **.bp** and **.sp** requests is that the **.sp** and **.bp** cause a break to take place. This means that all the input text collected but not yet printed is flushed out as soon as possible, and the next input line starts a new line of output. Had **.bp** been used in the **.NP** macro, a break in the middle of the current output line would occur when a new page is

started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. A single quote instead of a period before a request tells the **troff** formatter that no break is to take place. The output line currently being filled should not be forced out before the space or new page.

This is the list of requests that cause a break:

.bp	begin page
.br	break
.ce	center
.fi	fill mode
.nf	no-fill mode
.sp	space
.in	indent
.ti	temporary indent

Other requests cause no break, regardless of whether the period (.) or the single quotation mark (') is used. If a break is really needed, a **.br** request at the appropriate place provides it.

To ask for **.NP** at the bottom of each page, a statement like "when the text is within an inch of the bottom of the page, start the processing for a new page" is used. This is done with the **.wh** request. For example:

```
.wh -1i NP
```

No period character is used before **NP** since it is simply the name of a macro and not a macro call. The minus sign means "measure up from the bottom of the page"; so **-1i** means one inch from the bottom. The **.wh** request appears outside the definition of the **.NP** macro. For example:

```
.de NP
   - - - body of macro
..
.wh -1i NP
```

As text is actually being output, the **troff** formatter keeps track of its vertical position on the page; and after a line is printed within one inch from the bottom, the **.NP** macro is invoked.

The **.NP** macro causes a skip to the top of the next page (that is what the **'bp** was for) and prints the title with appropriate margins.

Beware of crossing a page boundary in an unexpected font or size when changing fonts or point sizes.

- Titles come out in the size and font most recently specified instead of what was intended.
- The length of a title is independent of the current line length, so titles come out at the default length of 6.5 inches unless changed. Changing title length is done with the **.lt** request.

There are several ways to fix the problems of point sizes and fonts in titles. The **.NP** macro can be changed to set the proper size and font for the title, and then restore the previous values, like this:

```
.de
'bp
'sp 0.5i
.fi R              \"set title font to Roman
.ps 10             \"set size to 10 point
.lt 6i             \"set length to 6 inches
.tl 'left top'center top'right top'
.ps                \"revert to previous size
.ft P              \"and to previous font
'sp 0.3i
..
```

This version of **.NP** does not work if the fields in the **.tl** request contain size or font changes. This can be corrected, as explained under *Environments* later in this section.

To get a footer at the bottom of a page, the **.NP** macro must be modified. One option is to have the **.NP** macro do some processing before the **'bp** request. Another option is to split the **.NP** macro into a footer macro (invoked at the bottom margin) and a header macro (invoked at the top of the page).

Page Numbers

Page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless explicitly requested. To get page numbers printed, the **%** character should be included in the **.tl** request at the position where the number is to appear. For example:

.tl ' '- % -' '

This centers the page number inside hyphens. The page number can be set at any time with either a **.bp** *n* request (which immediately starts a new page numbered *n*) or with **.pn** *n* (which sets the page number for the next page but does not cause a skip to the new page). The **.bp** +*n* sets the page number to *n* more than its current value. The **.bp** request, without an argument, means **.bp** +1.

Number Registers and Arithmetic

The **troff** processor does arithmetic. You can define and use variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. They also serve for any sort of arithmetic computation.

Number Registers

Like strings, number registers have one– or two–character names. They are set by the **.nr** request and are referenced anywhere by \n*x* or \n*xy* where *x* and *xy* are register names.

The **troff** formatter maintains several predefined number registers, including:

- % for the current page number,
- **nl** for the current vertical position on the page,
- **dy, mo,** and **yr** for the current day, month, and year, and
- **.s** and **.f** for the current size and font (the font is the number of a font position)

Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

An example of the use of number registers is in setting the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing, you may input:

```
.nr ps 9
.nr VS 11
```

The paragraph macro, **.PP**, is roughly defined as follows:

```
.de PP
.ps \\n(PS      \"reset size
.vs \\n(VSp     \"spacing
.ft R           \"font
.sp 0.5v        \"half a line
.ti +3m
..
```

This macro sets the font to Roman (regular) and the point size and line spacing to whatever values are stored in the number registers **PS** and **VS**.

Two backslashes (\\) in the definition are required to ensure that a backslash is left in the definition when the macro is used. If only one backslash is used, point size and vertical spacing are frozen at the time the macro is defined, not when it is used. Protection with an extra layer of backslashes is needed only for **\n**, *****, **\$**, and \ itself. Things like **\s**, **\f**, **\h**, **\v**, etc. do not need an extra backslash since they are immediately converted by the **troff** formatter to an internal code upon detection.

Arithmetic

Arithmetic expressions can appear anywhere that a number is expected. As an example:

```
.nr PS \\n(PS-2
```

decrements (reduces) register **PS** by two. Expressions can use the arithmetic operators +, −, *, \, % (mod), the relational operators >, >=, <, <=, =, != (not equal), and parentheses.

With Arithmetic:

- Number registers hold only integers. In the **troff** formatter, arithmetic uses truncating integer division, just like FORTRAN.
- In the absence of parentheses, evaluation is done left-to-right without any operator precedence, including relational operators. Thus:

  7*-4+3/13

  becomes **-1**.

Number registers can occur anywhere in an expression and so can scale indicators like **p, i, m,** etc. (but not spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (each unit is 1/432 inch) before any arithmetic is done, so **1i/2u** evaluates to **0.5i** correctly.

The scale indicator **u** often has to appear when least expected, in particular when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example, **.ll 7/2i** is not 3 ½ inches. Instead, it is 7 ems/2 inches. When translated into machine units, it becomes 0, because the default units for horizontal parameters (like **.ll**) are ems. Another incorrect try is **.ll 7i/2**. The 2 is 2 ems, so 7i/2 is small, although not 0. The correct way to specify 3 ½ inches is **.ll 7i/2u**. A safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a **.nr** request, there is no implication of horizontal or vertical dimension, so the default is units, and 7i/2 and 7i/2u mean the same thing. Thus:

```
.nr ll 7i/2
.ll \\n(llu
```

accomplishes what is desired as long as the **u** on the **.ll** request is included.

Macros with Arguments

Two things are needed to be able to define macros that can change from one use to the next according to parameters supplied as arguments:

1. When the macro is defined, indicate that some parts will be provided as arguments when the macro is called.
2. When the macro is called, you must provide the actual arguments to be plugged into the definition.

An example would be to define a macro (**.SM**) that prints its argument two points smaller than the surrounding text.

```
.de SM
\s-2\\$1\s+2
..
```

The macro call appears as:

```
.SM SMALL
```

The argument (SMALL in this example) then appears two points smaller than the rest of the print.

Within a macro definition, the symbol **$***n* refers to the *n*th argument with which the macro was called. Thus *$1* is the string to be placed in a smaller point size when **.SM** is called.

A slightly more complicated version is the following definition of **.SM**, which permits optional second and third arguments that are printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty. The macro call

```
.SM ABLE ),
```

would appear as **ABLE),** in output (note the smaller type).

The macro call

```
.SM BAKER ).
```

produces **(BAKER)**. with BAKER in smaller type:

(BAKER).

It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading. The number of arguments that a macro was called with is available in the **.$** number register.

Argument Rearrangement

The macro **.BD** is used to make "bold Roman" for **troff** formatter command names in text. It combines horizontal motions, width computations, and argument rearrangement:

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+2u'\\$1\fP\\$2
..
```

The **\h** and **\w** escape sequences need no extra backslash. The **\&** is there in case the argument begins with a period. Two backslashes (\\) are needed with the \\$*n* commands.

Numbered Section Headings

A **.SH** macro can be defined to produce automatically numbered section headings with the title in smaller sized, boldfaced print. The use is

.SH "*Section title ...*"

If the argument to a macro contains blanks, it must be surrounded by double quotes.

The definition of the **.SH** macro is

```
.nr SH 0                \"initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1         \"increment number
.ps \\n(PS-1            \"decrease PS number
\\n(SH. \\$1            \"title
.ps \\n(PS              \"restore PS
.sp 0.3i
.ft R
```

The section number is kept in number register **SH**, which is incremented (increased) each time just before use.

NOTE

A number register may have the same name as a macro without conflict but a string may not.

Note that **\\n(SH** and **\\n(PS** were used instead of a **\n(SH** and **\n(PS**. Had **\n(SH** been used, it would have yielded the value of the register at the time the macro was defined, not at the time it was used. Similarly, by using **\\n(PS**, the point size at the time the macro was called is obtained.

An example that does not involve numbers is the **.NP** macro (defined earlier) which had the request:

.tl 'left top'center top'right top'

The fields could be made into parameters by using:

.tl ' *(LT' *(CT' *(RT'

The title comes from three strings called *LT, CT,* and *RT.* (If these are empty, the title is a blank line). For example, *CT* could be set with:

.ds CT - % -

to give just the page number between hyphens. You can supply private definitions for any of the strings.

Conditionals

Suppose you want the **.SH** macro to leave two extra inches of space just before Section 1, but nowhere else. The best way to do that is to test inside the **.SH** macro whether the section number is 1, and add some space if it is. The **.if** command provides the conditional test that can be added just before the heading line is output:

.if \\n(SH=1 .sp 2i \"first section only

The condition after the **.if** request can be any arithmetic or logical expression. If the condition is logically true or arithmetically greater than zero, the rest of the line is treated as if it were text (a request in this case). If the condition is false, zero, or negative, the rest of the line is skipped.

It is possible to do more than one request if a condition is true. For example, if several operations are to be done prior to Section 1, the **.S1** macro is defined and invoked when Section 1 is almost complete (as determined by an **.if**).

```
.de S1
        - - -processing for section 1
..
.de SH
        - - -
.if \\n(SH=1 .S1
        - - -
..
```

An alternate way is to use the extended form of the **.if** request. For example:

.if \\n(SH=1 \{- - -*processing for section 1* - - -**\}**

The braces, \{ and \} must occur in the positions shown or unexpected extra lines are output. The **troff** processor also provides an if–else construction.

A condition can be negated by preceding it with an exclamation point (!). The same effect as above is obtained (but less clearly) by using:

.if !\\n(SH>1 .S1

There are a handful of other conditions that can be tested with **.if**. For example:

```
.if e .tl 'left top'center top'right top'    \"even page title
.if o .tl 'left top'center top'right top'    \"odd page title
```

gives facing pages different titles, depending on whether the page number is even or odd, when used inside an appropriate new page macro.

Two other conditions are **t** and **n**, which tell whether the formatter is **troff** or **nroff**:

.if t troff stuff . . .
.if n nroff stuff . . .

String comparisons may be made in a **.if** request:

.if *'string1'string2'* **stuff**

This executes the program **stuff** if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *, arguments with \$, and so forth.

Environments

There is a potential problem when going across a page boundary: parameters like the size and the font for a page title may be different from those in effect in the text when the page boundary occurs. A general way to deal with this and similar situations is provided by the **troff** formatter.

There are three environments. Each has independently selectable versions of many parameters associated with processing, including size, font, line and title lengths, fill/no–fill mode, tab stops, and partially collected lines. Thus the titling problem may be solved by processing the main text in one environment and titles in another with its own suitable parameters.

The **.ev** *n* request shifts to environment *n* (*n* must be 0, 1, or 2). The **.ev** request with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested.

If the main text is processed in environment 0 where the **troff** formatter begins by default, the new page macro **.NP** can then be modified to process titles in environment 1. For example:

```
.de NP
.ed 1        \"shift to new environment
.lt 6i       \"set parameters here
.ft R
.ps
    - - -any other processing
.ev          \"return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the **.NP** macro, but the version shown keeps all the processing in one place and is easier to understand and change.

Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example. Text of the footnote usually appears in the input before the place on the page is reached where it is to be printed. The place where the footnote is printed normally depends upon the size of the footnote. Therefore, there must be a way to determine the size of the footnote without printing it.

A mechanism called a *diversion* is provided by the **troff** formatter for doing this processing. Any part of the output may be diverted into a macro (diversion) instead of being printed; at some convenient time, the macro may be put back into the input.

The **.di** *xy* request begins a diversion. All subsequent output is collected into the macro *xy* until the **.di** request with no arguments is encountered. This terminates the diversion. Processed text is available at any time thereafter by giving the *.xy* request. The vertical size of the last finished diversion is contained in the built-in number register **dn**. For instance, to implement a keep operation so that text between the macros **.KS** and **.KE** is split across a page boundary (as for a figure or table), the following occurs:

- When a **.KS** is encountered, the output is diverted to determine its size.
- When a **.KE** is encountered and if the diverted text fits on the current page, it is printed there. If the diverted text does not fit on the current page, it is printed at the top of the next page.

The definitions of the **.KS** and **.KE** macros are as follows:

```
.de KS                          \"start keep
.br                             \"start fresh line
.ev 1                           \"collect in new environment
.fi                             \"make it filled text
.di XX                          \"collect in XX
..
.de KE                          \"end keep
.br                             \"get last partial line
.di                             \"end diversion
.if \\n(dn>=\\n(.t.bp           \"bp if does not fit
nf                              \"bring it back in no-fill
.XX                             \"text
.ev                             \"return to normal environment
..
```

The number register **nl** indicates the current position on the output page. Since output was being diverted, it remains at its value where the diversion started. The **dn** register contains the amount of text in the diversion. The distance to the next trap is in the built-in register **.t**. It is assumed that the next trap is at the bottom margin of the page. If the diversion is large enough to go past the trap, the **.if** is satisfied; a **.bp** request then is issued. In either case, the diverted output is brought back with *.XX*. It is essential to bring it back in no-fill mode so the **troff** formatter does no further processing on it.

This is only one possible keep operation. The most general keep macro would be more complex.

Tutorial Examples

Such common formatting needs as page margins and footnotes are deliberately not built into the **nroff** and **troff** formatters. You will probably have to prepare at least a small set of macro definitions to describe most documents. The macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

Examples in the following text are intended to be useful but not necessarily to cover all situations. Numerical parameters are used to make the examples easier to read and to show typical values. In many cases, number registers would be used to hold numerical information and to concentrate conditional parameter initialization data that depend on whether the **troff** or **nroff** formatter is being used.

Page Margins

Header and footer macros are defined to describe the top and bottom page margins areas, respectively. A trap is planted at page position 0 for the header and at *-n* (*-n* from the page bottom) for the footer. A simple header and footer macro definition follows:

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh -1i fo
```

This example provides blank one-inch top and bottom margins. The header occurs on the first page only if the definition and trap exist prior to the point at which the page breaks. In fill mode, the output line that activates the footer trap was typically ejected because some part or whole word did not fit on it. If anything in the footer and header causes a break, that word or part word is ejected. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character (’). When the header/footer design contains material to be handled independently, the environment may be switched to avoid confusion with running text.

A more complex example follows:

```
.de hd                          \"header
.if t .tl '\(rn' '\(rn'         \"troff cut mark
.if \\n%>1\{\
'sp |0.5i-1                     \"tl base at 0.5 inch
.tl ' '- %-' '                  \"centered page number
.ps                             \"restore size
.ft                             \"restore font
.vs \}                          \"restore vs
'sp |1.0i                       \"space to 1.0 inch
.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 0.5 inch up
.tl ' '- % -' '\}               \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

This example sets the size, font, and base-line spacing parameters for the footer material. Parameters are restored to their original values when the header is completed. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If the **troff** formatter is used, a cut mark is drawn in the form of root-en's at each margin and marks page breaks. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for the **sp** in the footer is that the footer is invoked by printing a line; this line's vertical spacing can move past the trap position by as much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** so that occurrences of **sp** at the top of the running text have no effect.

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 to save and to restore both the current and previous values, as shown for the size value in the following:

```
.de fo
.nr s1 \\n(.s       \"current size
.ps
.nr s2 \\n(.s       \"previous size
  - - -             \"rest of footer
..
.de hd
  - - -             \"header stuff
.ps \\n(s2          \"restore previous size
.ps \\n(s1          \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn              \"bottom number
.tl''-%-''          \"centered page number
..
.wh -0.5i-1v bn     \"tl base 0.5 inch up
```

Paragraphs and Headings

You should develop a paragraph macro that 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, requests a temporary indent, or does whatever else is necessary. For example:

```
.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            \"temporary indent
..
```

The first break in **pg** forces out any previous partial lines and must occur before the **.vs** request. The forcing of font, size, base–line spacing, and indent is to correct any prior errors and to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for the **troff** formatter; a larger space, at least as big as the output device vertical resolution, would be more suitable in the **nroff** formatter. The choice of remaining space to test for the **.ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to number section headings automatically might look like this:

```
.de sc                  \"section
  - - -                 \"force font, etc.
.sp 0.4                 \"prespace
.ne 2.4+\\n(.Vu         \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1               \"initial 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 the **.af** request.

Another common form is the labeled, indented paragraph where the label protrudes left into the indent space. For example:

```
.de lp              \"labeled paragraph
.pg
.in 0.5i            \"paragraph indent
.ta 0.2i 0.5i       \"label, paragraph
.ti 0
\t\\$1\t\c          \"flow into paragraph
```

The intended usage is:

.lp *label*

The label begins at 0.2 inch and cannot exceed a length of 0.3 inch without intruding into the paragraph. The *label* could be adjusted to the right against 0.4 inch by setting the tabs instead with the following entry:

.ta 0.4iR 0.5i

The last line of the **lp** macro ends with **\c** so that it becomes 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 begins a new column rather than produces the bottom margin. The header can initialize a column register that the footer increments and tests. The following example is arranged for two columns but is easily modified for more:

```
.de hd              \"header
  - - -
.nr cl 0 1          \"initial column count
.mk                 \"mark top of text
..
.de                 \"footer
.ie\\n+(cl<2\{\
.po +3.4i           \"next column; 3.1+0.3
.rt                 \"back to mark
.ns \}              \"no-space mode
.el \{\
.ps \\nMu           \"restore left margin
  - - -
'bp \}
..
.ll                 \"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** request, is made where the two-column output is to begin.

Footnote Processing

You embed 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

The following macro definitions cause footnotes to be 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 does not completely fit in the available space:

```
.de hd                  \"header
  - - -
.nr x 0 1               \"initial footnote count
.nr y 0-\\nb            \"current footer place
.ch fo -\\nbu           \"reset footer trap
.if \\n(dn .fz          \"leftover footnote
..
.de fo                  \"footer
.nr dn 0                \"zero last diversion size
.if \\nx \{\
.ev 1                   \"expand footnotes in environment 1
.nf                     \"retain vertical size
.FN                     \"footnotes
.rm FN                  \"delete it
.if '\\n(.z'fy' .di     \"end overflow diversion
.nr x 0                 \"disable fx
.ev  \}                 \"pop environment
  - - -
'bp
..
.de fx                  \"process footnote overflow
.if \\nx .di fy         \"divert overflow
..
.de fn                  \"start footnote
.da FN                  \"divert (append) footnote
.ev 1                   \"in environment 1
.if \\n+x=1 .fs         \"if first, include separator
.fi                     \"fill mode
..
.de ef                  \"end footnote
.br                     \"finish output
.nr z \\n(.v            \"save spacing
```

```
.ev                         \"pop environment
.di                         \"end diversion
.nr y -\\n(dn               \"new footer position
.if \\nx=1 .nr y -(\\n(.v-\\nz)\ \"uncertainty correction
.ch fo \\nyn                \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny)\
.ch fo \\n(nlu+1v           \"it did not fit
..
.de fs                      \"separator
\l'1i'                      \"one inch line
.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 0 hd                    \"header trap
.wh 12i fo                  \"footer trap, temp position
.wh -\\nbu fx               \"fx at footer position
.ch fo -\\nbu               \"conceal fx with fo
```

- The header macro (**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**.
- If the register **dn** indicates a leftover footnote, the **fz** macro is invoked to reprocess it.
- The footnote start macro (**fn**) begins a diversion in environment 1 and increments the footnote count register **x**. If the count is one, the footnote separator macro (**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 spacing size in register **z**.
- Register **y** is decremented by the size of the footnote that is available in register **dn**.
- On the first footnote, register **y** is further decremented by the difference in vertical base-line spacings of the two environments. This prevents late triggering of the footer trap from causing the last line of the combined footnotes to overflow.

- The footer trap is set to the lower 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 no-fill mode in environment 1 and deletes **FN**. If the footnotes were too large to fit, the macro **fx** is trap-invoked to redivert the overflow into **fy**, and the register **dn** later indicates to the header whether or not **fy** is empty.
- Both **fo** and **fx** macros 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 terminates the overflow diversion (if necessary) and zeros **x** to disable **fx**. This is because the uncertainty correction, together with a not-too-late triggering of the footer, can result in footnote macros finishing before reaching the **fx** trap.

Last Page

After the last input file has ended, **nroff** and **troff** formatters invoke the end macro, if any, and eject the remainder of the page.

```
.de en      \"end-macro
\c
'bp
..
.em en
```

During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro deposits a null partial word and creates another last page.

4B

Nroff/Troff Reference Guide

Introduction

The UTek **nroff** and **troff** text processors accept lines of text mixed with lines of format control information. They format the text into a printable, paginated document having a user–designed style. These formatters offer unusual freedom in document styling including:

- Arbitrary style headers and footers
- Arbitrary style footnotes
- Automatic sequence numbering for paragraphs and sections
- Multiple column output
- Dynamic font and point–size control
- Arbitrary horizontal and vertical local motions at any point
- Overstriking, bracket construction, and line drawing functions

The **nroff** text formatter formats text for typewriter–like terminals. This formatter can prepare output directly for a variety of terminal types and uses the full resolution of each terminal.

The **troff** (*device independent*) formatter formats text to be printed on a phototypesetter, but postprocessors must insert the codes that drive a particular phototypesetter. **Troff** drives virtually any phototypesetter since its output is an ASCII code describing the position, font, size, and so on, of characters to be typeset on a page. This output must be converted by another program, called a *postprocessor*, into codes that a particular phototypesetter understands. Parameters such as fonts, character sizes, and special characters depend on the phototypesetter being driven.

Full user control over fonts, sizes, and character positions, as well as the usual features of a formatter (right–margin justification, automatic hyphenation, page titling and numbering, and so on) are provided by the **troff** processor. It also provides macros, arithmetic variables and operations, and conditional testing for complicated formatting tasks.

Since the **nroff** and **troff** formatters are reasonably compatible, you can usually prepare input acceptable to either. Conditional input is provided that enables you to embed input expressly destined for either program.

Usage

The general form of invoking the **nroff** or **troff** formatter is:

nroff *options filenames*

or

troff *options filenames*

where *options* is any of a number of option arguments and *filenames* is the filenames of the documents to be formatted. An argument consisting of a single minus sign (-) is taken to be a filename corresponding to the standard input. Input is taken from the standard input if no filenames are given. Options can appear in any order so long as they appear before the files.

Nroff and Troff Options

OPTIONS	EFFECT
-o*list*	Prints 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* • *-n* means from the beginning to page *n* • *n-* means from page *n* to the end
-n*n*	Number the first generated page *n*.
-s*n*	Stop every *n* pages. The **nroff** formatter halts after every *n* pages (default *n* = 1) to allow paper loading or changing and resumes upon receipt of a newline. When using **troff**, you should use the **-s** option on the postprocessor, if one exists.
-m*name*	Add the macro file **/usr/lib/tmac/tmac.***name* to the beginning of the input files. Multiple **-m** macro package requests on a command line are accepted and are processed in sequence.

–c*name* Add the compacted macro files **/usr/lib/macros/cmp.[nt].[dt].***name* and **/usr/lib/macros/ucmp.[nt].***name* to the beginning of the input files. Multiple **–c** macro package requests on the command line are accepted. The compacted version of macro package *name* is used if it exists. If not, the **nroff** formatter tries the equivalent **–m***name* option instead. This option should be used instead of **–m** because it makes the **nroff** formatter execute significantly faster.

> **Note:** *This option only applies to the* **nroff** *formatter. Compacted macros are not supported with the* **troff** *formatter.*

–r*ar n* Set register *a* (one character) to *n*.

–i Read standard input after the input files are exhausted.

–q Invoke the simultaneous input/output mode of the **.rd** request.

–z Suppress formatted output. Only message output occurs (from **.tm** requests and diagnostics).

–k*name* Produce a compacted macro package from this invocation of the **nroff** formatter. This option has no effect if no **.co** request is used in the **nroff** formatter input. Otherwise, the compacted output is produced in files *d.name* and *t.name*.

> **Note:** *This option applies to* **nroff** *only. Compacted macros are not supported with the* **troff** *formatter.*

Options for Nroff Only

OPTIONS **EFFECTS**

–T*name* Specify the name of the output terminal type. Currently, defined names are:

Name	**Type**
37	(default) TELETYPE Model 37
tn300	GE TermiNet 300 (or any terminal without half–line capabilities)
300	DASI 300
300s	DASI 300s
450	DASI 450
X9700	Xerox 9700 laser printer
X	EBCDIC TX train printer

2631	Hewlett–Packard 2631 printer in regular mode
2631–c	Hewlett–Packard 2631 printer in compressed mode
2631–e	Hewlett–Packard 2631 printer in expanded mode
382	DCT–382 terminal
4000a	TRENDATA 4000a terminal
832	Anderson Jacobson 832 terminal
lp	(generic) printers that can underline and tab

–e Produce equally spaced words in adjusted lines using full terminal resolution.

–h Use output tabs during horizontal spacing to speed output and to reduce output byte count. Device tab settings are assumed to be every eight nominal character widths. The default settings of logical input tabs are also every eight nominal character widths.

–u*n* Set the emboldening factor (number of character overstrikes) in the **nroff** formatter for the third font position (bold) to be *n* (zero if *n* is missing).

Options for Troff Only

OPTIONS	EFFECT
–a	Send a printable approximation of the results in the ASCII character set to the standard output. This approximates a display of the document.
–T*name*	Specifies the intended output device (phototypesetter). The default output device is defined locally.
–F*dir*	Font information is accessed from the directory *dir/dev/name* where *name* is the default output device. The default font information directory is */usr/lib/font/dev/name*.

Each option is invoked as a separate argument. For example:

nroff –o4,8–10 –T300s –mabc *file1 file2*

This requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI 300s, and invokes the macro package **abc**.

Preprocessors and Postprocessors

Various preprocessors and postprocessors are available for use with the **nroff** and **troff** formatters:

- The equation preprocessor is **neqn** (for **nroff**).
- The table–construction preprocessor is **tbl**.
- A reverse–line postprocessor for multiple–column **nroff** formatter output on terminals without reverse–line ability is **col**. The TELETYPE Model 37 escape sequences that the **nroff** formatter produces by default are expected by **col**.
- The TELETYPE Model 37–simulator postprocessor for printing **nroff** formatter output on a Tektronix 4014 is **4014**.
- The phototypesetter–simulator postprocessor for the **troff** formatter that produces an approximation of phototypesetter output on a Tektronix 4014 is **tc**. For example, in:

  tbl *filenames* ¦ **eqn** ¦ **troff** [*options*] ¦ **tc**

  the first ¦ indicates the piping of **tbl** output to **eqn** input; the second ¦ indicates the piping of **eqn** output to the **troff** formatter input; and the third ¦ indicates the piping of the **troff** formatter output to the **tc** postprocessor.

Nroff/Troff Usage Guide

General Information

Form of Input

Under **nroff** and **troff**, control lines begin with a control character, normally a period (.) or an acute accent ('), followed by a one– or two–character name that specifies a basic request or the substitution of a user–defined macro in place of the control line. The acute accent control character suppresses the break function (the forced output of a partially filled line) caused by certain requests. Control characters can be separated from request/macro names by white space (spaces and/or tabs) for increased readability. Names must be followed by either a space or a newline character. Control lines with unrecognized request/macro names are ignored.

Formatter and Device Resolution

The **troff** text processor uses the resolution of the phototypesetter for which its output is being prepared (723 units/inch for the AUTOLOGIC APS–5 typesetter).

The **nroff** text processor internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter–like output devices. It rounds numerical input to the actual resolution of the output device indicated by the –T option (default is TELETYPE Model 37).

Numerical Parameter Input

Both **nroff** and **troff** formatters 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.

Table 4B–1
NROFF/TROFF SCALE INDICATORS

SCALE INDICATOR	MEANING	TROFF Basic Units	NROFF Basic Units
i	Inch	432	240
c	Centimeter	432x50/127	240x50/127
P	Pica = 1/6 inch	72	240/6
m	em = s points	6xs	c
n	en = em/2	3xs	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		

In the **nroff** processor, both em and en are taken to be equal to *c*, which is output–device dependent; common values are 1/10 and 1/12 inch. Actual character widths in the **nroff** formatter need not be all the same. Constructed characters (such as –>) are often extra wide. Default scaling is:

- **m** (em) for horizontally oriented requests (**.ll, .in, .ti, .ta, .lt, .po, .mc**) and functions (**\h, \l**)

- **v** for vertically oriented requests (**.pl, .wh, .ch, .dt, .sp, .sv, .ne, .rt**) and functions (**\v, \x, \L**)
- **p** for **.vs** request
- **u** for **.nr, .if**, and **.ie** requests

All other requests ignore scale indicators. When a number register containing an already scaled number is estimated to provide numerical input, the basic unit scale indicator (**u**) may need to be appended to prevent an additional inappropriate default scaling. The number *n* can be specified in decimal–fraction form; but the parameter finally stored is rounded to an integer number of basic units.

The absolute position indicator (¦) can be added before 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 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

spaces 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 can be used. The arithmetic operators are:

+, −, /, *, **%** (*mod*)

The logical operators are:

<, >, <=, >=, = (or ==) &(and), : (or)

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. 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
```

sets the line length to ½ the sum of 4.25 inches + 2 picas + 3 ems (30 points since the point size is 10.)

Notation

Numerical parameters are indicated in this section in two ways. A $\pm n$ means that the argument can 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, more requests expect to set parameters to nonnegative values; exceptions are **.sp, .wh, .ch, .nr**, and **.if**. The **.ps, .ft, .po, .vs, .ls, .ll, .in**, and **.lt** requests restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lowercase letters, and one– or two–character arguments are indicated by a pair of lowercase letters. Character string arguments are indicated by multicharacter mnemonics.

Font and Character Size Control

Fonts

Helvetica Regular, Helvetica Italic, and Helvetica Bold are among the many standard fonts available. Special fonts such as Mathematical fonts may also be available. The default fonts available with the **troff** formatter depend on the intended phototypesetter. These font styles are shown in the following figure.

Helvetica Regular

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + – . , / : ; = ? [ ] ¦
● □ — – _ ¼ ½ ¾ ° † ′ ¢ ® ©

Helvetica Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] ¦*
● □ — – _ ¼ ½ ¾ ° † ′ ¢ ® ©

Helvetic Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + – . , / : ; = ? [ ] ¦
● □ — – _ ¼ ½ ¾ ° † ′ ¢ ® ©

Special Mathematical Font

” ’ \ ^ _ § ~ / < > { } # @ + – = *
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu$
$\xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$
A B Γ Δ E Z Y Θ I K Λ M N
Ξ O Π R Σ T U Φ X Ψ Ω
$\sqrt{\ }$ ‾ ≤ ≥ ≡ ~ ≃ ≠ ← → ↑ ↓
× ÷ ± ∪ ∩ ■ ⊃ ⊆ ⊇ ∞ ∂ §
∇ ∫ ∈ ‡ ► ◄ | ◯ ⌠ ⌡ ⌉ ⌋
⌈ ⌊ ⌈ ⌊ ⌈ ⌉

Figure 4B–1. Example Font Styles.

The current font can be changed by use of the **.ft** request or by embedding at any desired point either \f*x*, \f(*xx*, or \f*n*, where *x* and *xx* are the name of a font and *n* is a numerical font position. With the **troff** formatter, the named font is loaded on font position 0 if the font exists and is not currently mounted by default or by a **.fp** request; but the font must still be in position 0 when the line is printed.

It is not necessary to change to the Special Font; characters on that font are automatically handled.

The **troff** text processor can be informed that any particular font is to be mounted by use of the **.fp** request. The list of know fonts is installation–dependent.

Font control is understood by the **nroff** formatter, which normally underlines italic characters. Subsection *Font Control Requests* contains a summary and explanation of font control requests.

In the subsequent discussion of font–related requests, *f* represents either a one– or two–character font name or the numerical font positions 1 through 4. The current font is available as numerical position in the read–only number register **.f**.

Character Set

The **troff** character set consists of the so–called Commercial II character set plus a Special Mathematical font character set, each having 102 characters. All ASCII characters are included with some of the Special Mathematical font. The ASCII characters are input as themselves (with three exceptions); the three ASCII character exceptions are mapped as shown in Table 4B–2. Non–ASCII characters are input in the form \(*xx*, where *xx* is a two–character name given in Tables 4B–3 and 4B–4.

Table 4B–2
TROFF ASCII CHARACTER MAPPING

ASCII INPUT	PRINTED BY TROFF
CHARACTER and NAME	CHARACTER and NAME
' acute accent	' close quote
` grave accent	` open quote
– minus sign	- hyphen

The characters ', `, and – can be input by \', \`, and \–, respectively, or by their special code name equivalents. The ASCII characters @ # " ' ` < > \ { } ~ ^ and _ exist on the Special Mathematical font and are printed as a one em space if that font is not mounted.

The **nroff** text preprocessor understands the entire **troff** character set but can only print:

- ASCII characters
- Additional characters as available on the output device
- Characters that can be constructed by overstriking or other combinations
- Characters that can reasonably be mapped into other printable characters

Exact behavior is determined by a device driver table prepared for each device. The characters ', ‘, and _ print as themselves.

Table 4B–3
STANDARD CONVENTION FOR NON–ASCII CHARACTERS

CHARACTER	INPUT NAME	CHARACTER NAME
‘	`	open quote
’	'	close quote
—	\(em	¾ em dash
-	-	hyphen
-	\(hy	hyphen
−	\(mi	current font minus
•	\(bu	bullet
□	\(sq	square
_	\(ru	rule
¼	\(14	one–fourth
½	\(12	one–half
¾	\(34	three–fourths
fi	\(fi	ligature
fl	\(fl	ligature
ff	\(ff	ligature
ffi	\(Fi	ligature
ffl	\(Fl	ligature
°	\(de	degree
†	\(dg	dagger
′	\(fm	foot mark
¢	\(ct	cent sign
®	\(rg	registered
©	\(co	copyright

Table 4B–4
NON–ASCII CHARACTERS IN SPECIAL FONT

CHARACTER	INPUT NAME	CHARACTER NAME
+	\(pl	plus sign
−	\(mi	minus sign
=	\(eq	equal sign
*	\(**	asterisk (multiply)
§	\(sc	section
'	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underline
/	\(sl	slash
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ϵ	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
τ	\(*t	tau
υ	\(*u	upsilon
ϕ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega

Table 4B–5
NON–ASCII CHARACTERS IN SPECIAL FONT

CHARACTER	INPUT NAME	CHARACTER NAME
A	A	Alpha
B	B	Beta
Γ	\(*G	Gamma
Δ	\(*D	Delta
E	E	Epsilon
Z	Z	Zeta
Y	Y	Eta
Θ	\(*H	Theta
I	I	Iota
K	K	Kappa
Λ	\(*L	Lambda
M	M	Mu
N	N	Nu
Ξ	\(*C	Xi
O	O	Omicron
Π	\(*P	Pi
R	R	Rho
Σ	\(*S	Sigma
T	T	Tau
U	U	Upsilon
Φ	\(*F	Phi
X	X	Chi
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
‾	\(rn	root extender
≥	\(>=	greater than or equal to
≤	\(<=	less than or equal to
≡	\(= =	identically equal
≃	\(~e	approximately equal
~	\(ap	approximates
≠	\(!=	not equal
←	\(->	right arrow
→	\(<-	left arrow

Table 4B–6
NON–ASCII CHARACTERS IN SPECIAL FONT

CHARACTER	INPUT NAME	CHARACTER NAME
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus–minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	equal subset
⊇	\(ip	equal superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
∫	\(is	integral sign
∅	\(es	empty set
∈	\(mp	member of
‡	\(dd	double dagger
►	\(rh	right hand
◄	\(lh	left hand
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top (big brace)
⎩	\(lb	left bottom (big brace)
⎫	\(rt	right top (big brace)
⎭	\(rb	right bottom (big brace)
⎨	\(lk	left center (big brace)
⎬	\(rk	right center (big brace)
⎪	\(bv	bold vertical
⎣	\(lf	left floor (big bracket)
⎦	\(rf	right floor (big bracket)
⎡	\(lc	left ceiling (big bracket)
⎤	\(rc	right ceiling (big bracket)

Character Size

Character sizes with the **troff** formatter depend on the phototypesetter installation. The **.ps** request is used to change or restore the point size. Alternatively, the point size can be changed between any two characters by embedding a **\s***n* at the desired point to set the size to *n*, or a **\s***n* ($1 \leq n \leq 9$) to increment/decrement the size by *n*; **\s0** restores the previous size.

With **nroff**, requested point size values that are between the two valid sizes yield the closer of the two. The current size is available in the **.s** number register. The **nroff** formatter ignores type size control. Subsection *Character Size Control Requests* contains a summary and explanation of character size requests.

Page Control

Top and bottom margins are not automatically provided. They can be defined by two macros that set traps at vertical positions 0 (top) and -*n* (*n* from the bottom). A pseudopage transition onto the first page occurs either when the first break occurs or when the first nondiverted text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. A summary and explanation of page control requests is given later in this section. References to the current diversion mean that the mechanism being described works during both ordinary and diverted output (the former is considered as the top diversion level).

Usable page width differs among phototypesetters. The left margin begins about 1/27 inch from the edge of the eight–inch wide, continuous roll page. Physical limitations on the **nroff** text processor output are output–device dependent.

Text Filling, Adjusting, and Centering

Filling and Adjusting

Normally, words are collected from input text lines and assembled into an output text line until some word does not fit. An attempt can be made to hyphenate the word in an effort to assemble a part of it into the output line. The spaces between the words on the output line are 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* backslash–space character (\ ). The adjusted word spacings are uniform in the **troff** formatter, and the minimum interword spacing can be controlled with the **.ss** request. In the **nroff** formatter, 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** number register, and text base–line position on the page is in the **nl** number register. The text *base-line high-water mark* (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, **:**, or **!** is taken to be the end of a sentence, and an additional space character is automatically provided during filling. Multiple interword 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** escape sequence can be embedded in 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 differentiated from a control line by prefacing it with the nonprinting, zero–width filler character **\&**. Another way is to specify output translation of some convenient character into the control character using the **.tr** request.

For example, if you specify the following on the input line, you can create a blank space wherever the tilde character (~) appears in text thereafter:

.tr~*<space>*

If you want a tilde to appear literally in the text, enter the following on the input line to reinstate the character:

.tr~~

Interrupted Text

Copying of an input line in no–fill mode can be interrupted by terminating the partial line with a **\c** escape sequence. The next encountered input text line is considered to be a continuation of the same line of input text. Similarly, a word within filled text can be interrupted by terminating the word (and line) with **\c**; the next encountered text is taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line is forced out along with any partial word. A summary and explanation of filling, adjusting, and centering requests is found later in this section.

Vertical Spacing

Base-line Spacing

Vertical spacing size (*v*) between base lines of successive output lines can be set using the **.vs** request. Spacing size must be large enough to accommodate character sizes on affected output lines. For the common type sizes (9 through 12 points), usual typesetting practice is to set *v* to two points greater than the point size; **troff** default is 10–point type on a 12–point spacing. The current *v* is available in the **.v** register. Multiple–*v* line separation (such as double spacing) can be obtained with a **.ls** request.

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 **ex**'*n*' can be embedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter, the delimiter choice is arbitrary except that it cannot look like the continuation of a number expression for *n*.

- If *n* is negative, the output line containing the word is preceded by *n* extra vertical spaces.
- If *n* is positive, the output line containing the word is followed by *n* extra vertical spaces.
- If successive requests for extra space apply to the same line, the maximum value is used.

Blocks of Vertical Space

A block of vertical space is ordinarily requested using **.sp**, which honors the no-space mode and does not space past a trap. A block of vertical space can be reserved using the **.sv** request.

A summary and explanation of vertical spacing requests is found later in this section.

Line Length and Indenting

The maximum line length for fill mode can be set with the **.ll** request. The indent can be set with a **.in** request; an indent applicable to only the next output line can be set with the **.ti** request. The line length includes indent space but not page-offset space. The line length minus the indent is the basis for centering with the **.ce** request. If a partially collected line exists, the effect of **.ll**, **.in** or **.ti** is delayed until after that line is output. In fill mode, the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i**, respectively. The length of three-part titles produced by **.tl** is independently set by **.lt**. A summary and explanation of line length and indenting requests is found later in this section.

Macro, Strings, Diversions, and Traps

Macros and Strings

A *macro* is a named set of arbitrary lines that can be invoked by name or with a trap. A *string* is a named string of characters, not including a newline character, that can be added by name at any point. Request, macro, and string names share the same name list. Macro and string names can be one- or two-characters long and can use previously-defined request, macro, or string names. Any of these entities can be renamed with **.rn** or removed with **.rm**.

- Macros are created by **.de** and **.di** and added to by **.am** and **.da** (**.di** and **.da** cause normal output to be stored in a macro).
- Strings are created by **.ds** and added to by **.as**.

A macros is invoked in the same way as a request; a control line beginning *.xx* interpolates the contents of macro *xx*. The remainder of the line can contain up to nine arguments. The strings *x* and *xx* are interpolated at any desired point with *x and *(*xx*, respectively. String references and macro invocations can 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:

- Contents of number registers indicated by *n* are interpolated
- Strings indicated by * are interpolated
- Arguments indicated by **\$** are interpolated
- Concealed newline characters indicated by \<*newline*> are eliminated
- Comments indicated by \" are eliminated
- **\t** and **\a** are interpreted as ASCII horizontal tab and start of heading (SOH), respectively
- \\ is interpreted as a single backslash \
- \. is interpreted as a single dot (.)

These interpretations can be suppressed by adding a backslash character \ before. For example, since \\ maps into a \, **n** copies as **\n** and is 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 can contain up to nine arguments. The argument separator is the space character, and arguments can be surrounded by quotation marks (" ") to permit embedded space characters. Pairs of quotation marks can be embedded in quoted arguments to represent a single quotation mark ("). If the desired arguments do not fit on a line, a concealed newline character can be used to continue on the next line.

When a macro is invoked, the input level is pushed down and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with **\$***n*, which interpolates the *n*th argument ($1 \leq n \leq 9$). If an invoked argument does not exist, a null string results. For example, the macro *xx* can be defined by:

```
de xx     \" begin definition
Today is \\$1 the \\ $2.
..        \" end definition
```

and called by:

```
.xx Monday 14th
```

to produce the text:

```
Today is Monday the 14th.
```

The \$ was concealed in the definition with a double backslash. The number of currently available arguments is in the **.$** register.

- No arguments are available at the top (nonmacro) level in this implementation.
- No arguments are available from within a string because string referencing is implemented as an input–level pushdown.
- 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 you should conceal string references (with an extra backslash \) to delay interpolation until argument reference time.

Diversions

Processed output can 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 can be set at a specified vertical position. The number registers **.dn** and **.dl**, respectively, contain the vertical and horizontal size of the most recent diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in no–fill 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 embed in the diversion the appropriate **.cs** or **.bd** request with the transparent mechanism described in subsection *Transparent Throughput*.

Diversions can be nested. Certain parameters and registers are associated with the current diversion level (the top nondiversion level can be thought of as diversion level 0). These parameters and registers are:

- Diversion trap and associated macro
- No–space mode
- Internally saved marked place (see **.mk** and **.rt**)
- Current vertical place (**.d** register)
- Current high–water text base line (**.h** register)
- Current diversion name (**.z** register)

Traps

Three types of trap mechanisms are available:

- Page trap
- Diversion trap
- Input–line–count trap

These macro–invocation traps can be planted using **.wh** requests at any page position including the top. This trap position can be changed using the **.ch** request. Trap positions at or below the bottom of the page have no effect until moved to within the page or rendered effective by an increase in page length. Two traps can 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 conceals the second until the first one is moved. If the first planted trap 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.

Macro–invocation traps, effective in the current diversion, can be planted using **.dt** requests. The **.t** register works in a diversion. If there is no subsequent trap, a large distance is returned

A summary and explanation of macros, strings, diversion, and position traps requests is found later in this section.

Number Registers

A variety of predefined number registers are available to you. In addition, you can define named registers. Register names are one– or two–characters long and must not conflict with request, macro, or string names. Except for certain predefined read–only number 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, and so forth. A number register can be used any time numerical input is expected or desired and can be used in numerical expressions.

Number registers are created and modified using the **.nr** request, which specifies name, numerical value, and automatic increment size. Registers are also modified if accessed with an automatic incrementing sequence. When the registers x and xx both contain n and have the automatic increment size m, the Table 4B–7 shows the values interpolated for the indicated access sequences.

Table 4B–7
NROFF/TROFF NUMBER REGISTER INTERPOLATION

SEQUENCE	EFFECT ON REGISTER	VALUE INTERPOLATED
nx	none	*n*
n(xx	none	*n*
nx	*x* incremented by *m*	*n + m*
n-x	*x* decremented by *m*	*n - m*
n+(xx	*xx* incremented by *m*	*n + m*
n-(xx	*xx* incremented by *m*	*n - m*

According to the format specified by the **.af** request, a number register is converted (when interpolated) to:

- Decimal (default)
- Decimal with leading zeros
- Lowercase Roman
- Uppercase Roman
- Lowercase sequential alphabetic
- Uppercase sequential alphabetic

A summary and explanation of number register requests is found later in this section.

Tabs, Leaders, and Fields

Tabs and Leaders

The ASCII horizontal tab character and the ASCII SOH character (the leader) 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 specified with a **.ta** request. 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–justified, right–justified, and centered. In Table 4B–8:

- *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line

- *d* is the distance from the current position on the input line (where a tab or leader was found) to the next tab stop
- *w* is the width of *next-string*

Table 4B–8
NROFF/TROFF TAB TYPES

TAB TYPE	LENGTH OF MOTION OR REPEATED CHARACTERS	LOCATION OF *next-string*
Left	*d*	Following *d*
Right	*d-w*	Right–justified within *d*
Centered	*d-w*/2	Centered on right end of *d*

The length of generated motion can 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 appended as motion. Tabs (or leaders) found after the last tab stop are ignored, but they may be used as *next-string* terminators.

Tabs and leaders are not interpreted in copy mode. The **\t** and **\a** always generate a noninterpreted 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. It consists of substrings 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 substrings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding can be negative. For example, if the field delimiter is # and the padding indicator is a caret ^, then:

```
#^xxx^right#
```

specifies a right–justified string with the string *xxx* centered in the remaining space.

A summary and explanation of tab, leader, and field requests is found later in this section.

Input/Output Conventions and Character Translations

Input Character Translations

The newline character delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted and can be used as delimiters or translated into a graphic with a **.tr** request. All others are ignored.

The backslash or "escape" character (\), introduces sequences that cause 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
- can be input with the sequence \\
- can be changed with **.ec**, and all that has been said about the default \ becomes true for the new escape character.

A **\e** sequence can be used to print the current backslash character. If necessary or convenient, the escape mechanism can be turned off with **.eo** and restored with **.ec**.

Ligatures

Five *ligatures* are available in the **troff** character set: , , , , and . They can be input (even in the **nroff** formatter) by **\(fi, \(fl, \(ff, \(Fi** and **\(Fl**, respectively. The ligature mode is normally on in the **troff** formatter and automatically invokes ligatures during input.

Backspacing, Underlining, and Overstriking

Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining is a form of line drawing and is a generalized overstriking function.

The **nroff** text processor underlines characters automatically in the underline font, specifiable with the **.uf** request. The underline font is normally on font position 2. In addition to the **.ft** request and the **\fF** escape sequence, the underline font can be selected by the **.ul** and **.cu** requests. Underlining is restricted to an output-device-dependent subset of reasonable characters.

Control Characters

Both the *break* control character (**.**) and the *no-break* control character (') can 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.

Output Translation

One character can be made a substitute for another character using the **.tr** request. All text processing (such as character comparisons) takes place with the input (substitute) character that appears to have the width of the final character. Graphic translation occurs at the moment of output (including diversion). Included later in this section is a summary and explanation of the output translation request.

Transparent Throughput

An input line beginning with a **\!** is read in copy mode and transparently output (without with initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism can be used to pass control information to a postprocessor or to embed control lines in a macro created by a diversion.

Comments and Concealed Newline Characters

A long input line that must stay on one line (for example, a string definition or no-filled text) can be split into many physical lines by ending all but the last one with the backslash \. The sequence *<newline>* is ignored except in a comment. Comments can be embedded at the end of any line by prefacing them with \". The newline character at the end of a comment cannot be concealed. A line beginning with \" appears as a blank line and behaves like **.sp 1**; a comment can be on a line by itself if you put a .\" at the beginning of the line.

Local Horizontal/Vertical Motion and Width Function

Local Motion

The function \v'*n*' and \h'*n*' can be used for local vertical and horizontal motion, respectively. The distance *n* may be negative; the positive directions are *rightward* and *downward*. A local motion is one contained within a line. To avoid unexpected vertical dislocations, the net vertical local motion (within a work in filled text and otherwise within a line) must balance to zero. Certain escape sequences providing local motion are summarized and explained in the Tables 4B–9 and 4B–10.

Table 4B–9
VERTICAL LOCAL MOTIONS

FUNCTION	EFFECT IN TROFF	EFFECT IN NROFF
\v'*n*'	Move distance *n*	
\u	1/2 em up	1/2 line up
\d	1/2 em down	1/2 line down
\r	1 em up	1 line up

Table 4B–10
HORIZONTAL LOCAL MOTIONS

FUNCTION	EFFECT IN TROFF	EFFECT IN NROFF
\h'*n*'	Move distance *n*	(same as in **troff**)
\<space>	Unpaddable space-sized space	(same as in **troff**)
\0	Digit-size space	(same as in **troff**)
\\|	1/6 em space	ignored
\^	1/12 em space	ignored

Width Function

The width function \w'*string*' generates the numerical width of *string* (in basic units). Size and font changes can be embedded in *string* and do not affect the current environment. For example:

```
.ti-\ w'1.'u
```

could be used to indent leftward, temporarily, a distance equal to the size of the **1.** string.

The width function also sets three number registers. The registers **st** and **sb** are set to the highest and lowest extent of *string* relative to the baseline respectively; then, for example, the total height of the string is \n(stu-\n(sbu. In the **troff** formatter, the number register **ct** is set to a value between 0 and 3:

- **0** means that all characters in *string* are short lowercase 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**).
- **3** means that both tall characters and characters with descenders are present.

Mark Horizontal Place

The escape sequence \k*x* causes current horizontal position in the input line to be stored in register *x*. As an example, the construction:

```
\kxword\h'|\nxu+2u'word
```

boldfaces *word* by backing up to almost its beginning and overprinting it, resulting in **word**.

Special Font Functions

Overstrike

Automatically centered overstriking of up to nine characters is provided by the overstrike function \o'*string*'. Characters in *string* are overprinted with centers aligned; the total width is that of the widest character. The *string* should not contain local vertical motion. For example, **\o'\(ci\(pl'** produces ⊕.

Zero-Width Characters

The function \z*c* outputs *c* without spacing over it and can be used to produce left–aligned overstruck combinations. As an example, **\(br\z\(rn\(ul\(br** produces the smallest possible constructed box □.

Large Brackets

The Special Mathematical font contains a number of bracket construction pieces that can be combined into various bracket styles. The function **\b**'*string*' can be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by one em and the total pile is centered one–half em above the current base line (one–half line in the **nroff** formatter).

Line Drawing

The **\l**'*nc*' function draws a string of repeated *c*'s toward the right for a distance *n* (**l** is lowercase **L**).

- If *c* looks like a continuation of an expression for *n*, it may be insulated from *n* with a backslash ampersand (**\&**).
- If *c* is not specified, the base–line 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 base–line rule _, underrule **\(ul**, and root en **\(ru**, the remainder space is covered by overlapping. 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'
..
```

such that

```
.us "underlined words"
```

yields words that are underlined.

The function **L**'*nc*' draws a vertical line consisting of the optional character *c* stacked vertically one em apart (one line in **nroff**), with the first two characters overlapped, if necessary, to form a continuous line. The default character is box rule **(br**; the other suitable character is bold vertical **(bv**. The line is begun without any initial motion relative to the current base line. A positive *n* specifies a line drawn downward, and a negative *n* specifies a line drawn upward. After the line is drawn, no compensating motions are made; the instantaneous base line is at the end of the line.

The horizontal and vertical line drawing functions can be used in combination to produce large boxes. The zero-width *box-rule* and the one-half em wide *underrule* were designed to form corners when using one em vertical spacings. For example, the macro:

```
.de eb
.sp -1    \" compensate for next automatic base-line spacing
.nf       \" avoid possibly overflowing word buffer
\h´ -.5n´\L´¦\\nau-1 ´\l\\n(.lu+1n\(ul´\L´-¦
\\nau+1´\l´¦0u-.5n\(ul´
.fi
..
```

draws a box around some text whose beginning vertical place was saved in number register *x* (for example, using **.mk x**).

Hyphenation

The automatic hyphenation can be switched off and on. When switched on with **.hy**, several variants can be set. A hyphenation indicator character can be embedded in a word to specify desired hyphenation points or can be prepended to suppress hyphenation. In addition, you may specify a small exception word list. The default condition of hyphenation is off.

Only words that consist of a central alphabetic string surrounded by nonalphabetic strings (usually null) are considered candidates for automatic hyphenation. Words that were input containing hyphens (**(mi**), em-dashes (**(em**), or hyphenation indicator characters (such as the hyphens in *mother-in-law*) are always subject to splitting after those characters whether or not automatic hyphenation is on or off. A summary and explanation of hyphenation requests is found later in this section.

Three–part Titles

The titling function **.tl** provides for automatic placement of three fields at the left, center, and right of a line with a title length specifiable with **.lt**. The **.tl** can be used anywhere and is independent of the normal text–collecting process. A common use is in header and footer macros. A summary and explanation of three–part title requests is found later in this section.

Output Line Numbering

Automatic sequence numbering of output lines can be requested with **.nm**. When in effect, a three–digit, Arabic number plus a digit space is added before output text lines. Text lines are offset by four digit spaces and otherwise retain their line length. A reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **.tl** are not numbered. Numbering can be temporarily suspended with **.nn** or with a **.nm** followed by a later **.mn +0**. In addition, a line number indent I and the number–text separation s can be specified in digit spaces. Further, if can be specified that only those line numbers that are multiples of some number m are to be printed (the others appear as blank number fields). A summary and explanation of output line numbering requests is found later in this section.

The following figure is an example of output line numbering. Paragraph portions are numbered with $m = 3$.

```
   Automatic sequence numbering of output lines can be
   requested with .nm.  When in effect, a three-digit,
3  Arabic number plus a digit-space is added before output
   text lines.  Text lines are offset by four digit-spaces
   and otherwise retain their line length.  A reduction in
6  line length may be desired to keep the right margin aligned
   with an earlier margin.  Blank lines, other vertical spaces,
   and lines generated by .tl are not numbered.  Numbering
9  can be temporarily suspended with .nn or with a .nm
   followed by a later .nm +0.  In addition, a line number
   indent I and the number-text separation s can be
12 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 appear as blank number fields).

15 As an example of output line numbering, paragraph portions
   of this figure are numbered with m=3:  .nm 1 3 was
   placed at the beginning;  .nm was placed at the end of the
18 first paragraph; and .nm +0 was placed in front of this
   paragraph;  and .nm placed at the end. Another example is
   .nm +5 5 x 3, which turns on numbering with the line number
21 of the next line to be five greater than the last numbered line,
   with m=5, spacing s untouched, and the indent I
   set to 3.
```

Figure 4B-2. Example of Output Line Numbering.

For this figure,

- **.nm 1 3** was placed at the beginning
- **.nm +0** was placed in front of the second and third paragraphs
- **.nm** was placed at the end

Another example is:

.nm +5 5 x 3

which turns on numbering with the line number of the next line to be five greater than the last numbered line, with $m = 5$, spacing s untouched, and the indent I set to 3.

Conditional Acceptance of Input

A summary and explanation of conditional acceptance requests where:

- *c* is a one–character, built–in condition name
- ! signifies *not*
- *n* is a numerical expression
- *string1* and *string2* are strings delimited by any nonblank, nonnumeric character not in the strings
- *anything* represents what is conditionally accepted

Built–in condition names are shown in the Table 4B–11:

Table 4B–11
NROFF/TROFF BUILT–IN CONDITION NAMES

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 condition *c* is true, if number *n* is greater than zero, or if strings compare identically (including motions and character size and font), *anything* is accepted as input. If an exclamation point (!) 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 multiline case, the first line must begin with a \{ (left delimiter), and the last line must end with a \} (right delimiter). If the left delimiter is the last thing on that line, the following newline should be concealed with a backslash \, or a blank line may be output.

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. The paired **.ie** and **.el** may be nested. For example:

```
.if e .tl 'Even page %'''
```

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
```

treats Page 1 differently from other pages.

Environment Switching

A number of parameters that control text processing are gathered together into an environment that can be switched by you. Environment parameters are those associated with some requests. In addition, partially collected lines and words are in the environment. Everything else is global; examples are page–oriented parameters, diversion–oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values. A summary and explanation of the environment switching request is found later in this section.

Insertions from Standard Input

The input can be switched temporarily to the system standard input with **.rd** and switched back when *two* newline characters in a row are found (the extra blank line is not used). This mechanism is intended for insertions in form–letter–like documentation. On the UTek system, the standard input can be the user keyboard, a pipe, or a filename.

If insertions are to be taken from the terminal keyboard while output is being printed on the terminal, the command line option **–q** turns 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 can be prepared by entering insertions for all copies in one file to be used as the standard input and causing the file containing the letter to reinvoke itself by using the **.nx** request. The process would be ended by a **.ex** request in the insertion file. A summary and explanation of insertions from the standard input requests is found later in this section.

Input/Output File Switching

Because of its simplicity, a description of input/output file switching is not included here. A summary and explanation of its requests is found later in this section.

Output and Error Messages

Output from **.tm**, **.pm**, and prompts from **.rd**, as well as various error messages are written onto the UTek standard output and error (message) output. By default, both are written onto your terminal, but they can be independently redirected.

Various error conditions can occur during the operation of the **nroff** and **troff** formatters. 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).
- *line overflow*— caused by an output line that grew too large to fit in the line buffer.

In both cases, a message is printed, and the offending excess is discarded. The affected word or line is marked at the point of truncation with an * (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 ends 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. A summary and explanation of output and error message requests is found later in this section.

Nroff Compacted Macros

The time required to read a macro package by the **nroff** formatter can be lessened by using a compacted macro (a preprocessed version of a macro package). The compacted version is equivalent to the noncompacted version, except that a compacted macro package cannot be read by the **.so** request. A compacted version of a macro package, called *name*, is used by the **-c***name* command line option, while the uncompacted version is used by the **-m***name* option. Because **-c***name* defaults to **-m***name* if the *name* macro package has not been compacted, you should always use **-c** rather than **-m**.

Building a Compacted Macro Package

Only macro, string, diversion definitions, number register definitions and values, environment settings, and trap settings can be compacted. End macro (**em**) request and any commands that may interact during package interpretation with command-line settings (such as references in the **MM** package to the number register **P**, which can be set from the command line) are not compactable.

There are two steps to make a compacted macro from a macro package:

1. Separate compactable from noncompactable parts.
2. Place noncompactable material at the end of the macro package with a **.co** request. The **.co** request indicates to the **nroff** formatter when to compact its current internal state.

Compactible Material

.
.
.

.co

Noncompactible Material

.
.
.

Produce Compacted Files

When compactable and noncompactable segments have been established, the **nroff** formatter may be run with the **-k** option to build the compacted files. For example, if the output file to be produced is called *mac*, the following may be used to build the compacted files:

nroff -kmac mac

This command causes the **nroff** formatter to create two files in the current directory, *d.mac* and *t.mac*.

NOTE

When **nroff** *is complied with the INCORE option (which is the default) only one file, d.mac, is created. In this case, only d.mac should be installed, ignoring the missing t.mac.*

The macro file must contain a **.co** request. Only lines before the **.co** request are compacted. Both **-k** and **.co** are necessary. If no **.co** is found in the file, the **-k** is ignored. Likewise, if no **-k** appears on the command line, the **.co** is ignored.

Each macro package must be compacted separately by the **nroff** formatter. Compacted macro packages depend on the particular version of the **nroff** formatter that produced them. Any compacted macro packages must be recompacted when a new version of an **nroff** formatter is installed. If a macro package was produced by a different version than the one attempting to read it, the **-c** is abandoned and the equivalent **-m** option is attempted instead.

Install Compacted Files

The two compacted files, *d.mac* and *t.mac*, must be installed into the system macro library, */usr/lib/macros*, with the proper names. If the files were produced by an **nroff** formatter, *cmp.n* must be added before their names. For example, if the macro package is called **mac**, the two **nroff** formatter compacted files may be installed by:

cp d.man /usr/lib/macros/cmp.n.d.mac

and

cp t.mac /usr/lib/macros/cmp.n.t.mac

Install Noncompactable Segment

The noncompactable segment from the original macro package must be installed on the system as:

/usr/lib/macros/ucmp.[nt].mac

where **n** of **[nt]** means the **nroff** formatter version, and the **t** means the **troff** formatter version. The noncompactable segment must be produced manually by using the editor. Using the **mac** package as an example, the following could be used to install the **nroff** formatter noncompactable segment:

```
$ ed mac
/^\.co$/+,$w /usr/lib/macros/ucmp.n.mac
```

Nroff/Troff Escape Sequences

Various special functions can be introduced anywhere in the input by means of an escape character (the backslash \). For example, the function \n*r* causes the interpolation of the contents of the number register *r* in place of the function. Number register *r* is either *x* for a single letter register name or *(xx* for a two-character register name. The later discussion, *Nroff/Troff Escape Sequences*, itemizes escape sequences for characters, indicators, and functions.

INPUT	TRANSLATES TO:
\\	\ (to prevent or delay the interpretation of \)
\'	Acute accent; equivalent to **\(aa**
\`	Grave accent; equivalent to **\(ga**
\-	Minus sign in the current font
\.	Period (dot)
\<*space*>	Unpaddable space-sized space character
\0	Unpaddable digit-width space
\\|	1/6 em marrow space character (zero width in the **nroff** formatter)
\^	1/12 em half-narrow space character (zero width in the **nroff** formatter)
\&	Nonprinting zero-width character
\!	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*
\{	Begin conditional input
\}	End condition input
\<*newline*>	Concealed (ignored) newline character
\a	Noninterpreted leader character
\b'*abc . . .*'	Bracket building function
\c	Continuation of interrupted text
\d	Forward (down) ½ em vertical motion (½ line in the **nroff** formatter)

\D'l dh dv' Draw a line from the current position by *dh, dv.*

\D'c d' Draw a circle of diameter *d* with left side at the current position

\D'e d1 d2' Draw an ellipse of diameters *d1* and *d2* with left side at current position

\D'a dh1 dv1 dh2 dv2' Draw a counterclockwise arc from current position to *dh1 + dh2, dv1 + dv2*, with center at *dh1, dv1* from current position

\D' dh1 dv1 dh2 dv2 . . . ' Draw a B–spline from current position by *dh1, dv1*, then by *dh2, dv2*, then . . .

\e Printable version of current escape character

\f*x*, \f(*xx*, \f*n* Change to font named *x* or *xx* or position *n*

\g*x*, \g(*xx* Return the **.af–type** format of the register *x* or *xx* (returns nothing if *x* or *xx* has not yet been referenced)

\h'*n*' Local horizontal motion; move right *n* (negative left)

\H'*n*' Character heights are set to *n* points without changing widths. A height of the form ±*n* is an increment to the current point size; a height of 0 restores the height to the current point size.

\k*x* Mark horizontal input place in register *x*

\l'*nc*' Horizontal line drawing function (optionally with *c*)

\L'*nc*' Vertical line drawing function (optionally with *c*)

\n*x*, \n(*xx* Interpolate number register *x* or *xx*

\o'*abc* . . . ' Overstrike characters *a, b, c* . . .

\p Break and spread output line

\r Reverse 1 em vertical motion (reverse line in the **nroff** formatter)

\s*n*, \s±*n* Point–size change function

\S'*n*' Output is slanted *n* degrees. The value of *n* may be negative. A value of 0 turns slant mode off.

\t Noninterpreted horizontal tab

\u Reverse (up) ½ em vertical motion (½ line in the **nroff** formatter)

\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 after)

\z*c* Print *c* with zero width (without spacing)

x any character not listed above

Predefined General Number Registers

% Current page number

ct Character type (set by width function)

dl Width (maximum) of last completed diversion

dn Height (vertical size) of last completed diversion

dw Current day of the week (1 through 7)

dy Current day of the month (1 through 31)

ln Output line number

mo Current month (1 through 12)

nl Vertical position of last–printed text base line

sb Depth of string below base line (generated by width function)

st Height of string above base line (generated by width function)

yr Last two digits of current year

c. Provides general register access to the input line number in the current input file. Contains the same value as the read–only **.c** register

Predefined Read–Only Number Registers

.$ Number of arguments available at the current macro level

$$ Process–id of the **troff** process (**troff** only)

.A Set to **1** in the **troff** formatter if **–a** option used; always **1** in the **nroff** formatter

.F Value is a *string* that is the name of the current input file

.H Available horizontal resolution in basic units

.L The current line spacing parameter (the value of the most recent **.ls** request)

.P The value **1** if the current page is being printed, and is 0 otherwise, that is, if the current page did not appear in the **-o** option list

.T Set to **1** in the **nroff** formatter if **-T** option used; always **0** in the **troff** formatter

.V Available vertical resolution in basic units

.a Post-line extra line space most recently used by **x**'*n*'

.b Boldfacing factor of the current font

.c Number of lines read from current input file

.d Current vertical place in current diversion; equal to **nl** if no diversion

.f Current font as physical quadrant (1 through 4)

.h Text base-line high-water mark on current page or diversion

.i Current indent

.j Indicates the current adjustment mode and type. Can be saved and later given to the **.ad** request to restore a previous mode.

.k The horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment

.l Current line length

.n Length of text portion on previous output line

.o Current page offset

.p Current page length

.r Number of number registers that remain available for use

.s Current point size

.t Distance to the next trap

.u Equal to **1** in fill mode and **0** in no-fill 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

Font Control Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.bd *F n*	off	n/a

Embolden font *F* by *n*–1 units. Characters in font *F* are boldfaced by printing each one twice, separated by *n*–1 basic units. A reasonable value for *n* is 3 when the character size is in the vicinity of 10 points. If *n* is missing, the embolden mode is turned off. The mode must still (or again) be in effect when the characters are physically printed. Theres is no effect in the **nroff** formatter.

.bd *S F n*	off	n/a

Embolden special font when current font is *F*. The characters in the special font are emboldened whenever the current font is *F*. The mode must still (or again) be in effect when the characters are physically printed. There is no effect in the **nroff** formatter.

.fp*n F*	R,I,B,S	ignored

Font position. A font named *F* is mounted on position *n*. It is a fatal error if *F* is not known.

.ft *F*	Roman	previous

Change to font *F* (*F* is *x, xx, digit*, or **P**). Font **P** means the previous font. For font changes within a line of text, sequences **f***x*, **f**(*xx*, or **f***n* can be used. Relevant parameters are a part of the current environment.

Character Size Control Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.cs *F n m*	off	n/a

Set constant character space (width) mode on for font *F* (if mounted). The width of every character is assumed to be *n*/36 ems. If *m* is absent, the em is that of the character point size; if *m* is given, the em is *m*–points. All affected characters are centered in this space including those with an actual width larger than this space. Special font characters occurring while the current font is *F* are also so treated. If *n* is absent, the mode is turned off. The mode must still (or again) be in effect when the characters are printed. There is no effect in the **nroff** formatter.

.ps ±*n*	10 point	previous

Set size to ±*n*. Any valid positive size value may be requested; if invalid, the next larger valid size is used (maximum of 36). Valid point sizes depend upon the typesetter used. A paired sequence *+n, -n* works because the previous requested value is remembered. For point size changes within a line of text, sequences **\s***n* or **\s**±*n* can be used. Relevant parameters are a part of the current environment. There is no effect in the **nroff** formatter.

.ss *n*	12/36 em	ignored

Set space–character size to *n*/36 ems. This size is the minimum word spacing in adjusted text. Relevant parameters are a part of the current environment. There is not effect in the **nroff** formatter.

Page Control Requests

Note: *Values separated by a semicolon(;) are for the* **nroff** *and* **troff** *formatters, respectively.*

The following heading apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.bp $\pm n$	$n=1$	n/a

Begin page. The current page is ejected and a new page is begun. If $\pm n$ is given, the new page number is $\pm n$. The scale indicator is ignored if not specified in the request. The request causes a break. The use of the apostrophe (') as the control character (instead of a dot (.)) suppresses the break function. The request with no n is inhibited by the **.ns** request.

.mk R	none	internal

Mark current, vertical place in an internal register (associated with the current diversion level) or in register R, if given. The request is used in conjunction with "return to marked vertical place in current diversion" request (**.rt**). Mode or relevant parameters are associated with current diversion level.

.ne n	none	$n=1v$

Need n vertical spaces. the scale indicator is ignored if not specified in the request.

- If the distance to the next trap position d is less than n, a forward vertical space of size d occurs, which springs the trap.
- If there are no remaining traps on the page, d is the distance to the bottom of the page.
- If d is less than vertical spacing 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. Mode or relevant parameters are associated with current diversion level.

.pl $\pm n$	11 in	11 in

Page length set to $\pm n$. The internal limitation is about 75 inches in the **troff** formatter and 136 inches in the **nroff** formatter. Current page length is available in the **.p** register. The scale indicator is ignored if not specified in the request.

.pn ±*n*	*n* = 1	ignored

Page number. the next page (when it occurs) has the page number ±*n*. The request must occur before the initial pseudopage transition to affect the page number of the first page. The current page number is in the **%** register.

.po ±*n*	0; 26/27in	previous

Page offset. The current left margin is set to ±*n*. The scale indicator is ignored if not specified in the request. The **troff** formatter initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. The current page offset is available in the **.o** register.

.rt ±*n*	none	internal

Return (upward only) to marked vertical place in current diversion. If ±*n* (with respect to place) is given, the vertical place is ±*n* from the top of the page or diversion. If *n* is absent, the vertical place is marked by a previous **.mk**. The **.sp** request can be used in all cases instead of **.rt** by spacing to the absolute place stored in an explicit register (for example, using the sequence **.mk** *r* . . . **.sp** |*r***u**). Mode or relevant parameters are associated with current diversion level. The scale indicator is ignored if not specified in the request.

Text Filling, Adjusting, and Centering Requests

Note: *Values separated by a semicolon (;) are for the* **nroff** *and* **troff** *formatters respectively.*

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ad *n*	adjust	adjust

Adjust. Output lines are adjusted with mode *n*. If the type indicator *n* is present, the adjustment type is as follows:

n	**ADJUSTMENT TYPE**
l	left margin only
r	right margin only
c	center
b or **n**	both margins
absent	unchanged

The adjustment type indicator *n* may also be a number obtained from the **.j** register. If fill mode is not on, adjustment is deferred. Relevant parameters are a part of the current environment.

.br	none	n/a

Break. 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.

.ce *n*	off	*n* = 1

Center. The next *n* input text lines are centered within the current line-length. 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 is left adjusted. The request normally causes a break. Relevant parameters are a part of the current environment.

.fi	fill	n/a

Fill mode. The request causes a break. Subsequent output lines are filled to provide an even right margin. Relevant parameters are a part of the current environment.

.na	adjust	n/a

No adjust. Output line adjusting is not done. Since adjustment is turned off, the right margin is ragged. Adjustment type for the **.ad** request is not changed. Output line filling still occurs if the fill mode is on. Relevant parameters are a part of the current environment.

.nf	fill	n/a

No–fill mode. Subsequent output lines are neither filled nor adjusted. The request normally causes a break. Input text lines are copied directly to output lines without regard for the current line length. Relevant parameters are a part of the current environment.

Vertical Spacing Requests

Note: *Values separated by a semicolon (;) are for the* **nroff** *and* **troff** *formatters respectively.*

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ls *n*	$n = 1$	previous

Line spacing set to $\pm n$. Output *n-1* blank lines (**Vs**) after each output text line. If the text or a previously appended blank line reached a trap position, appended blank lines are omitted. Relevant parameters are a part of the current environment.

.ns	space	n/a

Set no-space mode on. The no-space mode inhibits **.sp** and **.bp** requests without a next page number. It is turned off when a line of output occurs or with the **.rs** request. Mode or relevant parameters are associated with current diversion level.

.os	none	n/a

Output saved vertical space. The request is used to output a block of vertical space requested by an earlier **.sv** request. The no-space mode **.ns** has no effect.

.rs	none	n/a

Restore spacing. The no-space mode **.ns** is turned off. Mode or relevant parameters are associated with current diversion level.

.sp *n*	none	$n = 1v$

Space vertically. The request provides spaces 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 **.ns** is on, no spacing occurs. The scale indicator is ignored if not specified in the request. The request causes a break.

.sv *n*	none	*n* = *1v*

Save a contiguous vertical block of size *n*. If the distance to the next trap is greater than *n*, *n* vertical spaces are output. If the distance to the next trap is less than *n*, no vertical space is immediately output; but *n* is remembered for later output (**.os**). Subsequent **.sv** requests overwrite any still remembered *n*. The no-space mode **.ns** has no effect. The scale indicator is ignored if not specified in the request.

.vs *n*	1/6in;12pts	previous

Set vertical base-line spacing size *v*. Transient extra vertical spaces are available with **\x'***n*'. The scale indicator is ignored if not specified in the request. Relevant parameters are a part of the current environment.

Blank text line	none	n/a

This condition causes a break and output of a blank line (just as does **.sp 1**).

Line Length and Indenting Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.in ±*n*	*n*=0	previous

Indent. The indent is set to ±*n* and added before to each output line. The scale indicator is ignored if not specified in the request. Relevant parameters are a part of the current environment. The request causes a break.

.ll ±*n*	6.5in	previous

Line length. The line length is set to ±*n*. The scale indicator is ignored if not specified in the request. Relevant parameters are a part of the current environment.

.ti ±*n*	none	ignored

Temporary indent. The next output text line is indented a distance ±*n* with respect to the current indent. The resulting total indent can not be negative. The current indent is not changed. The scale indicator is ignored if not specified in the request. Relevant parameters are a part of the current environment. The request causes a break.

Macro, String, Diversion, and Trap Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.am *xx yy*	none	*.yy* = ..

Append to macro *xx* (append version of **.de**).

.as *xx string*	none	ignored

Append *string* to string *xx* (append version of **.ds**).

.ch *xx n*	none	n/a

Change trap location. Change the trap position for macro *xx* to be *n*. In the absence of *n*, the trap, if any, is removed. The scale indicator is ignored if not specified in the request.

.da *xx*	none	end

Divert and append to macro *xx* (append version of the **.di** request). Mode or relevant parameters are associated with current diversion level.

.de *xx yy*	none	*.yy* = ..

Define or redefine macro *xx*. The contents of the macro begin of the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with *.yy*. The macro *yy* is then called. In the absence of *yy*, the definition is terminated by a line beginning with dotdot (..). A macro can contain **.de** requests provided the terminating macros differ or the contained definition terminator is concealed; .. can be concealed as \\.. which copies as \.. and be reread as .. (dotdot).

.di *xx*	none	end

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. Mode or relevant parameters are associated with current diversion level.

.ds *xx string*	none	ignored

Define a string *xx* containing *string*. Any initial quotation mark (") in *string* is stripped to permit initial blanks.

.dt *n xx*	none	off

Install a diversion trap at position *n* in the current diversion to invoke macro *xx*. Another **.dt** redefines the diversion trap. If no arguments are given, the diversion trap is removed. Mode or relevant parameters are associated with current diversion level. The scale indicator is ignored if not specified in the request.

.em *xx*	none	none

End macro. Macro *xx* is 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.

.it *n xx*	none	off

Input–line–count trap. An input–line–count trap is set to invoke the macro *xx* after *n* lines of text input have been read (control or request lines do not count). Text can be in–line, or interpolated by in–line or trap–invoked macros. Relevant parameters are a part of the current environment.

.rm *xx*	none	ignored

Remove. A request, macro, or string is removed. The name *xx* is removed form the name list and any related storage space is freed. Subsequent references have no effect.

.rn *xx yy*	none	ignored

Rename. Rename request, macro, or string from *xx* to *yy*. If *yy* exists, it is first removed.

.wh *n xx*	none	n/a

When. A location trap is set to invoke macro *xx* at page position *n*; a negative *n* is 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* is removed. The scale indicator is ignored if not specified in the request.

Number Registers Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.af *r c*	Arabic	n/a

Assign format. Format *c* is assigned to register *r*. Available formats are:

c	**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, . . . z,aa,ab, . . . zz,aaa, . . .
A	0,A,B, . . . ,Z,AA,AB, . . . ,ZZ,AAA, . . .

An Arabic format having *n* digits specifies a field width of *n* digits. Read–only registers and width function are always Arabic.

.nr *r* ±*n m*	none	n/a

Number register. The number register *r* is assigned the value ±*n* with respect to the previous value, if any. The automatic incrementing value is set to *m*. The number register value *n* is ignored if not specified in the request.

.rr *r*	none	n/a

Remove register. The number register *r* is removed. If many registers are being created dynamically, it may be necessary to remove registers that are no longer used in order to recapture internal storage space for newer registers.

Tab, Leader, and Field Requests

Note: *Values separated by a semicolon (;) are for the* **nroff** *and* **troff** *formatters respectively.*

The following headings apply throughout this subsection.

REQUEST FOR	INITIAL VALUE	IF NO ARGUEMENT
.fc *a b*	off	off

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.

.lc *c*	none	n/a

Leader repetition character becomes *c* or is removed, specifying motion. Relevant parameters are a part of the current environment.

.ta *nt...*	8n;0.5in	n/a

Set tab stops and types. The adjustment within the tab is as follows:

t	**ADJUSTMENT TYPE**
R	right
C	centering
absent	left

Tab stops for the **troff** formatter are preset every 0.5 inch; tab stops for the **nroff** formatter are preset every 8 nominal character widths. Stop values are separated by spaces, and a value preceded by a plus (+) is treated as an increment to the previous stop value. Relevant parameters are a part of the current environment. The scale indicator is ignored if not specified in the request.

.tc *c*	none	n/a

Tab repetition character becomes *e* or is removed specifying motion. Relevant parameters are a part of the current environment.

Input/Output and Translation Requests

Note: *Values separated by a semicolon (;) are for the* **nroff** *and* **troff** *formatters respectively.*

The following headings apply throughout this subsection.

3REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.cc *c*	.(dot)	n/a

Set control character to *c* or reset to dot (.). Relevant parameters are a part of the current environment.

.cu *n*	off	$n = 1$

Continuous underline in the **nroff** formatter. A variant of **.ul** that causes every character to be underlined. Identical to **.ul** in the **troff** formatter. Relevant parameters are a part of the current environment.

.c2 *c*	'	'

Set no-break control character to *c* or reset to apostrophe ('). Relevant parameters are a part of the current environment.

.ec *c*	\	\

Set escape character to \ or to *c*, if given.

.eo	on	n/a

Turn escape character mechanism off.

.lg *n*	off;on	on

Ligature mode is turned on if *n* is absent or nonzero and turned off if $n = 0$. If $n = 2$, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macros, string, register, filenames, and copy mode. There is no effect in the **nroff** formatter.

.tr *abcd...*	none	n/a

Translate *a* to *b*, *c* into *d*, and so forth. If an odd number of characters is given, the last one is mapped into the space character. To be consistent, a particular translation must stay in effect from input to output time. Initially there are no translation values.

.uf *f*	Italic	Italic

Underline font set to *f* (to be switched to by **.ul**). In the **nroff** formatter *f* can not be on position 1 (initially Times Roman).

.ul *n*	off	*n* = 1

Underline in the **nroff** formatter (italicize in **troff**) the next *n* input text lines. Switch to underline fort saving the current font for later restoration; other font changes within the span of a **.ul** take effect, but the restoration undoes the last change. Output generated by **.tl** is affected by the font change but does not decrement *n*. If *n* is greater than 1, there is the risk that a trap–interpolated macro may provide text lines within the span, which environment switching can prevent. Relevant parameters are a part of the current environment.

Hyphenation Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.hc *c*	\%	\%

Hyphenation character. Hyphenation indicator character is set to *c* or to the default **\%**. The indicator does not appear in the output. Relevant parameters are a part of the current environment.

.hw *word1...*	none	ignored

Exception words. Hyphenation points in words are specified with embedded minus signs. Versions of a word with terminal **s** are implied; that is, *dig-it* implies *dig-its*. This list is examined initially and after each suffix stripping. Space available is small— about 128 characters.

.hy *n*	off, $n = 0$	on, $n = 1$

Hyphenate. Automatic hyphenation is turned on for $n \geq 1$ or off for $n = 0$. If $n = 2$, last lines (ones that cause a trap) are not hyphenated. For $n = 4$ the last two characters of a word are not divided. For $n = 8$ the first two characters of a word are not divided. These values are additive; that is, $n = 14$ invokes all three restrictions. Relevant parameters are a part of the current environment.

.nh	no hyphen	n/a

No hyphenation. Automatic hyphenation is turned off. Relevant parameters are a part of the current environment.

Three–Part Title Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.lt $\pm n$	6.5in	previous

Length of title set to $\pm n$. LIne length and title length are independent. Indents do not apply to titles; page offsets do. Relevant parameters are a part of the current environment. The scale indicator is ignored if not specified in the request.

.pc *c*	%	off

Page number character set to *c* or removed. The page number register remains **%**.

.tl *'l'c'r'*	none	n/a

Three–part title. The strings *l*, *c*, and *r* are respectively left–adjusted, centered, and right–adjusted in the current title length. Any of the strings may be empty, and overlapping is permitted. If the page number character (initially **%**) is found within any of the fields, it is replaced by the current page number having the format assigned to register **%**. Any character can be used as the string delimiter.

Output Line Numbering Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.nm ±*n m s i*	none	off

Line number mode. If ±*n* is given, line numbering is turned on, and the next output line is numbered \(+*n*. Default values are *m* = 1, *s* = 1, and *i* = 0. Parameters corresponding to missing arguments are unaffected; a nonnumeric argument is considered missing. In the absence of all arguments, numbering is turned off, and the next line number is preserved for possible further use in number register **ln**. Relevant parameters are a part of the current environment.

.nn *n*	none	*n* = 1

Next *n* lines are not numbered. Relevant parameters are a part of the current environment.

Conditional Acceptance Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.el *anything*	none	n/a

The "else" portion of an *if-else* statement.

.ie *c anything*	none	n/a

The "if" position of an *if-else* statement. The *c* can be any of the forms acceptable with the **.if** request.

If condition *c* is true, accept *anything* as input; for multiline case, use \{*anything*\}. The scale indicator is ignored if not specified in the request.

.if !*c anything*	none	n/a

If condition *c* is false, accept *anything*.

.if *n anything*	none	n/a

If expression *n*>0, accept *anything*. The scale indicator is ignored if not specified in the request.

.if !*n anything*	none	n/a

If expression $n \leq 0$, accept *anything*. The scale indicator is ignored if not specified in the request.

.if *'s1 s2' anything*	none	n/a

If string *s1* is identical to string *s2*, accept *anything*.

.if !*'s1 s2' anything*	none	n/a

If string *s1* is not identical to string *s2*, accept *anything*.

.if *c anything*	none	n/a

Envir
Requ

The following h

REQUEST FORM
.ev *n*

Environment sv
restoring a pre
reference.

Insert
Requ

The following h

REQUEST FORM
.ex

Exit from the n
input had ende

.rd *prompt*

Read insertion
found. If stand
the user termin
after *prompt*.

Miscellaneous Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE
.co	none

Specify the point in the macro file at which compaction ends. W
called on the command line, all lines in the file *name* before the .
compacted.

.fl	none

Flush output buffer. Used in interactive debugging to force outp
causes a break.

.ig *yy*	none

Ignore input lines until call of *yy*. This request behaves like the .
that the input is discarded. The input is read in copy mode, and
incremented registers are affected.

.mc *c n*	none

Sets margin character *c* and separation *n*. Specifies that a marg
appear a distance *n* to the right of the right margin after each no
(except those produced by **.tl**). If the output line is too long (as c
mode), the character is appended to the line. If *n* is not given, th
used; the initial *n* is 0.2 inches in the **nroff** formatter and 1 em ir
parameters are a part of the current environment. The scale ind
not specified in the request.

.pm *t*	none

Print macros. The names and sizes of all defined macros and st
the user terminal. If *t* is given, only the total of the sizes is print
in blocks of 128 characters.

.sy *cmd args*	none

The UTek command *cmd* is executed. Its output is not captured.
input for *cmd* is closed.

.tm *string*	none	newline

Print *string* on terminal (UTek operating system standard message output). After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user terminal.

Output and Error Messages Request

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ab *text*	none	n/a

Prints *text* on the message output and terminates without further processing. If *text* is missing, *User Abort* is printed. This request does not cause a break. The output buffer is flushed.

If condition *c* is true, accept *anything* as input; for multiline case, use \{*anything*\}. The scale indicator is ignored if not specified in the request.

.if !*c anything*	none	n/a

If condition *c* is false, accept *anything*.

.if *n anything*	none	n/a

If expression *n*>0, accept *anything*. The scale indicator is ignored if not specified in the request.

.if !*n anything*	none	n/a

If expression $n \leq 0$, accept *anything*. The scale indicator is ignored if not specified in the request.

.if *'s1 s2' anything*	none	n/a

If string *s1* is identical to string *s2*, accept *anything*.

.if !*'s1 s2' anything*	none	n/a

If string *s1* is not identical to string *s2*, accept *anything*.

.if *c anything*	none	n/a

Environment Switching Request

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ev *n*	*n* = 0	previous

Environment switched to 0, 1, or 2. Switching is done in pushdown fashion so that restoring a previous environment must be done with **.ev** rather than specific reference.

Insertions from Standard Input Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ex	none	n/a

Exit from the **nroff/troff** formatter. Text processing is terminated exactly as if all input had ended.

.rd *prompt*	*prompt* = BEL	n/a

Read insertion from the standard input until two newline characters in a row are found. If standard input is the user keyboard, a *prompt* (or a BEL) is written onto the user terminal. The request behaves like a macro; arguments can be placed after *prompt*.

Input/Output File Switching Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENTP
.cf *filename*	none	n/a

Copy file. This request copies the contents of *filename* into the **troff** output file at this point, uninterpreted. Problems occur unless the motions in the file restore current horizontal and vertical position.

.nx *filename*	end-of-file	n/a

Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.

.pi *program*	none	n/a

Pipe output to *program*. This request must occur before any printing occurs. No arguments are transmitted to *program*.

.so *filename*	none	n/a

Switch source file (pushdown). The top input level (file reading) is switched to *filename*. Contents are interpolated at the point the request is encountered. When the new file ends, input is again taken from the original file. The **.so** request may be nested.

Miscellaneous Requests

The following headings apply throughout this subsection.

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.co	none	n/a

Specify the point in the macro file at which compaction ends. When **-k***name* is called on the command line, all lines in the file *name* before the **.co** request are compacted.

.fl	none	n/a

Flush output buffer. Used in interactive debugging to force output. The request causes a break.

.ig *yy*	none	*.yy*=..

Ignore input lines until call of *yy*. This request behaves like the **.de** request except that the input is discarded. The input is read in copy mode, and any automatically incremented registers are affected.

.mc *c n*	none	off

Sets margin character *c* and separation *n*. Specifies that a margin character *c* appear a distance *n* to the right of the right margin after each nonempty text line (except those produced by **.tl**). If the output line is too long (as can happen in no–fill mode), the character is appended to the line. If *n* is not given, the previous *n* is used; the initial *n* is 0.2 inches in the **nroff** formatter and 1 em in **troff**. Relevant parameters are a part of the current environment. The scale indicator is ignored if not specified in the request.

.pm *t*	none	all

Print macros. The names and sizes of all defined macros and strings are printed on the user terminal. If *t* is given, only the total of the sizes is printed. Sizes are given in blocks of 128 characters.

.sy *cmd args*	none	n/a

The UTek command *cmd* is executed. Its output is not captured. The standard input for *cmd* is closed.

.tm *string*	none	newline

Print *string* on terminal (UTek operating system standard message output). After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user terminal.

Output and Error Messages Request

REQUEST FORM	INITIAL VALUE	IF NO ARGUMENT
.ab *text*	none	n/a

Prints *text* on the message output and terminates without further processing. If *text* is missing, *User Abort* is printed. This request does not cause a break. The output buffer is flushed.

The MS Text-Formatting Macros

Invoking Ms

The **ms** text formatter is a macro package for the **nroff** and **troff** text processing programs. Invoke it as follows:

nroff -ms *options files*

The *options* argument includes preprocessors, as well as the frequently used **-T** option. It specifies the type of device where you direct the output. For example, **-Tlpr** sends the output to the Printronix line printer. The topic *The -T Option* lists available terminal types. If you do not enter the **-T** option, a standard ASCII terminal is assumed.

Basic Text Formatting

To use **ms**, prepare a file that contains the text to be formatted and the **ms** commands that control its appearance. Commands start with a period at the beginning of the line and consist of one or two uppercase letters, possibly followed by arguments. You can inbed certain commands directly in the text — in this case they start with a backslash (\) character. To print a literal backslash in a document use the \e command. For example, A\eB prints as A\B.

Line breaks are made only on word boundaries or at standard hyphenation points within words. So if you include spaces in the text that shouldn't be padded or broken across a line, enter an unpaddable space as a backslash, followed by a space. For example, New\ York displays on one line with no extra spaces. Hyphenation is normally done automatically. Do not hyphenate a word you enter unless it is a compound word. If you prefer unhyphenated text, the **.HY** command turns off hyphenation. Similarly, **.AD** specifies whether text is adjusted to the right margin, or printed with a ragged right margin. Commands are also available for beginning a new line before the margin is reached (**.BR**), inserting blank lines (**.SP**), requesting double-spacing (**.LS**), moving to a new page (**.BP**) or column (**.BC**), conditionally moving to a new page if not enough space remains on the current page (**.NE**), and changing the page number (**.PN**).

Overall Format

Manuscript files can start with a simple paragraph (**.PP** or **.LP**), a section heading (**.SH** or **.NH**), or a title line (**.TL**). They can also start with a command that describes the general format of the document. The **.RP** macro signifies a released paper with title page. **.TR** or **.TM** signifies an internal technical report or memo. The command **.IOC** signifies a memo (Inter–office Communication), and **.LT** signifies a letter. You can find examples of these formats at the end of this section.

Indentation

Several mechanisms generate indented text in **ms**. An individual paragraph can be indented using the **.IP** command, with an optional label at the left of the paragraph. The margin is reset at the next paragraph command unless the command **.RS** is given. **.RS** fixes the current level of indentation until a corresponding **.RE** is encountered. The command **.IS** combines **.IP** and **.RS**. Use it to start an indented section that contains multiple paragraphs; all text up to the next **.IE** is indented. You can nest **.IS/.IE** pairs to have several levels of indentation.

The commands **.QP**, **.QS**, and **.QE** are similar to **.IP**, **.IS**, and **.IE**, but indentation is made from both the left and right margins, for example in long quotes. If you use numbered headings, you can indent entire sections by placing the command **.HS I** at the beginning of the document. Each section is indented an amount proportional to the level of the section heading. Finally, segments of unformatted text can be indented using the **.DS I** or **.ID** commands.

Character Fonts and Underlining

The commands **.B**, **.I**, and **.R** switch between boldface, italic (underline on lineprinters), and regular text. Another way to change fonts is the in–line command **\F** or **\f**, followed by the font name. For example, you can place word in bold type two ways:

```
.B word
```

or

```
\FBword\FR
```

Note that the in–line command lets you switch fonts between individual characters without intervening spaces.

The available fonts include:

R	Regular (Roman) font
B	Boldface font
I	Italic font
U	Underline letters and numbers
BI	Boldface italics
BU	Underlined boldface

Combination fonts with two–character font names (BI, BU, and BC) are requested by including a left parenthesis before the name, for example, \F(BI. Note that italics are available only for troff output.

The font is always reset to normal at the beginning of each paragraph. To have multiple paragraphs printed in some font other than Roman, change the string NF (Normal Font) from R to the font you want.

Special Characters

In addition to the normal ASCII characters that you can type directly from the keyboard, there are a number of special characters in **ms** that you enter using the form \(*xx*, where *xx* is the name of the character. For example, the bullet character is produced using the sequence \(bu, and the Greek letter π is producing using \(*p. The later topic *The Special Character Set* illustrates all the available special characters and their sequences.

Combinations of overstruck characters that are not available as predefined special characters can be constructed using the overstrike command \o'*chars*'. This prints the characters in string *chars* one on top of the other. For example, \o'\(ci\(mu' prints as ⊗.

Superscripts and Subscripts

Superscripts and subscripts can be entered using the sequences *{*superscript**} and *{*subscript**}. For example, X*{2*} prints as:

X^2

These sequences insert white space before or after the line so that the superscript or subscript does not touch the line above or below the current line.

Because the superscript and subscript command require reverse movement of the paper, place the command **.COL** at the beginning of the file. This processes reverse linefeeds correctly when the output is directed to devices without a reverse linefeed capability.

Character Size

The character size is given in points, with point sizes of 6–12, 14, 16, 18, 20, 22, and 24. The default is 10 points. You can use the commands **.SZ, .LG**, and **.SM** to temporarily change the point size. **.SZ** takes as an argument an absolute point size, or a relative point size, for example **.SZ –3**. **.LG** is equivalent to **.SZ +2**, and **.SM** is equivalent to **.SZ –2**. As with font switches, an in–line command exists to change character sizes. The command \s$\pm n$ increases or decreases the point size by the specified amount. The command \s0 restores the previous point size.

As with fonts, the character size is automatically reset at the start of each paragraph. To have multiple paragraphs printed in a size other than 10 points, set the register PS to the desired size. Also reset register VS to specify the vertical spacing between the baselines of successive lines. Note how this differs from the command **.LS 2**, which causes a full blank line to be inserted between each line. The appropriate vertical spacing is set automatically by **.SZ, .SM**, and **.LG**, but is not changed by \s, so use \s for spacing changes within lines.

Command Descriptions

Following is a list of the commands available in **ms**, organized alphabetically. Optional arguments are enclosed in brackets. When you must choose one argument they are enclosed in braces.

.1C — Change back to one column after printing in two columns (see .2C).

.2C — Change to two–column output. Use **.BC** to move to the top of the next column. Use **.IC** to change back to one–column output. Causes reverse linefeeds to be inserted, so place a **.COL** at the beginning of the file for correct processing of reverse linefeeds.

.AB [no] — Begin an abstract. Follows the **.TL, .AU**, and **.AI** commands, if any. Text between the **.AB** and **.AE** commands is collected and printed as an indented block, with the word ABSTRACT centered before the text. Type **.AB no** to omit the word ABSTRACT.

.AD {0,1} — Set text even with the right margin. This is set by default. Use **.AD 0** to make a ragged right margin. The commands **.AD** or **.AD 1** turn adjustment back on.

.AE — End abstract (see **.AB**).

.AI Author's institution. The following lines give the group affiliation of the author(s) specified in the **.AU** command(s). Enter as many lines as necessary. Define an automatic increment number with name *c*. It is recommended that you use a single upper-case letter to avoid name conflicts. Entering *c causes 1 to be inserted the first time it is used, 2 the second time, and so on. This defines an automatic footnote number that can be used in combination with the superscript mechanism as follows:

```
text\*{\*F\*}
.FS \nF.
This is the footnote.
.FE
text
```

.AN You can reset the auto increment number to 0 by reissuing the **.AN** command. If *type* is included in the **.AN** command it specifies numbering other than the standard 1,2,3, etc. A *type* of **A** specifies the sequence A,B,C, etc. A *type* of **a** specifies a,b,c, etc. A *type* **I** specifies I,II,III,IV, etc. A *type* **i** specifies i,ii,iii,iv, etc. Note that **.PN** is really a predefined auto increment number that is incremented each time a new page is started. To print page numbers in small Roman numerals you could enter **.AN PN i**.

.AU The following line specifies the author of this document. For multiple authors, type several **.AU** commands.

.B [*text*] Print *text* in boldface. If no argument, switch to boldface.

.BC Begin a new column when in two-column input. Used like the **.BP** command for one-column input.

.BD [0] Start a centered block display. Lines up to the next **.DE** are collected and the entire block is centered. Identical to **.DS B**, except the display can extend onto the next page. It is preceded by a blank line unless you enter the optional argument 0.

.BE End a section to be enclosed in a box (see **.BS**).

.BP Begin a new page. It is ignored if you are already at the top of a page.

.BR Break the current line and start a new line.

.BS [C] Start a section of text to be enclosed in a box. The end of the boxed text is indicated using **.BE**. If the optional argument **C** is entered, the box is centered on the page. This causes reverse linefeeds to be inserted. Enter the **.COL** command at the beginning of the file for correct processing.

.BU [0] Start an indented paragraph marked with a bullet character. The paragraph is preceded with a blank line unless you enter the argument 0.

.BX *word* Enclose *word* in a box.

.CD [0] Start a centered display. Each line is centered relative to the current left and right margins. Identical to **.DS C**, except the display can extend onto the next page. Entering the argument 0 precedes the display with a blank line.

.CN Place the standard Tektronix Labs confidentiality note at this point in the document. This is done automatically at the bottom of the cover sheet for a technical report (**.TR**) or memo (**.TM**). You can redefine the macro to put a different note on the cover sheet. You can change this text in the file */usr/lib/tmac/tmac.sconfid.*

.COL When the document contains commands that cause reverse movement of the paper (super and subscripts, **.BX, .BS, .2C**) place this command at the beginning of the file.

.DA Places the current date at the bottom of each page. This is done using the CF string register. If you want the date to appear only on draft copies of the document, use the **.DR** command.

.DE [0] End a display (see **.DS, .ID, .LD, .BD**). Insert a blank line following the display unless the optional argument 0 is present.

.de *name* Define a command, where its *name* is one or two characters. Avoid using the names of existing **nroff** or **ms** commands. Since **nroff** commands are all lowercase and **ms** commands are all uppercase, a combination of upper and lowercase letters is always safe. Use **nroff** and **ms** commands that define the *name* on the following lines. The definition ends with two periods on the last line. Then each time **ms** encounters *name* all text and commands between **.de** and **..** are inserted. Any references to in–line commands that start with a backslash should contain two backslashes in a command definition. As a simple example, the following sequence defines a numbered paragraph command that automatically numbers lists:

```
.de Np
.IP (\\*N)
..
```

Then you can enter the following:

```
.AN N
.Np
First item
.Np
Second item
```

This produces the output:

(1) First item

(2) Second item

For more information on defining complex commands that accept arguments and include conditional sequences, see section 4B, *Nroff/Troff Reference Guide.*

.DR Insert the word DRAFT and the current date at the bottom of each page, immediately following the page footer. Normally placed at the beginning of a draft document.

.DS {**I,L,C,B**} [*indent*] [0] Start a display. Lines up to the next **.DE** are read and printed unchanged, except for their horizontal placement relative to the current left and right margins— L left justifies, C centers each line, B centers the entire block, and I indents 8 spaces. I is the default. If an I is present, you can change the indentation from 8 spaces by including a number *indent*. A blank line is inserted preceding the display unless the argument 0 is present. If the display does not fit on the current page, **ms** inserts a **.BP** to move the display to the top of the following page. Displays that can extend over page boundaries can be requested using **.ID, .LD, .CD**, and **.BD**.

.ds *name string* Define a string register. This is typically used to redefine one of the strings discussed in the later topic *String and Number Registers*. You can also use this to define your own strings. After you define the string, reference it using *x where x is a one-character string, or *xy where xy is a two-character string. As with **.de**, references to in-line commands preceded by a backslash require two backslashes. For example, a chapter (with Roman numerals) and page numbers within the chapter can be printed at the bottom of the page. Set up the string registers as follows:

```
.AN C I
.ds CH
.ds CF \\nC-\\n(PN
```

Each chapter would then start with the following sequence:

```
.PN 1
.BP
.SH CE
CHAPTER
.PP
```

.EH End a heading. This is necessary only when a heading appears on the same line as the following text.

.FE End a footnote (see **.FS**).

.FS [*label*] Begin a footnote. Text between **.FS** and **.FE** is saved and placed at the bottom of the page. The argument *label* is the footnote label placed at the left of the footnote, for example *.

.HL [*c*] Draw a horizontal line across the page from the current left margin to the current right margin. By default, the line is drawn using hyphens. If you enter *c*, that character is used to draw the line.

.HS {O,I} Specify heading style. Place this command at the beginning of the text. The default heading style is numbers, 1., 1.1, 1.1.1, and so on. The O argument specifies outline form, I., A., 1., a., i. The I argument specifies that numbered sections are automatically indented an amount proportional to the heading level. The default amount for each indentation is 4 spaces (the value of number register NI).

.HY {O,1} Set hyphenation. It is set by default. The command **.HY 0** turns off automatic hyphenation, while **.HY** or **.HY 1** turns it back on.

.I [*text*] Print *text* in italics.

.ID [*in*] [0] Start an indented display. Each line up to the next **.DE** is printed as you enter it, and indented 8 spaces or *in* spaces that you specify. This is identical to **.DS I** except that the display can extend onto the next page. It is preceded by a blank line unless 0 is present.

.IE [0] End an indented section. Identical to **.RE** followed by **.LP**, with the left margin restored to its value before the last **.IS**. It is followed by a blank line unless 0 is present.

.IOC [H]

Start a Tektronix Inter–office Communication (IOC). If the argument H is present, a fake letterhead is printed. The current date (which you can change using **.ND**) is printed along with information obtained from the following commands:

.TO The person to whom the IOC is being sent. You can use up to five **.TO** commands. To create a distribution list, type **.TO** by itself, and follow it with the recipients, one per line. This places the word "Distribution" in the "To" field, and a distribution list (in two columns if necessary) is placed at the bottom of the page. Note that distribution lists kept in separate files can be included using the **.RD** command. You can change this use of the word "Distribution" to another word using the Di register.

.FR The person sending the IOC. You can use up to 5 **.FR** commands for multiple senders.

.CC People to whom you are sending copies of the memo. Enter up to five names, or enter only **.CC** followed by names that are placed at the end of the IOC.

.SU The subject of the memo. This also appears in the page header if it has more than one page.

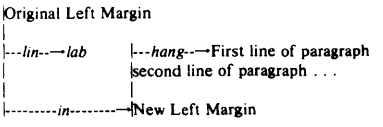
.DA The date of a meeting.

.TI The time of a meeting.

.PL The place of a meeting.

These commands are followed by an **.LP**, **.PP**, **.SH**, or **.NH** to start the text of the IOC. See the examples at the end of this section for sample memos.

.IP [*lab [in [lin [hang]]] [0]* Start an indented paragraph. The text is indented *in* spaces, the label *lab* is indented *lin* spaces, and the first line of text is indented *hang* spaces from the rest of the text. The argument *hang* can be negative to cause a hanging indent. For example:



The default label is "" (the null string). The default indents are 5 (the value of number register PI), 0, and 0 unless the immediately preceding paragraph is also started with **.IP**. In that case, the values specified there are used. If *lab* contains any blanks, enter them as unpaddable spaces, for example long label. The paragraph is preceded by a blank line unless a final 0 is present.

.IS [*in*] [0] Start an indented section. All the text up to the next **.IE** is indented *in* spaces. The default value for *in* is 5, and this value is in number register PI. Preceded by a blank line unless 0 is present.

.JU *'left' center' right'* Justify a single line. The text *left* is at the left margin, *center* is centered, and *right* is at the right margin. Any of the fields can be empty. For example, to right justify some text type:

```
.JU "'Right justify this'
```

If any of the fields have several spaces together, enclose the entire string in double quotes, outside the first and last single quotes.

.KE End a keep (See **.KS** and **.KF**).

.KF Start a floating keep. All text up to the next **.KE** is collected and printed on the current page. If it doesn't fit on one page, it is moved to the top of the next page. The **.HL** command is useful for setting floating keeps off from the rest of the text.

.KS Start a regular keep. All text up to the next **.KE** is collected and printed, preceded by a page eject if it does not fit on the current page. The **.DS** command does an automatic **.KS**, so it is rarely used.

.LS [0] Start a left–justified display. Each line up to the next **.DE** is printed as you type it. This is identical to **.DS L**, except that the display can extend onto the next page. I is preceded by a blank line unless the argument 0 is present. Start a left–blocked paragraph. Precede with a blank line unless the argument 0 is present.

.LS *n* Set the line spacing. The default line spacing is single space. Enter **.LS 2** to switch to double–spacing, and **.LS 1** to switch back to single–spacing.

.LT [*in*] Start a letter. This skips over the letterhead, prints the current date, and accepts an address and salutation. Star the text of the letter with **.PP** or **.LP**. The argument *in* is the number of spaces to indent the date from the left margin. If you don't enter anything for this argument, the date is adjusted to the right margin.

.ND *date* Set a date different than the current date.

.NE *n* Begin a new page if less than *n* lines remain on the current page. You can specify *n* in inches (*n*i) or in centimeters (*n*c). This command is done automatically for displays and keeps.

.NH [*level*] [*font*] [CE] Print a numbered heading at level *level*. The default value of *level* is 1, and that produces a top–level heading. The heading is preceded by a blank line. If you enter the CE argument, the heading is centered. If *font* is present, the heading is printed in that font. Otherwise, the heading is printed in boldface (string register HF). You can also set the register HS to print headings in different point sizes. The text following the heading is automatically indented an appropriate amount if you use the **.HS I** command. The heading should follow the **.NH** command and precede a paragraph start command.

.NL Return the type size to normal.

.nr Set a number register of *name* to *value*. Typically, this is used to redefine the existing number registers.

.P1 Place this command at the beginning of the file to print the header at the top of page one on all pages.

.PN *n* Set the page number of the next page to *n*.

.PP [0] Start a normal paragraph, indenting the first line five spaces from the left margin. Precede with a blank line unless the argument 0 is present.

.QE [0] End a quoted section (see.**QS**). Insert a blank line unless the argument 0 is present.

.QP [0] Start a quoted paragraph that that is indented 5 spaces from both the left and right margins (the value of number register QI). Precede with a blank line unless the argument 0 is present.

.QS [0] Start a quoted section, with both margins moved in five spaces (the value of number register QI). The margins remain in effect until the next **.QE**. Precede with a blank line unless the argument 0 is present.

.R Return to regular (Roman) font.

.RD [*file*] Read input from *file* just as if it were included in the text. If *file* is missing, read from the standard input until a blank line is encountered.

.RE End a relative indented section (see **.RS**).

.RP Start a released paper with a title page. The same format as **.TR**, but without a cover sheet.

.RS Start a relative indented section. This follows an indented paragraph to make succeeding paragraph commands relative to the current left margin. This margin is in effect until the next **.RE**.

.SH [*level*] [*font*] [CE] Print a non-numbered heading using *font*, preceded by a blank line. The default font is boldface (string register HF). If the argument CE is present, the heading is centered. The argument *level* is ignored, except that it indicates the indentation level in the table of contents for a following **.TC** command.

.SP [*n*] Insert *n* blank lines (or *n*i for inches and *n*c for centimeters). If you do not enter *n*, the value of register PD is used for the distance to space, which is 1 line by default. This command at the beginning of a page is ignored. To ensure that blank lines appear in the text, put the **.SP** inside a display.

.TA *t1 t2..*

Set tabs. Use inside a display to set the tab stops for column alignment. You can enter them as character positions (8n 16n . . .), inches (0.5i 1i . . .), or centimeters (1c 2c . . .). In addition to normal tab stops, that cause the text following the tab to be left–justified at the stop, you can also specify right–justifying or centering tab stops by following the tab stop with R or C. Note the following example, where Ⓣ is the tab character:

```
.DS B
.TA 2iC 4iR
\FIItemⓉManufacturerⓉCost\FR
.SP
BaublesⓉPenneysⓉ$5.98
BanglesⓉSearsⓉ$234.98
BeadsⓉMontgomery WardsⓉ$49.95
.DE
```

This produces the following output:

Item	Manufacturer	Cost
Baubles	Penneys	$5.98
Bangles	Sears	$234.98
Beads	Montgomery Wards	$49.95

Note that tab stops are set, but not returned to their original values by some **ms** commands (like **.IP**), so you need to reset the tab stops for displays.

.TC *text*

Place *text* in the table of contents and also include it in the text. If *text* has more than one space in a row, enclose it in double quotes. Use this after a section heading to place the heading title in the table of contents. For example:

```
.NH 1
.TC This is the section heading.
.PP
```

The table of contents is preceded by the word CONTENTS, centered in italics. To change this heading, redefine the **.PC** command.

.TL	The title of the document follows. It is printed in the center of the first page, in boldface.
.TM [*number*]	Start a technical memo. The argument *number* is the optional memo number printed on the cover sheet.
.TR [*number*]	Start a technical report. The argument *number* is the optional report number printed on the cover sheet.
.UL *word* [*x*]	Underline *word*. Make any spaces in *word* unpaddable, for example long\ word. If the argument *x* is present, it is appended immediately after *word*, without intervening spaces.
.XN *text*	Add *text* to the index without a page number.
.XX *text*	Add *text* to the index with the current page number. The index is automatically sorted and printed at the end of the document. It is preceded by the word INDEX, centered in italics. To change the INDEX heading, you can redefine the **.PX** command.

The –T Option

The following options are available on the **nroff/troff** command line to specify the output terminal.

–Taaa	The Ann Arbor Ambassador.
–Tascii	This is the default terminal. It specifies a standard ascii terminal with spacing at 10 chars/inch and 6 lines/inch. The only non–standard feature is the use of ESC 7 for reverse linefeeds.
–Tlp	The generic ascii lineprinter.
–Tlp–t	A Printronix printer in 3 lines per inch mode.
–Tlp–t–8	A Printronix printer in 4 lines per inch mode.
–Tlp–8	A Printronix printer in 8 lines per inch mode.
–Tq–lg	The Qume Sprint–5 printer in 12 pitch and 6 lines per inch mode.
–Tq–lg–8	The Qume Sprint–5 printer in 12 pitch and 8 lines per inch mode.
–Tq–lg–10	The Qume Sprint–5 printer in 10 pitch and 10 lines per inch mode.
–Tq–8	The Qume Sprint–5 printer in 12 pitch and 8 lines per inch mode.
–Tq–10	The Qume Sprint–5 printer in 10 pitch and 8 lines per inch mode.
–Ttn300	The GE TermiNet 300, or any terminal without half–line capability.
–Tup	The standard lineprinter driver in 10 pitch and 6 lines per inch mode.

-Tvt100 The DEC vt-100 and compatible terminals.

-TX The generic ENCDIC printer.

-T37 The TELETYPETm Model 37 terminal.

-T300 The DASI 300 terminal.

-T300s The DASI 300s terminal.

-T300-12 The DASI 300 terminal in 12 pitch mode.

-T300s-12 The DASI 300s terminal in 12 pitch mode.

-T382 The DTC-382.

-T450 The DASI 450.

-T450-12 The DASI 450 terminal in 12 pitch mode.

-T450-12-8 The DASI 450 terminal in 12 pitch and 8 lines per inch mode.

-T832 The Anderson Jacobson 832.

-T2631 The Hewlett Packard 2631 lineprinter.

-T2631-c The Hewlett Packard 2631 lineprinter in compressed mode.

-T2631-e The Hewlett Packard 2631 lineprinter in expanded mode.

-T4025 The Tektronix 4025 terminal.

Special Character Set

Following are all the special characters available in **ms**. The printed character is first, then the input name, followed by an explanatory name.

α	\(*a	alpha	O	\(*O	Omicron
β	\(*b	beta	Π	\(*P	Pi
γ	\(*g	gamma	P	\(*R	Rho
δ	\(*d	delta	Σ	\(*S	Sigma
ε	\(*e	epsilon	T	\(*T	Tau
ζ	\(*z	zeta	Y	\(*U	Upsilon
η	\(*y	eta	Φ	\(*F	Phi
θ	\(*h	theta	X	\(*X	Chi
ι	\(*i	iota	Ψ	\(*Q	Psi
κ	\(*k	kappa	Ω	\(*W	Omega
λ	\(*l	lambda	\	\e	back slash
μ	\(*m	mu	'	'	close quote
ν	\(*n	nu	`	`	open quote
ξ	\(*c	xi	´	\(aa	acute accent (or \')
o	\(*o	omicron	`	\(ga	grave accent (or \`)
π	\(*p	pi	-	\(hy	hyphen (or -)
ρ	\(*r	rho	−	\(mi	math minus (or \-)
σ	\(*s	sigma	—	\(--	short dash
ς	\(ts	terminal sigma	*	\(**	math star
τ	\(*t	tau	×	\(mu	multiply
υ	\(*u	upsilon	÷	\(di	divide
φ	\(*f	phi	•	\(bu	bullet
χ	\(*x	chi	□	\(sq	square
ψ	\(*q	psi	○	\(ci	circle
ω	\(*w	omega	°	\(de	degree
A	\(*A	Alpha	_	\(ru	rule
B	\(*B	Beta	¼	\(14	1/4
Γ	\(*G	Gamma	½	\(12	1/2
Δ	\(*D	Delta	¾	\(34	3/4
E	\(*E	Epsilon	†	\(dg	dagger
Z	\(*Z	Zeta	‡	\(dd	double dagger
H	\(*Y	Eta	′	\(fm	foot mark
Θ	\(*H	Theta	§	\(sc	section
I	\(*I	Iota	¢	\(ct	cent sign
K	\(*K	Kappa	®	\(rg	registered
Λ	\(*L	Lambda	©	\(co	copyright
M	\(*M	Mu	/	\(sl	alternate slash
N	\(*N	Nu	√	\(sr	square root
Ξ	\(*C	Xi		\(rn	root en extender

5318-16

≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≠	\(!=	not equal
≃	\(~=	approx =
~	\(ap	approximates
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of
|	\(br	box vertical rule
☛	\(rh	right hand
☚	\(lh	left hand
Ⓐ	\(bs	bell symbol
|	\(or	or
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bottem
⎨	\(lk	left center
⎬	\(rk	right center
⎪	\(bv	bold vertical
⎣	\(lf	left floor (left bottom of big square bracket)
⎦	\(rf	right floor (right bottom)
⎡	\(lc	left ceiling (left top)
⎤	\(rc	right ceiling (right top)
—	*-	long dash
*	**	big star
“	*(“	open doublequote
”	*(”	close doublequote
≅	*(~=	approx. equal
≐	*(.=	dot equal
∨	*(lo	logical or
∧	*(la	logical and
∴	*(tf	therefore
Å	*(ag	Angstrom
é	*'e	acute accent mark
è	*`e	grave accent mark
ü	*:u	umlat
ê	*^e	caret
ã	*~a	tilde
č	*Ce	Czech v
ç	*,c	cedilla

Available Point Sizes:

Point Size 6
Point Size 7
Point Size 8
Point Size 9
Point Size 10
Point Size 11
Point Size 12
Point Size 14
Point Size 16
Point Size 18
Point Size 20
Point Size 22
Point Size 24

5318-17

String and Number Registers

Following are the initial definitions of some of the string and number registers. You can change them using the **.ds** and **.nr** commands. Enter numbers within a "scale indicator": **n** for character positions, **c** for centimeters, **v** for vertical lines, or **i** for inches.

.ds LH	Left portion of page header (initially null)
.ds CH –\\n(PN–	Center portion of page header
.ds RH	Right portion of page header
.ds LF	Left portion of page header
.ds CF	Center portion of page footer (*(DY if .DA)
.ds RF	Right portion of page footer
.ds NF R	Normal text font
.ds HF B	Heading font (.SH, .NH).
.ds DI Distribution	Default for missing .TO argument in IOC
.nr HS 0	Heading size in points (0 means no change)
.nr LL 6i	Line length (6.5 inches for IOC)
.nr LT 6i	Header/footer length (6.5 inches for IOC)
.nr FL 6i–3n	Footnote line length
.nr HM 1i	Top margin (header in middle of margin)
.nr FM 1i	Bottom margin (footer in middle of margin)
.nr PI 5n	Paragraph indent (.PP, .IP, .IS)
.nr QI 5n	Quoted section indent (.QP, .QS)
.nr NI 4n	Auto indent for numbered sections (.HS I)
.nr PS 10	Character point size (range 6–24)
.nr VS 12	Vertical spacing (normally PS + 2)
.nr CS 24	Constant spacing character width (.CS)

Example 1 — A Simple Document

```
.LP
Note that every document must start with a command\*-LP
is the simplest such command.
Let's try a simple list:
.IP 1.
This is the first item in the list.
It will have a "label" of 1.
.IP 2.
This is the second item in the list.
Now let's start a "relative indent" section so that the
following sublist will appear indented from item 2.
.RS
.IP a)
This is sublist item a.
.IP b)
This is sublist item b.
.RE
.IP 3.
This is the third item in the top-level list.
.LP
We're now back at the original left margin with a new
left-justified paragraph.
Now let's do a "bullet list":
.BU
First bullet.
This can go on for awhile before we get to . . .
.BU
The second (and last) bullet.
.LP
We can offset quotations:
.QP
This famous quotation includes both \FIitalics\FR and
\FBboldface\FR text.
It is set off from both the right and left margins.
.LP
We can also offset unformatted text via the use of
displays:
.DS C
These lines are centered
exactly as they are typed.
.DE
```

5318-09

Note that every document must start with a command—LP is the simplest such command. Let's try a simple list:

1. This is the first item in the list. It will have a "label" of 1.
2. This is the second item in the list. Now let's start a "relative indent" section so that the following sublist will appear indented from item 2.
   a) This is sublist item a.
   b) This is sublist item b.
3. This is the third item in the top-level list.

We're now back at the original left margin with a new left-justified paragraph. Now let's do a "bullet list":

- First bullet. This can go on for awhile before we get to . . .
- The second (and last) bullet.

We can offset quotations:

> This famous quotation includes both *italics* and **boldface** text. It is set off from both the right and left margins.

We can also offset unformated text via the use of displays:

These lines are centered
exactly as they are typed.

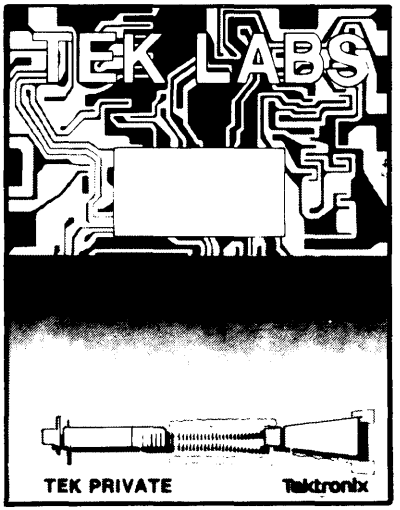
5318-10

Example 2 — A Technical Report

```
.TR
.TL
An Investigation into the Nature of Research
.AU
A. R. Researcher
.AI
Research Research Group
Applied Research Group
Tektronix Labs
.AB
This report contains a detailed analysis of a recently
completed five-year research program into the nature of
research carried out by the ARG Research Research Group.
It is a prelude to a newly-formed research project*
.FS *
See "A Long-Term Plan to Research Research Research" by
A. R. Researcher, et al.
.FE
to investigate the potential impact of research research
on ARG research programs.
.AE
.SH
Background
.PP
Research Research is a growing and dynamic discipline . . .
```

The .TR will produce a cover sheet with a technical report confidentiality statement placed automatically at the bottom of the page. Note that the .TR could be replaced with .RP to cause the technical report cover sheet and title page to be replaced with a released paper title page, or could be removed altogether if no cover page is desired.

5318-11



An Investigation into the Nature of Research

A. R. Researcher

Research Research Group
Applied Research Group
Tektronix Labs

February 3, 1981

This information is confidential and no further disclosure thereof can be made to other than Tektronix personnel without written authorization from the Director of Tek Labs, Tektronix, Inc., Beaverton, Oregon.

An Investigation into the Nature of Research

ABSTRACT

This report contains a detailed analysis of a recently completed five-year research program into the nature of research carried out by the ARG Research Research Group. It is a prelude to a newly formed research project to investigate the potential impact of research research on ARG research programs.

"See "A Long-Term Plan to Research Research Research" by A. R. Researcher et al.

An Investigation into the Nature of Research

A. R. Researcher

Research Research Group
Applied Research Group
Tektronix Labs

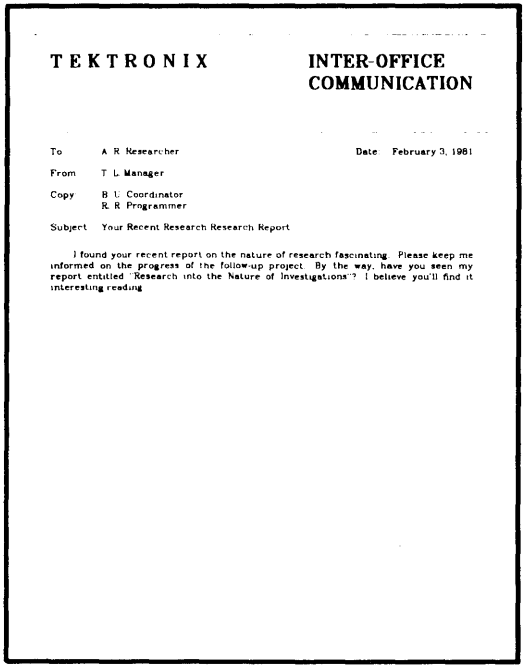
Background

Research Research is a growing and dynamic discipline.

5318-12

Example 3 — An IOC

```
.IOC H
.TO A. R. Researcher
.FR T. L. Manager
.CC B. U. Coordinator
.CC R. R. Programmer
.SU Your Recent Research Research Report
.PP
I found your recent report on the nature of research
fascinating.
Please keep me informed on the progress of the
follow-up project.
By the way, have you seen my report entitled "Research
into the Nature of Investigations"?
I believe you'll find it interesting reading.
```



TEKTRONIX

INTER-OFFICE COMMUNICATION

To A R Researcher

Date February 3, 1981

From T L Manager

Copy B U Coordinator
R R Programmer

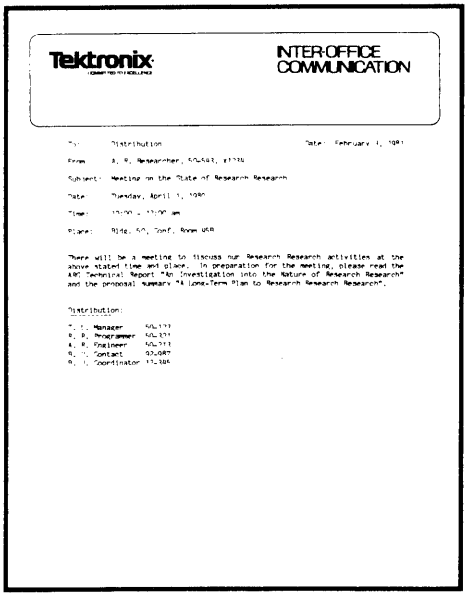
Subject Your Recent Research Research Report

I found your recent report on the nature of research fascinating. Please keep me informed on the progress of the follow-up project. By the way, have you seen my report entitled "Research into the Nature of Investigations"? I believe you'll find it interesting reading.

5318-13

Example 4 — An IOC Announcing a Meeting

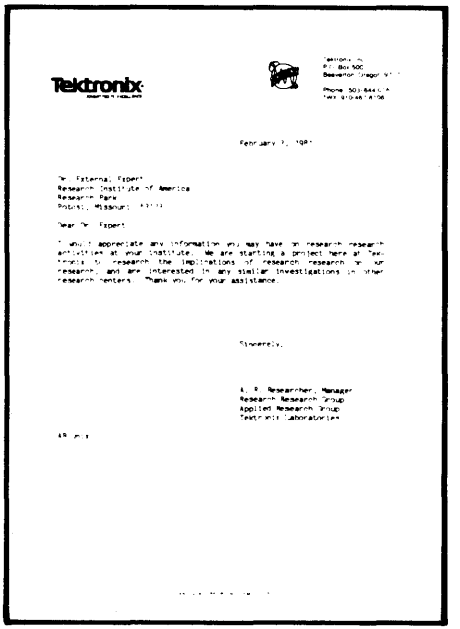
```
.IOC
.TO
T. L. Manager                50-123
R. R. Programmer             50-321
A. R. Engineer               50-213
B. U. Contact                92-987
B. U. Coordinator            12-345
.FR A. R. Researcher, 50-543, x1234
.SU Meeting on the State of Research Research
.DA Tuesday, April 1, 1980
.TI 10:00 - 12:00 am
.PL Bldg. 50, Conf. Room 45B
.LP
There will be a meeting to discuss our Research Research
activities at the above stated time and place.
In preparation for the meeting, please read the ARG
Technical Report "An Investigation into the Nature of
Research Research" and the proposal summary "A Long-Term
Plan to Research Research Research".
```



5318-14

Example 5 — A Business Letter

```
.LT 40
Dr. External Expert
Research Institute of America
Research Park
Potosi, Missouri  63123
.SP
Dear Dr. Expert:
.LP
I would appreciate any information you may have on
research research activities at your institute.
We are starting a project here at Tektronix to research
the implications of research research on our research,
and are interested in any similar investigations in other
research centers.
Thank you for your assistance.
.SP 5
.DS I 40
Sincerely,
.SP 4
A. R. Researcher, Manager
Research Research Group
Applied Research Group
Tektronix Laboratories
.DE
AR:unix
```



5318-15

The MM Text-formatting Macros

Introduction

This section is a guide and reference manual for users of Memorandum Macros (MM). These macros provide a general purpose package of text formatting macros for use with the UTek operating system text formatters **nroff** and **troff**.

Conventions

Each part of this chapter explains a single facility of MM and progresses from general case to special case facilities. It is recommended that users read a part in detail only to the point where there is enough information to obtain the desired format, then skim the rest. This should save users time, because some details may be of use to only a few.

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

In those cases in which the behavior of the two formatters **nroff** and **troff** is obviously different, the **nroff** formatter output is described first with the **troff** formatter output following in parentheses.

For example:

The title is underlined (italic).

means that the title is underlined by the **nroff** formatter
and italicized by the **troff** formatter.

Document Structure

Input for a document to be formatted with the MM text formatting macro package has four major segments. Any of these may be omitted; if present, the segments must occur in the following order:

- *Parameter setting segment* sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for heading and lists, page headers and footers, and many other properties of the document. The user can also add macros or redefine existing ones. This segment can be omitted entirely if the user is satisfied with the default values; it produces no actual output, but performs only the formatter setup for the rest of the document.

- *Beginning segment* includes those items that occur only once at the beginning of a document. This may include such items as title, author's name, and date.
- *Body segment* is the actual text of the document. It may be as small as a single paragraph or as large as hundreds of pages. It may have a hierarchy of headings up to seven levels deep. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of list macros for automatic numbering, alphabetic sequencing, and "marking" of list items. The body may also contain various types of displays, tables, figures, references, and footnotes.
- *Ending segment* contains those items that occur only once at the end of a document. Included are signature(s), in addition to notations, such as "Copy to" lists. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet.

Existence and size of these four segments varies widely among different document types. Although a specific item (such as author names, titles, dates, etc.) of a segment 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:

- Input lines should be kept short.
- Lines should be broken at the end of clauses.
- Each new sentence should begin on a new line.

Definitions

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

Requests are built-in commands recognized by the formatter. Although a user seldom needs to use these requests directly, this section contains references to some of the requests. For example, the request

```
.sp
```

inserts a blank line in the output at the place the request occurs in the input text file.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. The MM package supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

A complete listing of memorandum macros is given in the *MM Macro Name Summary* at the end of this section.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. A string can be given a value via the **.ds** (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for one–character names) or "*(" (for two–character names). For instance, the string **DT** in MM normally contains the current date; thus, the input line

Today is October 23, 1984.

may result in the following output:

Today is July 25, 1983.

The current date can be replaced with the following command:

.ds DT 01/01/79

This is done by invoking a macro designed specifically for that purpose. A listing of MM string names is given in the *MM String Name Summary* at the end of this section.

Number registers fill the role of integer variables. These registers are used for options and for arithmetic and automatic numbering. The registers share the pool of names ussed by requests and macros. A register can be given a value using a **.nr** request and be referenced by preceding its name by "\n" (for one–character names) or "\n(" (for two–character names). For example, the following sets the value of the register **d** to one more than that of the register **dd**:

.nr d 1+0

A complete listing of MM number registers is given in the *MM Number Register Summary* at the end of this section.

Also later in this section are naming conventions for requests, macros, strings, and number register. Following them are lists of all macros, strings, number registers, and error messages defined in MM.

Usage

This part describes how to access MM, illustrates UTek operating system command lines appropriate for various output devices, and describes command line options for the MM text formatting macro package.

The mm Command

The **mm** command can be used to prepare documents using the **nroff** formatter and the MM macro package; this command invokes **nroff** with the *-cm* option. The **mm** command has options to specify preprocessing by **tbl** and/or by **neqn** and for postprocessing by various output filters.

NOTE

Options can occur in any order but must appear before the filenames.

Any arguments or options that are not recognized by the **mm** command (such as **-rC3**) are passed to the **nroff** formatter or to MM, as appropriate. Options are:

OPTIONS	MEANING
-e	The **neqn** is invoked; also causes **neqn** to read */usr/pub/eqnchar* (see **eqnchar**).
-t	The **tbl** preprocessor is invoked.
-c	The **col** postprocessor is invoked.
-E	The **-e** option of the **nroff** formatter is invoked.
-y	The uncompacted macros (**-mm**) are to be used instead of compacted macros (**-cm**).
-12	The 12-pitch mode is to be used. The pitch switch on the terminal should be set to 12 if necessary.
-T450	Output is to a DASI 450. This is the default terminal type (unless $TERM is set; see **sh**). It is also equivalent to -T1620.
-T450-12	Output is to a DASI 450 in 12-pitch mode.
-T300'	Output is to a DASI 300 terminal.
-T300-12	Output is to a DASI 300 in 12-pitch mode.
-T300s	Output is to a DASI 300S.
-T300s-12	Output is to a DASI 300S in 12-pitch mode.
-T4014	Output is to a TEK 4014.
-T37	Output is to a TELETYPE Model 37.
-382	Output is to a DTC-382.
-T4000a	Output is to a TRENDATA 4000A.
-TX	Output is prepared for an EBCDIC line printer.
-Thp	Output is to a Hewlett-Packard 262x or 264x (implies **-c**).
-T43	Output is to a TELETYPE Model 43 (implies **-c**).

-T40/4	Output is to a TELETYPE Model 40/4 (implies **-c**).
-T745	Output is to a Texas Instrument 700 Series terminal (implies **-c**).
-T2631	Output is prepared for a Hewlett-Packard 2631 printer where **-T2631-e** and **-T2631-c** may be used for expanded and compressed modes, respectively (implies **-c**).
-Tlp	Output is to a device with no reverse or partial line motions or other special features (implies **-c**).

Any other **-T** option given does not produce an error; it is equivalent to **-Tlp**. A similar command is available for use with the **troff** formatter (see **mmt**).

The -cm or -mm Option

The MM package can also be invoked by including the **-cm** or **-mm** option as an argument to the formatter. The **-cm** option causes the precompacted version of the macros to be loaded.

NOTE

The **-cm** *option cannot be used with* **troff** *(device independent). The* **troff** *formatter does not allow compacted macros.*

The **-mm** option causes the file */usr/lib/tmac/tman.m* to be read and processed before any other files. This action:

- Defines the Memorandum Macros,
- Sets default values for various parameters,
- Initializes the formatter to be ready to process input files.

Typical Command Lines

The prototype command lines are as follows:

- Text without tables or equations:

mm [*options*] *filename* . . .
or
nroff[*options*] **-cm** *filename* . . .
mmt [*options*] *filename* . . .
or
troff [*options*] **-mm** *filename* . . . .

- Text with tables:

mm –t [*options*] *filename*...
or
tbl *filename*... ¦ **nroff** [*options*] **–cm**
mmt –t [*options*] *filename*...
or
tbl *filename*... ¦ **troff** [*options*] **–mm**
or

- Text with equations:

mm –e [*options*] *filename*...
or
neqn */usr/pub/eqnchar filename*... ¦ **nroff** [*options*] *-cm*
mmt –e [*options*] *filename*...
[*options*] *-mm*

- Text with both tables and equations:

mm–t –e [*options*] *filename*...
or
tbl *filename*... ¦ **neqn** */usr/pub/eqnchar-* ¦ **nroff**
[*options*] **–cm**
mmt –t –e [*options*] *filename*...

NOTE

On any line shown above with a call to **nroff** *using the* **–cm** *options, the* **–mm** *options may be used instead of* **–cm.**

When formatting a document with the **nroff** processor, the output should normally be processed for a specific type of terminal. This is because the output may require some features that are specific to a given terminal, such as reverse paper motion or half–line paper motion in both directions.

Some commonly used terminal types and the command lines appropriate for them are given below. More information is found later in this section, and in *term(4)* of the *UTek Command Reference.*

- DASI 450 in 10–pinch, six lines/inch mode, with 0.75 inch offset, and a line length of six inches (60 characters). This is the default terminal type; therefore, no **–T** option is needed (unless $TERM is set to another value):

mm *filename*...
or
nroff –T450 –h –cm *filename*...

- DASI 450 in 12–pitch, six line/inch mode, with 0.75 offset, and a line length of six inches (72 characters):

mm –12 *filename*...
or
nroff –T450 –12 –f –cm *filename*...

To increase the line length to 80 characters and decrease the offset to three characters:

mm –12 –rW80 –rO3 *filename*...
or
nroff –T450–12 –rW80 –rO3 –h –cm *filename*...

- Hewlett–Packard HP262x or HP264x CRT family:

mm –Thp *filename*...
or
nroff –cm *filename*... ¦ **col** ¦ **hp**

- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 Series, etc.):

mm –T745 *filename*...
or
nroff –cm *filename*... ¦ **col –x**

The **tbl** and **neqn** preprocessors must be invoked as shown in the command lines illustrated earlier.

If two–column processing is used with the **nroff** formatter, either the **–c** option must be specified to **mm** (**mm** uses **col** automatically for many terminal types), or the **nroff** formatter output must be postprocessed by **col**. In the latter case, the T37 terminal type must be specified to the **nroff** formatter, the **–h** option must not be specified, and the output of **col** must be processed by the appropriate terminal filter (such as by 450); **mm**, 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 with **.nr** requests. In addition, some of these registers can be set from the command line. This is a useful feature for those parameters that should not be permanently embedded within the input text. If used, the number registers (with exception to the P register) must be set on the command line or before the MM macro definitions are processed. The number register meanings are as follows:

–rA*n*	*n* = 1 has effect of invoking the **.AF** macro without an argument.
–rC*n*	sets type of copy, such as DRAFT, to be printed at the bottom of each page. *n* = 1 for OFFICIAL FILE COPY. *n* = 2 for DATE FILE COPY.

n = 3 for DRAFT with single spacing and default paragraph style.
n = 4 for DRAFT with double spacing and 10–space paragraph indent.

–rD1 sets *debug* mode.
This option requests the 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 the **troff** formatter).
n = 1, fields are Roman font (regular text default for the **nroff** formatter).

–rL*k* sets length of physical page to *k* lines.
For the **nroff** formatter, *k* is an unscaled number representing lines.
For the **troff** formatter, *k* must be scaled.
Default value is 66 lines per page.
This option is used, for example, when directing output to a Versatec. printer.

–rN*n* specifies page numbering style.
n = 0 (default), all pages get the prevailing header.
n = 1, page header replaces footer on page one only.
n = 2, page header is omitted from page one.
n = 3, "section–page" numbering occurs (.FD and .RP defines footnotes and reference numbering in section.
n = 4, default page header is suppressed; however, a user–specified header is not affected.
n = 5, "section–page" and "section–figure" numbering occurs.

Table 4D–1
EFFECTS OF THE N REGISTER ON PAGE NUMBERING STYLE

n	PAGE 1	PAGE 2–END
0	header	header
1	header replaces footer	header
2	no header	header
3	"section–page" as footer	same as page 1
4	no header	no header unless **.PH** defined
5	"section–page" as footer and "section–figure"	same as page 1

Contents of the prevailing header and footer do not depend on number register *N* value; *N* controls whether the header (*N* = 3) or the footer (*N* = 5) is printed, as well as the page numbering style. If header and footer are null, the value of *N* is irrelevant.

-rO*k* offsets output *k* spaces to the right.
For the **nroff** formatter, *k* is an unscaled number representing lines or character positions.
For the **troff** formatter, *k* must be scaled.
This option is helpful for adjusting output positioning on some terminals. The default offset, if this register is not set on the command line, is 0.75 inch (**nroff**) and 0.5 inch (**troff**).

> **Note:** *This register name is the capital letter "O".*

-rP*n* specifies that pages of the document are to be numbered starting with *n*.
This register may also be set via a **.nr** request in the input text.

-rS*n* sets point size and vertical spacing for the document.
The default *n* is 10 (10-point type on 12-point vertical spacing, giving six lines per inch).
This option applies to the **troff** formatter only.

-rT*n* provides register settings for certain devices.
If *n* is one , line length and page offset are set to 80 and three, respectively.
Setting *n* to two changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer.
The default value for *n* is zero.
This option applies to the **nroff** formatter only.

-rU1 controls underlining of section headings.
This option causes only letters and digits to be underlined.
Otherwise, all characters (including spaces) are underlined.
This option applies to the **nroff** formatter only.

-rW*k* sets page width (line length and title length) to *k*.
For the **nroff** formatter, *k* is an unscaled number representing character positions.
For the **troff** formatter, *k* must be scaled.
This option can be used to change page width from the default value of six inches (60 characters in 10-pitch or 72 characters in 12-pitch).

Omission of -cm or -mm Options

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

zero or more initializations of registers listed in the last section
.so */usr/lib/tmac/tmac.m*
remainder of text.

In this case, the user must not use the **-cm** or **-mm** options (nor the **mm** or **mmt** commands); the **.so** request has the equivalent effect, but registers shown in the last section 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 action of the formatters is to fill output lines from one or more input lines. Output lines may be justified so that both the left and right margins are aligned. As lines are being filled, words may also be hyphenated as necessary. It is possible to turn any of these modes on and off (with **.SA**, *Hy*, and the **.nf** and **.fi** formatter requests). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause filling of the current output line to cease, the line (of whatever length) to be printed, and subsequent text to begin a new output line. This printing of a partially filled output line is known as *break*. A few formatter requests and most of the MM macros cause a break.

Formatter requests can be used with MM; however, there are consequences and side effects that each such request might have. A good rule is to use formatter requests only when absolutely necessary. The MM macros described herein should be used in more cases because:

- It is much easier to control (and change at any later point in time) the overall style of the document.
- Complicated features (such as footnotes or tables of contents) can be obtained with ease.
- User is insulated from peculiarities of the formatter language.

Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has special meaning; the preferred form for a null argument is "". *Omitting an argument is not the same as supplying a null argument!* Omitted arguments can occur only at the end of an argument list; null arguments can occur anywhere in the list.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes. A double quote (") is a single character that must not be confused with two apostrophes ('), acute accents ('), or grave accents(`). Otherwise, it will be treated as several separate arguments.

Double quotes are not permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If it is necessary to have a macro argument value, two grave accents (``) and/or two acute accents ('') may be used instead. This restriction is necessary because many macro arguments are processed (interpreted) a varying number of times. For example, headings are first printed in the text and may be reprinted 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 alignment of text. To avoid this distortion, it is necessary to specify a space that cannot be expanded during justification (an unpaddable space, for example). There are several ways to accomplish this:

- The user may type a backslash followed by a space (\ ). This pair of characters directly generates an unpaddable space.
- The user may sacrifice some seldom-used character to be translated into a space upon output.

Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used with the translation macro for this purpose. To use the tilde in this way, the following is inserted at the beginning of the document:

.tr~ *<sp>*

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting

.tr~~

before the place where needed. Its previous usage is restored by repeating the **.tr~** *<sp>* after a break or after the line containing the tilde has been forced out.

NOTE
Use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

Hyphenation

Formatters do not perform hyphenation unless requested. Hyphenation can be turned on in the body of the text by specifying

.nr Hy 1

at the beginning of the document input file. A special case exists for hyphenation within footnotes and across pages and is discussed later in this manual.

If hyphenation is requested, formatters will automatically hyphenate words if need be. However, the user may specify hyphenation points for a specific occurrence of any word with a special character, known as a *hyphenation indicator*, or may specify hyphenation points for a small list of words (about 128 characters).

If the hyphenation indicator (initially, the two–character sequence **\%** appears at the beginning of a word, the word is not hyphenated). Alternatively, it can be used to indicate legal hyphenation points inside a word. All occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator in the following way:

.HC [*hyphenation-indicator*]

The circumflex (^) is often used for this purpose by inserting the following at the beginning of a document input text file:

.HC^

NOTE

Any word containing hyphens or dashed (also know as em dashes) will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, even if the formatter hyphenation function is turned off.

The user may supply, via the exception word **.hw** request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word "printout", the user may specify

.hw print—out

Tabs

Macros **.MT, .TC,** and **.CS** use the formatter tab **.ta** request to set tab stops and then restore the default values of tab settings (every eight characters in the **nroff** formatter; every ½ inch in the **troff** formatter). Setting tabs to other than the default values in the user's responsibility.

Default tab setting values are 9, 17, 25, . . . , 161 for a total of 20 tab stops. Values may be separated by commas, spaces, or any other nonnumeric character. A user may set tab stops at any value desired. For example:

.ta 9 17 25 33 41 49 57 . . . 161

A tab character is interpreted with respect to its position on the input line rather than its position on the output line. In general, tab characters should appear only on lines processed in no–fill **(.nf)** mode.

The **tbl** program changes tab stops but does not restore default tab setting.

BEL Character

The nonprinting character **BEL** is used as a delimiter in many macros to compute the width of an argument or to delimit arbitrary text. For example, it may do so in page headers and footers, headings, and lists. Users who include **BEL** characters in their input text file especially in arguments to macros) will receive mangled output.

Bullets

A bullet (●) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". For compatibility with the **troff** formatter, a bullet string is provided by MM with the following sequence:

***(BU**

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

Dashes, Minus Signs, and Hyphens

The **troff** formatter has distinct graphics for a dash, a minus sign, and a hyphen; the **nroff** formatter does not.

- Users who intend to use the **nroff** formatter may use only the minus sign (-) for the minus, hyphen, and dash.
- Users who plan to use the **troff** formatter primarily should follow **troff** escape conventions.
- Users who plan to use both formatters must take care during input text file preparation. Unfortunately, these graphic characters cannot be represented in a way that is both compatible and convenient for both formatters.

The following approach is suggested:

Dash	Type *(**EM** for each text dash for both **nroff** and **troff** formatter. This string generates an em dash in the **troff** formatter and two dashes (—) in the **nroff** formatter. Dash list (**.DL**) macros automatically generate the em dash for each list item.
Hyphen	Type — and use as is for both formatters. The **nroff** formatter will print it as is. The **troff** formatter will print - (a true hyphen).
Minus	Type \- for a true minus sign regardless of formatter. The **nroff** formatter will ignore the \. The **troff** formatter will print a true minus sign.

Trademark String

A trademark string *(**Tm** is available with MM. This places the letters "TM" one-half line above the text that it follows. For example:

```
The
.I
UTek
.R
\*(Tm
.I
Command Reference Manual
.R
is available from the library.
```

yields:

The *UTek* ™ *Command Reference Manual* is available from the library.

Use of Formatter Requests

Most **nroff/troff** requests should not be used with MM because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests. However, some formatter requests are useful with MM, namingly the following:

.af	Assign format
.br	Break
.ce	Center
.de	Define macro
.ds	Define string
.fi	Fill output lines
.hw	Exception word
.ls	Line spacing
.nf	No filling of output lines
.nr	Define and set number register
.nx	Go to the next file (does not return)
.rm	Remove macro
.rr	Remove register
.rs	Remove spacing
.so	Switch to source file and return
.sp	Space
.ta	Tab stop settings
.ti	Temporary indent
.tl	Title
.tr	Translate
.	Escape

The **.fp, .lg,** and **.ss** requests are also sometimes useful for the **troff** formatter. Use of other requests without fully understanding their implications very often leads to disaster.

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 a nonindented paragraph is defined with the *type* argument:

type	RESULT
0	left justified
1	indent

In a left–justified paragraph, the first line begins at the left margin. In an indented paragraph, the paragraph is indented the amount specified in the **Pi** register (default value is five). For example, to indent paragraphs by ten spaces, the following is entered at the beginning of the document input file:

```
.nr Pi 10
```

A document input file possesses a default paragraph type obtained by specifying **.P** before each paragraph that does not follow a heading. Default paragraph type is controlled by the **Pt** number register.

- The initial value of **Pt** is zero, and provides left–justified paragraphs.
- All paragraphs can be forced to be indented by inserting the following at the beginning of the document input file:

  ```
  .nr Pt 1
  ```

- All paragraphs can be indented except after headings, lists, and displays by entering the following at the beginning of the document input file:

  ```
  .nr Pt 2
  ```

Both the **Pi** and **Pt** register values must be greater than zero for any paragraphs to be indented.

NOTE

Values that specify indentation must be unscaled and are treated as character positions (for example, as a number of ens). In the **nroff** *formatter, an en is equal to the width of a character. In the* **troff** *formatter, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size.*

Regardless of the value of **Pt**, an individual paragraph can be forced to be left–justified or indented. The **.P 0** macro request forces left justification; **.P 1** causes indentation by the amount specified by the register **Pi**.

If **.P** occurs inside a list, the indent (if any) of the paragraph is added to the current list indent.

Numbered Paragraphs

Numbered paragraphs may be produced by setting the **Np** register to 1. This produces paragraphs numbered within first level headings, such as 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the **.nP** macro rather than the **.P** macro for paragraphs. This produces paragraphs that are numbered within second level headings.

.H 1 *"FIRST HEADING"*
.H 2 *"Second Heading"*
.nP
one or more lines of text

The paragraphs contain a "double-line indent" in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

Spacing Between Paragraphs

The **Ps** number register controls the amount of spacing between paragraphs. By default, **Ps** is set to one, yielding one blank space (one-half vertical space).

Numbered Headings

.H *level* [*heading text*] [*heading-suffix*]
zero or more lines of text

The *level* argument provides the numbered heading level. There are seven heading levels; level one is the highest, level seven is the lowest.

The *heading-text* argument may be used for footnote marks which should not appear with heading text in the table of contents.

There is no need for a **.P** macro immediately after a **.H** or a **.HU** because the **.H** macro also performs the function of the **.P** macro. Anything immediately following **.P** macro is ignored. It is, however, good practice to start every paragraph with a **.P** macro, thereby ensuring that all paragraphs uniformly begin with a *.P* throughout an entire document.

Normal Appearance

The effect of the **.H** macro varies according to the *level* argument. First-level headings are preceded by two blank lines (one vertical space); all others are preceded by one blank line (one-half a vertical space). The following table describes the default effect of the *level* argument:

.H 1 *heading-text*	Produces an underlined (italicized) font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should be used to make the heading stand out.

.H 2 *heading-text* — Produces an underlined (italicized) font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Initial capitals should be used in the head text.

.H *n heading-text* — Produces an underlined (italicized) heading followed by two spaces ($3 \leq n \leq 7$). The following text begins on the same line.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading-text argument is omitted from a **.H** macro call.

The following list gives the first few **.H** calls used for this part:

```
.H 1 "Paragraphs and Headings"
.H 2 "Paragraphs"
.H 3 "Paragraph Indentation"
.H 3 "Numbered Paragraphs"
.H 3 "Spacing Between Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance"
.H 3 "Altering Appearance"
.H 4 "Prespacing and Page Ejection"
.H 4 "Spacing After Headings"
.H 4 "Centering Headings"
.H 4 "Bold, Italic, and Underlined Headings"
.H 5 "Control by Level"
```

NOTE

Users satisfied with the default appearance of headings may skip to the paragraph entitled "Unnumbered Headings".

Altering Appearance

The user can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document input text file. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines rather than being distributed throughout the document.

Prespacing and Page Ejection

A first-level heading (**.H 1**) normally has two blank lines (one vertical space) preceding it, and all other headings are preceded by one blank line (one-half a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document input text file. Long documents may be made more manageable if each section starts on a new page. Setting the **Ej** register to a higher value causes the same effect for headings up to that level. For example, a page eject occurs if the heading level is less than or equal to the **Ej** value.

Spacing After Headings

Three registers control the appearance of text immediately following a **.H** call. The registers are **Hb** (heading break level), **Hs** (heading space level), and **Hi** (post-heading indent).

- If the heading level is less than or equal to **Hb**, a break occurs after the heading.
- If the heading level is less than or equal to **Hs**, a blank line (one-half a vertical space) is inserted after the heading.
- If a heading level is greater than **Hb** and also greater than **Hs**, then the heading (if any) is immediately followed by text on the same line.

These registers permit headings to be separated from the text in a consistent way throughout a document while allowing easy alteration of white space and heading emphasis. The default value for **Hb** and **Hs** is 2.

For any stand-alone heading, such as a heading on a line by itself, alignment of the next line of output is controlled by the **Hi** number register.

- If **Hi** is 0, text is left-justified.
- If **Hi** is 1 (the default value), text is indented according to the paragraph type as specified by the **Pt** register.
- If **Hi** is 2, text is indented to line up with the first word of the heading itself so that the heading number stands out more clearly.

To cause a blank line (one-half a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of **Pt**), the following should appear at the beginning of the document input text file:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

Centered Headings

The **Hc** register can be used to obtain centered headings. A heading is centered if its *level* argument is less than or equal to **Hc** and if it is also a stand-alone heading. The **Hc** register is 0 initially (no centered headings).

Bold, Italic, and Underlined Headings

Control by Level: Any heading that is underlined by the **nroff** formatter is italicized by the **troff** formatter. The string **HF** (heading font) contains seven codes that specify fonts for heading levels one through seven. Legal codes, code interpretations, and defaults for **HF** codes are shown in Table 4D-2:

Table 4D–2
HF STRING CODES, EFFECTS, AND DEFAULT VALUES

FORMATTER	HF CODE			DEFAULT
	1	2	3	HF CODE
nroff	no underline	underline	bold	2 2 2 2 2 2
troff	Roman	italic	bold	2 2 2 2 2 2

Thus, all levels are underlined by the **nroff** formatter and italicized by the **troff** formatter. The user may reset **HF** as desired. Any value omitted from the right end of the list is assumed to be a 1. The following request would result in five bold levels and two underlined (italic) levels:

.ds HF 3 3 3 3 3

NROFF Underlining Style: The **nroff** formatter underlines in either of two styles:

- The normal style (**.ul** request) underlines only letters and digits.
- The continuous style (**.cu** request) underlines all characters including spaces.

By default, MM attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined but is longer than a single line, the heading is underlined in the normal style.

All underlining of headings can be forced to the normal style by using the **–rU1** option when invoking the **nroff** formatter.

Heading Point Sizes: The user may specify the desired point size for each heading level with the **HP** string (for use with the **troff** formatter only).

.ds HP *[ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]*

By default, the text of headings (**.H** and **.HU**) is printed in the same point size as the body except that bold stand–alone headings are printed in a size one point smaller than the body. The string **HP**, similar to the string **HF**, can be specified to contain up to seven values, corresponding to the seven levels of headings.
For example:

.ds HP 12 12 10 10 10 10 10

specifies that the first and second level headings are to be printed in 12–point type with the remainder printed in 10–point. Specified values may also be relative point–size changes. For example:

.ds HP +2 +2 –1 –1

If absolute point sizes are specified, then absolute sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then point sizes for headings will be relative to the point size of the body even if the latter is changed.

Null or zero values imply that default size will be used for the corresponding heading level.

NOTE

Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via **.HX** *and/or* **.HZ***) may cause overprinting.*

Marking Styles — Numerals and Concatenation

.HM [*arg1*] . . . [*arg2*]

The registers named **H1** through **H7** are used as counters for the seven levels of headings. Register values are normally printed using Arabic numerals. The **.HM** macro (heading mark style) allows this choice to be overridden thus providing "outline"and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal arguments and their meanings are as follows:

ARGUMENT	MEANING
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Uppercase alphabetic
a	Lowercase alphabetic
I	Uppercase Roman
i	Lowercase Roman
omitted	Interpreted as 1 (Arabic)
illegal	No effect

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks (in other words, to obtain just the current level mark followed by a period), the heading mark type register (**Ht**) is set to 1. For example, a commonly used "outline" style is obtained with the following:

```
.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 two. 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 two and resetting to zero the counters for levels three through seven. Typically, the value of **Hu** should be set to make unnumbered headings (if any) be the lowest–level headings in a document.

The **.HU** macro can be especially helpful in setting up appendices and other sections that may not fit well into the numbering scheme of the main body of a document.

Headings and Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following:

- Specifying in the contents level register, **Cl**, what level headings are to be saved.
- Invoking the **.TC** macro 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; in other words, the first two levels of headings are saved.

Due to the way headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays and footnotes. If this happens, the "Out of temp file space" formatter error message will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

First–Level Headings and Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the "section–page" form where *section* is the number of the current first–level heading. This page numbering style can be achieved by specifying the **–rN3** or **–rN5** option on the command line. As a side effect, this also has the effect of setting **Ej** to 1 (each first level section begins on a new page). In this style, the page number is printed at the bottom of the page so that the correct section number is printed.

User Exit Macros

NOTE
This paragraph is intended primarily for users who are accustomed to writing formatter macros.

.HX *d-level r-level heading-text*
.HY *d-level r-level heading-text*
.HZ *d-level r-level heading-text*

The **.HX, .HY,** and **.HZ** macros are the means by which the user obtains a final level of control over the previously described heading mechanism. These macros are not defined by MM; they are intended to be defined by the user. The **.H** macro call invokes **.HX** shortly before the actual heading text is printed; it call **.HZ** as its last action. After **.HX** is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in **.HX**, such as **.ti** for temporary indent, to be lost. After the size calculation, **.HY** is invoked so that the user may respecify these features. All default actions occur if these macros are not defined. If **.HX, .HY,** and **.HZ** are defined by the user, user-supplied definition is interpreted at the appropriate point. These macros can therefore influence handling of all headings because the **.HU** macro is actually a special case of the **.H** macro.

If the user originally invoked the **.H** macro, then the derived level argument (*d-level*) and the real level argument (*r-level*) are both equal to the level given in the **.H** invocation. If the user originally invoked the **.HU** macro, *d-level* is equal to the contents of the register **Hu**, and *r-level* is zero. 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 *r-level* 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 *r-level* is 0, this string is null.
register **;0**	This register indicates the type of spacing that 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 blank line (one-half a vertical space) is to follow the heading.
string **}2**	If register **;0** is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) heading from the follow text. If register **;0** is nonzero, this string is null.

register **;3** — This register contains an adjustment factor for a **.ne** request issued before the heading is actually printed. On entry to **.HX**, it has the value 3 if *d-level* equals 1, and 1 otherwise. The **.ne** request is for the following number of lines: the contents of the register **;0** taken as blank lines (halves of vertical space), plus the contents of register **;3** as blank lines (halves of vertical space), plus the number of lines of the heading.

The user may alter the values of **}0,}2,** and **;3** within **.HX**. The following are example actions that might be performed by defining **.HX** to include the lines shown. The notation *<sp>* denotes a space:

- Change first-level heading mark from format *n.* to *n.0*:

  .if\\$1=1 .ds }0\\n(H1.0<sp>****<sp>

- Separate run-in heading from the text with a period and two unpaddable spaces:

  .if\\n(;0=0 .ds }2.<sp>****<sp>

- Assure that at least 15 lines are left on the page before printing a first-level heading:

  .if\\$1=1.nr ;3 (15-\\n(;0)v

- Add three additional blank lines before each first-level heading:

  .if\\$1=1 .sp 3

- Indent level 3 run-in headings by five spaces:

  .if\\$1=3.ti 5n

If temporary strings or macros are used within **.HX**, their names should be chosen with care.

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; the user may want to change a page header of footer in the following way:

```
.de HZ
.if\\$1=1 .PF "Section \\$3"
..
```

Hints for Large Documents

A large document is often organized for convenience into one input text file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections (for example, use suffix numbers 01 though 20 rather than 1 through 9 and 10 through 20).

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register **H1** to one less than the number of the section just before the corresponding **.H 1** call. For example, at the beginning of Part 5, insert:

```
.nr H1 4
```

NOTE

This is not good practice. It defeats the automatic (re)numbering of sections when sections are added or deleted. Such lines should be removed as soon as possible.

Lists

This part describes different styles of lists: automatically numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings (such as those with terms or phrases to be defined).

List Macros

In order to avoid repetitive typing of arguments to describe the style or appearance of items in a list, MM provides a convenient way to specify lists. All lists share the same overall structure and are composed of the following basic parts:

- A *list-initialization* macro (**.AL, .BL, .MI, .RL**, or **.VL**) determines the style of list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing of list items.
- One or more *list-item* macros (**.LI**) identifies each unique item to the system. It is followed by the actual text of the corresponding list item.
- The *list-end* macro (**.LE**) identifies the end of the list. It terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list–initialization macro saves the previous list status (indentation marking style, etc.); the **.LE** macro restores it.

With this approach, the format of a list is specified only once at the beginning of the list. In addition, by building onto the existing structure, users may create their own customized sets of list macros with relatively little effort.

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 the **.AL** macro is called. This leaves room for a space, two digits, a period, and two spaces before the text. Values that specify indentation must be unscaled and are treated as *character positions* (such as number of ens).

Spacing at the beginning of the list and between items can be suppressed by setting the list space register (**Ls**). The **Ls** register is set to the innermost list level for which spacing is done.
For example:

.nr Ls 0

specifies that no spacing will occur around any list items. The default value for **Ls** is six (which is the maximum list nesting level).

- The *type* argument may be given to obtain a different type of sequencing. Its value indicates the first element in the sequence desired. If *type* argument is omitted or null, the value 1 is assumed.

ARGUMENT	INTERPRETATION
1	Arabic (default for all levels)
A	Uppercase alphabetic
a	Lower alphabetic
I	Uppercase Roman
i	Lowercase Roman

- If *text-indent* argument is nonnull, it is used as the number of spaces from the current indent to the text; for example, the argument is used instead of the **Li** register for this list only. If *text-indent* argument is null, the value of **Li** will be used.
- If the third argument is given, a blank line (one-half a vertical space) will *not* separate items in the list. A blank line will occur before the first item, however.

Bullet List

.BL [*text-indent*] [1]

The **.BL** macro begins a bullet list. Each list item is marked by a bullet (●) and followed by one space.

- If the *text-indent* argument is nonnull, it overrides the default indentation (the amount of paragraph indentation as given in the **Pi** register). In the default case, the text of a bullet list lines up with the first line of indented paragraphs.
- If the second argument is specified, no blank lines will separate the items in the list.

Dash List

.DL [text–indent] **[1]**

The **.DL** macro begins a dash list. Each list item is marked by a dash (—) and followed by one space.

- If the *text-indent* argument is nonnull, it overrides the default indentation (the amount of paragraph indentation as given in the **Pi** register). In the default case, the text of a dash list lines up with the first line of indented paragraphs.
- If the second argument is specified, no blank lines will separate items in the list.

Marked List

.ML *mark* **[***text-indent***] [1]**

The **.ML** macro is much like **.BL** and **.DL**, but it expects the user to specify an arbitrary *mark* which may consist of more than a single character.

- Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of the *mark*.
- If the third argument is specified, no blank lines will separate items in the list.

NOTE

The mark must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified.

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 nonnull, it is used as the number of spaces from the current indent to the text; for example, it is used instead of **Li** for this list only. If the *text-indent* argument is omitted or null, the value of **Li** is used.

- If the second argument is specified, no blank lines will separate the items in the list.

Variable-Item List

.VL *text-indent* [*mark-indent*] [1]

When a list begins with a **.VL** macro, there is effectively no current *mark*; it is expected that each **.LI** will provide its own mark. This form is typically used to display definitions of terms or phrases.

- *Text-indent* provides the distance from the current indent to the beginning of the text.
- *Mark-indent* produces the number of spaces from the current indent to the beginning of the *mark*. Its default is zero if it is omitted or null.
- If the third argument is specified, no blank lines will separate items in the list.

An example of **.VL** macro usage follows:

```
.tr~
.VL 20 2
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".
.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 lone mark;  one space
separates the mark from the text.
.LI~
This item effectively has no mark because the tilde
following the .LI is translated into a space.
.LE
```

When this is formatted, it 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.
```

The tilde argument on the last **.LI** in the previous example is required; otherwise, a *hanging indent* would have been produces. A hanging indent is produced by using **.VL** and calling **.LI** with no arguments or with a null first argument.
For example:

```
   .VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

When this is formatted, it yields:

```
Here is some text to show a hanging indent.  The first line of text
          is at the left margin.  The second is indented 10 spaces.
```

NOTE

The mark must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified.

List-Item Macro

.LI [*mark*] [1]
one or more lines of text that make up the list item.

The **.LI** macro is used with all lists and for each list item. It normally causes output of a single blank line (one–half a vertical space) before its list item although this may be suppressed.

- If no arguments are given, **.LI** labels the items with the current *mark* which is specified by the more 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 this is formatted, it yields:

- This is a simple bullet item.

+ This replaces the bullet with a *plus*.

+ • This uses a *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 first argument of **.LI** is omitted or null, the resulting effect is that of a hanging indent. For instance, the first line of the following 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, the **.LE** outputs a blank line (one–half a vertical space). This option should generally be used only when the **.LE** is followed by running text but not when followed by a macro that produces blank lines of its own, such as the **.P**, **.H**, or **.LI** macros.

The **.H** and **.HU** macros automatically clear all list information. The user may omit the **.LE** macros that would normally occur just before either of these macros and not receive the **LE:mismatched** error message. Such a practice is not recommended because errors will occur if the list text is separated from the heading at some later time (such as by insertion of text).

Example of Nested Lists

An example of input for the several lists and the corresponding output is shown in the following example (the **.AL** and **.DL** macro calls contained therein are examples of list–initialization macros).
Input text is:

```
.AL A
.LI
This is alphabetized list item A.
```

```
This text shows the alignment of the
second line of the item.
Notice the text indentations and alignment of
left and right margins.
.AL
.LI
This is number 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 a prefix.
This text shows the alignment of the
second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized list item B.
This text shows the alignment of the
second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph follows a list item and is aligned with
the left margin.
A paragraph following a list resumes the normal line
length and margins.
```

The output is:

A. This is alphabetized list item A. This text shows the alignment of the second line of the item. Notice the text indentations and alignment of left and right margins.

    1. This is numbered item 1. This text hows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

```
      - This is a dash item.  This text shows the alignment
        of the second line of the item.  The quick brown fox
        jumped over the lazy dog's back.

    + - This is a dash item with a plus as prefix.  This text
        shows the alignment of the second line of the item.
        The quick brown fox jumped over the lazy dog's back.

  2.  This is numbered item 2.

B.  This is another alphabetized list item B.  This text shows the
    alignment of the second line of the item.  The quick brown fox
    jumped over the lazy dog's back.

This paragraph follows a list item and is aligned with the left
margin.  A paragraph following a list resumes the normal line length
and margins.
```

List–Begin Macro and Customized Lists

.LB *text-indent mark-indent pad type*
[*mark*] [*LI-space*] [*LB-space*]

List–initialization macros described in the preceding line suffice for almost all cases. However, if necessary, the user may obtain more control over the layout of lists by using the basic list–begin macro (**.LB**). The **.LB** macro is used by the other list–initialization macros. Its arguments are as follows:

- The *text-indent* argument provides the number of spaces that the 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 (for example, it ends *text-indent* spaces to the right of the current indent). The *mark-indent* argument is typically zero.
- Within the *mark area*, the mark is left justified if the *pad* argument is zero. If *pad* is a number *n* (greater than zero) then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. The *mark* is effectively right–justified *pad* spaces immediately to the left of text.
- Arguments *type* and *mark* interact to control the type of marking used. If *type* is zero, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than zero, automatic numbering or alphabetizing is done; *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing (it is chosen from the set (1, A, a, I, i) as shown in the section entitled *Automatically Numbered or Alphabetized Lists*). This is summarized as follows:

ARGUMENT type	RESULT mark	
0	omitted	hanging indent
0	string	*string* is the mark
>0	omitted	Arabic numbering
>0	one of;	automatic numbering or 1,A,a,I,i

Each nonzero value of *type* from one to six selects a different way of displaying the marks. The following table shows the output appearance for each value of *type*:

VALUE	APPEARANCE
1	*x.*
2	*x)*
3	*(x)*
4	*[x]*
5	<*x*>
6	{*x*}

where *x* is the generated number or letter.

NOTE

The mark must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified.

- 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 one; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the **.LI** macro issues a **.ne** request for two lines just before printing the mark.
- The *LB-space* argument is the number of blank lines (one–half a vertical space) to be output by **.LB** itself. If omitted, *LB-space* defaults to zero.

There are three combinations of *LI-space* and *LB-space*:

- The normal case is to set *LI-space* to one and *LB-space* to zero, yielding one blank line before each item in the list; such a list is usually terminated with a **.LE 1** macro to end the list with a blank line.
- For a more compact list, *LI-space* is set to zero, *LB-space* is set to one, and the **.LE 1** macro is used at the end of the list. The result is a list with one blank line before and after it. If both *LI-space* and *LB-space* are set to zero and the **.LE** macro is used to end the list, a list without any blank lines will result.

The following subsection 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 for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to define the appearance for each list level only once instead of having to define the appearance at the beginning of each list. This permits consistency of style in a large document. A generalized list-initialization macro might be defined in such a way that what the macro does depends on the list-nesting level in effect at the time the macro is called. Levels one through five of the lists to be formatted may have the following appearance:

A.

[1]

- a)

+

The following code defines a macro (**.aL**) that always begins a new list and determines the type of list according to the current list level. To understand it, the user should know that the number register **:g** is used by the MM list macros to determine the current list level; the level is zero if there is no current, active list. Each macro 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
.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** in the following text) 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, as follows:

```
.de bL
.ie \\n(.$.nr g \\$1          \"argument given, that is
                              the level
.el .nr g \\n(:g              \"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 g \\n(:g                  \"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 following example 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 Documents

One use of MM is for the preparation of memoranda and released–paper documents which have special requirements for the first page and for the cover sheet. Data needed (title, author, date, case numbers, etc.) is entered the same for both styles; an argument to the **.MT** macro indicates which style is being used.

Sequence of Beginning Macros

Macros, if present, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg]
.AT [title]...
.TM [number]...
.AS [arg] [indent]
one or more lines of abstract text
.AE
.NS [arg] [1]
one or more lines of "copy to" notation
.NE
.OK [keyword]...
.MT [type] [addressee]
```

The only required macros for a memorandum for file or a released-paper document are **.tl, .AU**, and **.MT**; all other macros (and their associated input lines) may be omitted if the features are not needed. Once **.MT** has been invoked, none of the above macros (except **.NS** and **.NE**) can be reinvoked because they are removed from the table of defined macros to save memory space.

If neither the memorandum nor released-paper style is desired, the **TL, AU, TM, AE, OK, MT, ND,** and **AF** macros should be omitted from the input text. If these macros are omitted, the first page will have only the page header followed by the body of the document.

NOTE

The macros for memorandum and released-paper documents require the postprocessor **col** *when you are using the* **nroff** *text formatter.*

Title

.TL [*charging-case*] [*filing-case*]
one or more lines of title text

Arguments to the **.TL** macro are the *charging-case* number(s) and *filing-case* number(s).

- The *charging-case* argument is the case number to which time was charged for the development of the project described in the memorandum. Multiple charging-case numbers are entered as *subarguments* by separating each from the previous with a comma and a space and enclosing the entire argument within double quotes.

- The *filing-case* argument is a number under which the memorandum is to be filed. Multiple filing case 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 may be used to break the title into several lines as follows:

```
.TL 12345
First Title Line
.br
\!.br
Second Title Line
```

On output, the title appears after the word *subject* in the memorandum style and is centered and printed in bold in the released-paper document style.

If only a charging-case number or only a filing-case number is given, it will be separated from the title in the memorandum style by a dash and will appear on the same line as the title. If both case numbers are given and are the same, then "Charging and Filing Case" followed by the number will appear on a line following the title. If the two case numbers are different, separate lines for "Charging-Case" and "Filing-Case" will appear after the title.

Authors

.AU *name* [*intials*] [*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 give. A separate **.AU** macro is required for each author.

The **.AT** macro is used to specify the author's title. Up to nine arguments may be given. Each will appear in the signature block for memorandum style on a separate line following the signer's name. The **.AT** must immediately follow the **.AU** for the given author. For example:

```
.AU "J.J.Jones" JJJ PY 9876 5432 1Z-234
.AT Director "Materials Research Laboratory"
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next line. The *x* for the extension is added automatically. Printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register **Au** to zero; the default value for **Au** is 1. Arguments seven through nine of the **.AU** macro, if present, will follow this normal author information in the "from" portion, each on a separate line. These last three arguments may be used for organizational number 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 its location are centered. The paragraph on memorandum types contains information regarding authors from different location.

TM Numbers

.TM [*number*] . . .

If the memorandum is a technical memorandum, the TM numbers are supplied via the **.TM** macro. Up to nine numbers may be specified. For example:

.TM 7654321 77777777

This macro call is ignored in the released-paper and external-letter styles.

Abstract

.AS [*arg*] [*indent*]
text of abstract
.AE

If a memorandum has an abstract, the input is identified with the **.AS** (abstract start) and **.AE** (abstract end) delimiters. Abstracts are printed on page one of a document and/or on its cover sheet. There are three styles of cover sheets:

- 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 (first argument of the **.MT** macro is 4) and in technical memorandum style if the first argument of **.AS** is:

0 - Abstract will be printed on page one and on the cover sheet (if any).

1 - Abstract will appear only on the cover sheet (if any).

In memoranda for file style and in all other documents (other than external letters) if the first argument of **.AS** is:

0 - Abstract will appear on page one and there will be no cover sheet printed.

2 - Abstract will appear only on the cover sheet which will be produced automatically (for example, without invoking the **.CS** macro).

It is not possible to get either an abstract or a cover sheet with an external letter (first argument of the **.MT** macro is 5).

Notations such as a "copy to" list are allowed on memorandum for file cover sheets; the **.NS** and **.NE** macros must appear after the **.AS 2** and **.AE** macros. Headings and displays are not permitted within an abstract.

The abstract is printed with ordinary text margins; an indentation to be used for both margins can be specified as the second argument of **.AS**. Values that specify indentation must be unscaled and are treated as "character positions" (such 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 may 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 of a released-paper document and the format of the cover sheets. The *type* arguments and corresponding values are as follows:

type	VALUE
""	no memorandum type printed
0	no memorandum type printed
none	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 the *type* argument indicates a memorandum style document, the corresponding statement indicated under **VALUE** above will be printed after the last line of author information. If *type* is longer than one character, then the string, itself, will be printed. For example:

.MT "Technical Note #5''

A simple letter is produced by calling **.MT** with a null (but not omitted) or 0 argument.

The second argument to **.MT** is the name of the addressee of a letter. If present, that name and the page number replace the normal page header on the second and following pages of a letter. For example:

.MT 1 "John Jones"

produces

John Jones - 2

The *addressee* argument may not be used if the first argument is 4 (released–paper style document).

The released–paper style is obtained by specifying

.MT 4 [1]

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following the company name. If the optional second argument to **.MT 4** is given, then the name of each author is followed by the respective company name and location. Information necessary for the memorandum style document but not for the released–paper style document is ignored.

If the released–paper style document is utilized, the macros for the end of a memorandum and their associated lines of input are likewise ignored. Authors may include their affiliations in the released–paper style by specifying the appropriate **.AF** macro and defining a string (with a two–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 "Tektronix, Inc."
.AU "Same 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

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 the default company name without affecting other headings. If the argument is null, the default company name is suppressed; extra blank lines are inserted to allow room for stamping the document with a logo or a company stamp.

The **.AF** with no arguments suppresses the default company name 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 **—rO***K* and **—rW***k* are not effective with **.AF**. The only **.AF** use appropriate for the **troff** formatter is to specify a replacement for the default company name.

The command line option**—rE***n* controls the font of the *Subject/Date/From* block.

Example

Input text for a document may begin as follows:

```
.TL
MM\*(EMMemorandum Macros
.AU "D.W. Smith" DWS PY
.AU "J.R. Mashey" JRM PY
.AU "E.C. Pariser (January 1980 Revision)" ECP PY
.AU "N.W. Smith (June 1980 Revision)" NWS PY
.MT 4
```

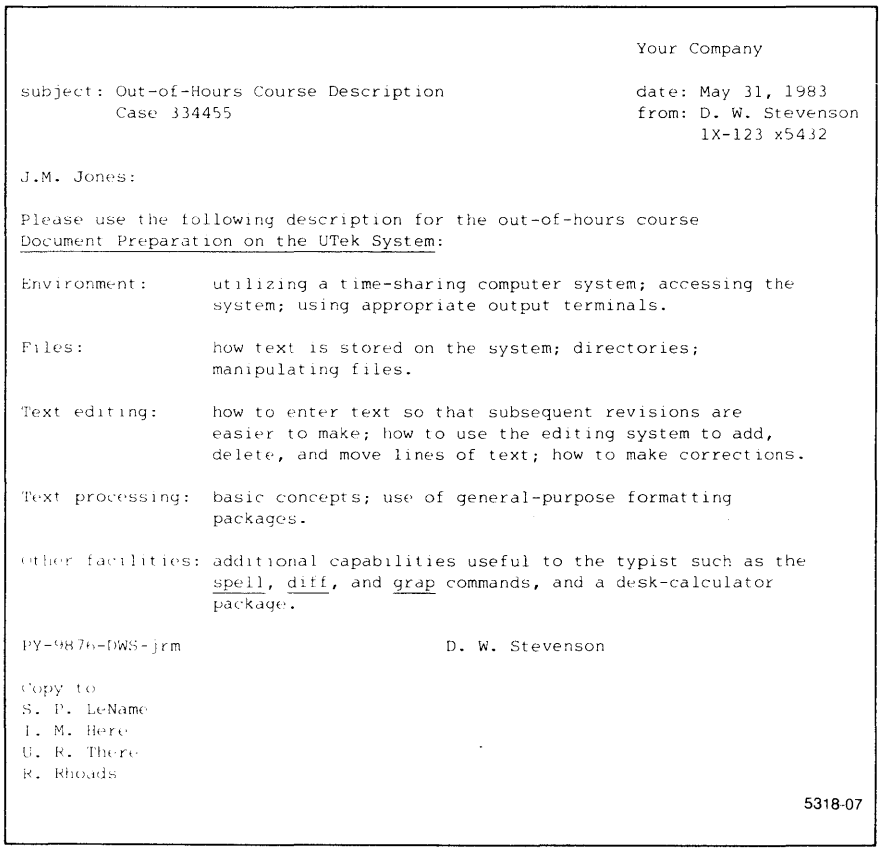
Figures 4D–1, 4D–2, and 4D–3 show the input text file and both the **nroff** and **troff** formatter outputs for a single letter.

```
.ND "May 31, 1983"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.AF "Your Company"
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the
out-of-hours course
.I
Document Preparation on the UTek*
.R
.FS *
Trademark of Tektronix, Inc.
.FE
.I "System:"
.P
The course is intended for clerks, typists, and others
who intend to use the UTek  system for preparing
documentation.
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system; using appropriate output terminals.
.LI Files:
how text is stored on the system;  directories;
manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions
are easier to make;
how to use the editing system to add, delete,
and move lines of text;
how to make corrections.
```

Figure 4D–1. Example of Input for a Simple Letter.

```
.LI "Text processing:"
basic concepts;  use of general purpose
formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist
such as the \fIspell\fR,
\fIdiff\fR, and \fIgrep\fR commands,
and a desk-calculator package.
.LE
.SG jrm
.NS 0
S. P. LeName
I. M. Here
U. R. There
R. Rhoade
.NE
```

Figure 4D-1 (cont). Example of Input for a Simple Letter.



```
                                                        Your Company

subject: Out-of-Hours Course Description                date: May 31, 1983
         Case 334455                                    from: D. W. Stevenson
                                                              1X-123 x5432

J.M. Jones:

Please use the following description for the out-of-hours course
Document Preparation on the UTek System:

Environment:        utilizing a time-sharing computer system; accessing the
                    system; using appropriate output terminals.

Files:              how text is stored on the system; directories;
                    manipulating files.

Text editing:       how to enter text so that subsequent revisions are
                    easier to make; how to use the editing system to add,
                    delete, and move lines of text; how to make corrections.

Text processing:    basic concepts; use of general-purpose formatting
                    packages.

Other facilities:   additional capabilities useful to the typist such as the
                    spell, diff, and grap commands, and a desk-calculator
                    package.

PY-9876-DWS-jrm                             D. W. Stevenson

Copy to
S. P. LeName
I. M. Here
U. R. There
R. Rhoads
```

5318-07

Figure 4D–2. Example of Nroff Output for a Simple Letter.

Your Company

subject: **Out-of-Hours Description**
Case 334455

date: **May 31, 1983**
from: **D.W. Stevenson**
PY 9876
1X-123 x5432

J.M. Jones:

Please use the following description for the out-of-hours course *Document Preparation on the UTek System*:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the *spell*, *diff*, and *grep* commands, and a desk-calculator package.

PY-9876-DWS-jrm

D.W. Stevenson

Copy to
S.P. LeName
I.M. Here
U.R. There
R. Rhoade

5318-08

Figure 4D-3. Example of Troff Output for a Simple Letter.

End of Memorandum Macros

At the end of a memorandum document (but not of a released–paper document), signatures of authors and a list of notations can be requested. The following macros and their input are ignored if the released–paper style document is selected.

Signature Block

.FC [*closing*]
.SG [*arg*] [1]

The **.FC** macro prints "Yours very truly," as a formal closing, if no closing argument is used. It must be given before the **.SG** macro. A different closing may be specified as an argument to **.FC**.

The **.SG** macro prints the author's name(s) after the formal closing, if any. Each name begins at the center of the page. Three blank lines are left above each name for the actual signature.

- If no arguments are given, the line of reference date (location code, department number, author's initials, and typist's initials, all separated by hyphens) will not appear.
- A nonnull first argument is treated as the typist's initials and is appended to the reference date.
- A null first argument prints reference data without the typist's initials or the preceding hyphen.
- If there are several authors and if the second argument is given, reference date is placed on the line of the first author.

Reference date 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 date should be supplied manually after the invocation (without arguments) of the **.SG** macro.
For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

Copy to and Other Notations

.NS [*arg*] [1]
zero or more lines of the notation
.NE

Many types of notations (such as a list of attachments or "Copy to" lists) may follow signature and reference data. Various notations are obtained through the **.NS** macro, which provides for proper spacing and for breaking notations across pages, if necessary.

The optional second argument, if present, causes the first argument to be used as the *entire* notation string. Codes for *arg* and the corresponding notations are as follows:

arg	**NOTATIONS**
none	Copy to
""	Copy to
0	Copy to
1	Copy (with att.) to
2	Copy (without att.) to
3	Att.
4	Atts.
5	Enc.
6	Encs.
7	Under Separate Cover
8	Letter to
9	Memorandum to
10	Copy (with atts.) to
11	Copy (without atts.) to
12	Abstract Only to
13	Complete Memorandum to
"string"	Copy (string) to
"string",with 2nd *arg*	string

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to".

For example:

.NS "with att. 1 only"

will generate

Copy (with att. 1 only) to

as the notation.

More than one notation may be specified before the **.NE** macro because a **.NS** macro terminates the preceding notation, if any.
For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J.J. Jones
```

```
.NS 2
S.P. LeName
G.H. Hurtz
.NE
```

would be formatted as the following text demonstrates:

```
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
```

If the second argument is used, then the first argument becomes the entire notation.

```
For example:
```

.NS "Table of Contents to" 1

would be formatted in the following way:

Table of Contents to

The **.NS** and **.NE** macros may also be used at the beginning following **.AS 2** and **.AE** to place the notation list on the memorandum for file cover sheet. If notations are given at the beginning without **.AS 2**, they will be saved and output at the end of the document.

Approval Signature Line

.AV *approver's-name* [1]

The **.AV** macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example,

.AV ``*Jane Doe*''

produces this format:

APPROVED:

_____ _____

Jane Doe Date

The optional second argument, if present, prevents the "APPROVED:" mark from appearing above the approval line.

One-Page Letter

At times, the user may like more space on the page forcing the signature or items within notations to the bottom of the page so that the letter or memo is only one page in length. This can be accomplished by increasing the page length with the *—rLn* option (for example, *—rL90*). This has the effect of making the formatter believe that the page is 90 lines long; it therefore provides more space than usual to place the signature or the notations.

NOTE

This will work only for a single-page letter or memo.

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 display is maintained so that their relative order of appearance in the text is not disturbed.

By default, a display is processed in no-fill mode with single spacing and is not indented from the existing margins. The user can specify indentation or centering as well as fill-mode processing.

NOTE

Displays and footnotes may never be nested in any combination. Although lists and paragraphs are permitted, no headings (**.H** or **.HU**) *can occur within displays or footnotes.*

Static Displays

.DS [*format*] [*fill*] [*r-indent*]
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 node
omitted	no-fill mode

- The *r-indent* argument is the number of characters that the line length should be decreased (such as an indentation from the right margin). This number must be unscaled in the **nroff** formatter and is treated as *ens*. It may be scaled in the **troff** formatter or else defaults to *ems*.

The standard amount of static display indentation is taken from the **Si** register, a default value of five spaces. Thus, text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the **Pi** register. Even though their initial values are the same (default values), these two registers are independent.

The display *format* argument value 3 (or CB) centers (horizontally) the entire display as a block (as opposed to **.DS 2** and **.DF 2** which center each line individually). All collected lines are left justified, and the display is centered based on width of the longest line. This format must be used in order for the **neqn** *mark* and *lineup* features to work with centered equations.

By default, a blank line (one-half a vertical space) is placed before and after static and floating displays. These blank lines before and after static displays can be inhibited by setting the register **Ds** to zero.

The following example shows usage of all three arguments for static display. This block of text will be indented five spaces (ems in **troff**) from the left margin, filled, and indented five spaces (ems in **troff**) from the right margin (centered, for example). 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 to the
United States of America."
```

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 the Constitution to the United States of America."

Floating Displays

.DF [*format*] [*fill*] [*r-indent*]
one or more lines of text
.DE

A floating display is started by the **.DF** macro and terminated by the **.DE** macro. Argument have the same meanings as static displays described above, except indent, no indent, and centering are calculated with respect to the initial left margin. This is because prevailing indent may change between when the formatter first reads the floating display and when the display is printed. One blank line (one-half a vertical space) occurs before and after a floating display.

The user may exercise precise control over the output positioning of floating displays through the use of two number registers, **De** and **Df** (see the following explanation). When a floating display is encountered by the **nroff** or **troff** formatter, it is processed and placed onto a queue of displays waiting to be output. Displays are removed from the queue and printed in the order entered, which is the order they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, the new display is a candidate for immediate output on the current page. Immediate output is governed by size of display and the setting of the **Df** register code. The **De** register code controls whether text will appear on the current page after a float display has been produced.

As long as the display queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in two–column mode), the next display from the queue becomes a candidate for output if the **Df** register code has specified "top–of–page" output. When a display is output, it is also removed from the queue.

When the end of a section (using section–page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This occurs before a **.SG, .CS**, or **.TC** macro is processed.

A display will fit on the current page if there is enough room to contain the entire display or if the display is longer than one page in length and less than half of the current page has been used. A wide (full–page width) display will not fit in the second column of a two–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	A page eject will always follow the output of each floating display, so only one floating display will appear on a page and no text will follow it.

NOTE

For any other code, the action performed is the same as for code 1.

Df REGISTER

CODE	ACTION
0	Floating displays will not be output until end of section (when section–page numbering) or end of document.
1	Output new floating display on current page if there is space; otherwise, hold it until end of section or document.
2	Output exactly one floating display from queue to the top of a new page or column (when in two–column mode).
3	Output one floating display on current page if there is space; otherwise, output to the top of a new page or column.
4	Output as many displays as will fit (at least one) starting at the top of a new page or column. If the **De** register is set to 1, each display will be followed by a page eject causing a new top of page to be reached where at least one more display will be output.

5 Output a new floating display on the current page if there is room (default condition). Output as many displays (but at least one) as will fit on the page starting at the top of a new page or column. If the **De** register is set to 1, each display will be followed by a page eject causing a new top of page to be reached where at least one more display will be output.

NOTE

For any code greater than 5, the action performed is the same as for code 5.

The **.WC** macro may also be used to control handling of displays in double-column mode and to control the break in text before floating displays.

Tables

.TS [*H*]
global options;
column descriptors.
title lines
[**.TH** [*N*]
data within the table.
.TE

The **.TS** (table start) and **.TE** (table end) macros make possible the use of the **tbl** program. These macros are used to delimit text to be examined by **tbl** and to set proper spacing around the table. The display function and the **tbl** delimiting function are independent. In order to permit the user to keep together blocks that contain any mixture of tables, equations, filled text, unfilled text, and caption lines, the **.TS/.TE** block should be enclosed within a display (**.DS/.DE**). Floating tables may be enclosed inside floating displays (**.DF/.DE**).

Macros **.TS** and **.TE** permit processing of tables that extend over several pages. If a table heading is needed for each page of a multi-page table, the *H* argument should be specified to the **.TS** macro as in the opening synopsis. Following the options and format information, table title is typed on as many lines are required and is followed by the **.TH** macro. The **.TH** macro must occur when "**.TH H**" is used for a multi-page table. This is not a feature of **tbl** but of the definitions provided by the MM macro package.

The **.TH** (table header) macro may take as an argument the letter *N*. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller **.TS H/.TE** segments.

For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

This input will cause the table heading to appear at the top of the first table segment and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (such as one with too many blocks of filled text). If each segment had its own **.TS H/.TH** sequence, it would have its own header. However, if each table segment after the first uses **.TS H/.TH N**, the table header will appear only at the beginning of the table and the top of each new page or column that the table continues onto.

For the **nroff** formatter, the **—e** option (**—E** for **mm**) may be used for terminals, such as the 450, that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. The **—e** is not effective with **col**.

Figure, Table, Equation, and Exhibit Titles

.FG [*title*] [*override*] [*option*]
.TB [*title*] [*override*] [*option*]
.EC [*title*] [*override*] [*option*]
.EX [*title*] [*override*] [*option*]

The **.FG** (figure title), **.TB** (table title), **.EC** (equation caption), and **.EX** (exhibit caption) macros are normally used inside **.DS** pairs to automatically number and title figures, tables, and equations. These macros use registers **Tg, Tb, Ec,** and **Ex**, respectively (see the earlier section on **—rN5** to reset counters in sections).
For example:

.FG "This is a Figure Title"

yields

Figure 1. This is a Figure Title

The **.TB** macro replaces *Figure* with *TABLE*, the **.EC** macro replaces *Figure* with *Equation*, and the **.EX** macro replaces *Figure* with *Exhibit*. The output title is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the **.af** request of the formatter. By setting the **Of** register to 1, the format of the caption may be changed from

Figure 1. Title

to

Figure 1 — Title

The *override* argument is used to modify normal numbering. If the *option* argument is omitted or is 0, *override* is used as a prefix to the number; if the *option* argument is 1, *override* is used as a suffix; and if the *option* argument is 2, *override* replaces the number. If **—rN5** is given, "section–figure" numbering is set automatically and user–specified *override* argument 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 the number registers **Lf, Lt, Lx,** and **Le** (respectively) are set to 1. The **Lf, Lt,** and **Lx** registers are 1 by default; **Le** is 0 by default.

Titles of these lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES
.ds Lt LIST OF TABLES
.ds Lx LIST OF EXHIBITS
.ds Le LIST OF EQUATIONS
```

Footnotes

There are two macros (**.FS** and **.FE**) that delimit text of footnotes, a string (**F**) 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 Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "**F**" (invoking the string *F*) immediately after the text to be footnoted without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half line above the text to be footnoted.

Delimiting Footnote Text

.FS [*label*]
one or more lines of footnote text
.FE

There are two macros that delimit the text of each footnote. The **.FS** (footnote start) macro marks the beginning of footnote text, and the **.FE** (footnote end) macro marks the end. The *label* on the footnote start macro, if present, will be used to mark footnote text. Otherwise, the number retrieved from the string *F* will be used. Automatically numbered and user–labeled footnotes may be intermixed. If a footnote is labeled (**.FS** *label*), the text to be footnoted must be followed by *label*, rather than by "**F**". Text between **.FS** and **.FE** is processed in fill mode. Another **.FS**, a **.DS**, or a **.DF** is not permitted between the footnote start and end 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.
```

Labeled footnote:

```
This is a labeled*
.FS*
The footnote is labeled with an asterisk
footnote.
```

Text of the footnote (enclosed within the **.FS**/**.FE** pair) should immediately follow the word to be footnoted in the input text, so that "**F**" or *label* occurs at the end of a line of input and the next line is the **.FS** macro call. It is also good practice to append an unpaddable space to "**F**" or *label* when they follow an end–of–sentence punctuation mark (such as a period, question mark, exclamation point).

Format Style of Footnote Text

.FD *[arg] [1]*

Within footnote text, the user can control formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left or right justification of the label when text indenting is used. The **.FD** macro is invoked to select the appropriate style.

The first argument (*arg*) is a number from the left column of Table 4D–3. Formatting style for each number is indicated in the remaining four columns. Further explanation of the first two of these columns is given in the definitions of the **.ad, .na, .hy,** and **.nh** requests (adjust, no adjust, hyphenation, and no hyphenation, respectively).

Table 4D–3
FORMAT STYLE OF FOOTNOTE TEXT

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 first argument to **.FD** is greater than 11, the effect is as if **.FD 0** were specified. If the first argument is omitted or null, the effect is equivalent to **.FD 10** in the **nroff** formatter and to **.FD 0** in the **troff** formatter; these are also the respective initial default values.

If the second argument is specified, then when a first–level heading is encountered, automatically numbered footnotes begin again with 1. This is most useful with the "section–page" page numbering scheme. As an example, the input line

```
.FD "" "" 1
```

maintains the default formatting style and causes footnotes to be numbered afresh after each first–level heading in a document.

Hyphenation across pages is inhibited by MM except for long footnotes that continue to the following page. If hyphenation is permitted, it is possible for the last word on the last line on the current page footnote to be hyphenated. The user has control over this situation by specifying an even **.FD** argument.

Footnotes are separated from the body of the text by a short line rule. Those that continue to the next page are separated from the body of the text by a full-width rule. In the **troff** formatter, footnotes are set in type two points smaller than the point size used in the body of text.

Spacing Between Footnote Entries

Normally, one blank line (a three-point vertical space) separates footnotes when more than one occurs on a page. To change this spacing, the **Fs** number register is set to the desired value. For example:

.nr Fs 2

will cause two blank lines (a six-point vertical space) to occur between footnotes.

Figure 4D-4 shows input for a number of footnote styles:

```
.FD 10
.P
This example illustrates several footnote styles
for both labeled and automatically numbered footnotes.
With the footnote style set to the \fBnroff\fR
default style,
the first footnote is processed \*F
.FS
This is the first footnote text example.
This is the default style (.FD 10) for the \fBnroff\fR
formatter.
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 followed by a second footnote.*****
.FS *****
This is the second footnote text example.
This is also the \fBnroff\fR formatter
default style (.FD 10)
but with a long footnote label (*****)
by the user.
.FE
.FD 1
```

Figure 4D-4. Example of Input for Various Footnote Styles.

```
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 \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
Footnote style is set again via the .FD macro
for no hyphenation,
no right margin justification,
no indentation, and with the label left justified.
This produces the fifth footnote. \*F
.FS
This is the fifth footnote example (.FD 6).
The right margin is not justified,
hyphenation is not permitted,
footnote text is not indented,
and the label is placed at the beginning of the first line.
.FE
```

Figure 4D-4 (cont). Example of Input for Various Footnote Styles.

The results of this input are shown on this page and the next page. The output illustrates several footnote styles for both labeled and automatically numbered footnotes. With the footnote style set to the **nroff** default style, the first footnote is processed[1] and followed by the second footnote.***** Footnote style is changed by using the **.FD** macro

1. This is the first footnote text example. This is the default style (.FD 10) for the **nroff** formatter. The right margin is not justified, hyphenation is not permitted, text is indented, and the automatically generated label is right justified in the text-indent space.

***** This is the second footnote text example. This is also the **nroff** formatter default style (.FD 10) but with a long footnote label (*****) provided by the user.

to specify hyphenation, right margin justification, indentation, and left justification of the label. This produces the third footnote, [2] and then the fourth footnote.† Footnote style is set again via the **.FD** macro for no hyphenation, no right margin justification, no indentation, and with the label left justified. This produces the fifth footnote.[3]

Page Headers and Footers

Text printed at the top of each page is called *page header*. Text printed at the bottom of each page is called *page footer*. There can be up to three lines of text associated with the header — every page, even page only, and odd page only. Thus, the page header may have up to two lines of text — the line that occurs at the top of every page and the line for the even-numbered or odd-numbered page. The same is true for the page footer.

This part describes the default appearance of page headers and page footers and ways of changing them. The term *header* (not qualified by *even* or *odd*) is used to mean the page header line that occurs on every page, and similarly for the term *footer*.

Default Headers and Footers

By default, each page has a centered page number as the header. There is no default footer and no even/odd default headers or footers except as specified later in this subsection.

In a memorandum or a released-paper style document, the page header on the first page is automatically suppressed provided a break does not occur before the **.MT** macro is called. Macros and text in the following categories do not cause a break and are permitted before the memorandum type (**.MT**) macro:

- Memorandum and released-paper style document macros (**.TL, .AU, .AT, .TM, .AS, .AE, .OK, .ND, .AF, .NS,** and **.NE**)
- Page headers and footers macros (**.PH, .EH, .OH, .PF, .EF,** and **.OF**)
- The **.nr** and **.ds** requests.

---

2. This is the third footnote example (.FD 1). This right margin is justified, the footnote text is indented, and the label is left justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). This 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.

Header and Footer Macros

For the header and footer macros (**.PH, .EH, .OH, .PH, .EF,** and **.OF**) the argument [*arg*] is of the form:

```
"'left-initials-part'right-part' "
```

If it is inconvenient to use an apostrophe (') as the delimiter because it occurs within one of the parts, it may be replaced uniformly by any other character. The **.fc** request redefines the delimiter. 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 may be changed by the **.af** macro request.

If *debug mode* is set using the option **—rD1** 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 the same as the **.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 **—rC***n* option 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 supplies a line to be printed at the bottom of each odd-numbered page immediately preceding the footer. Initial value is a blank line.

First Page Footer

By default, the first page footer is a blank line. If, in the input text file, the user specifies **.PF** and/or **.OF** before the end of the first page of the document, these lines will appear at the bottom of the first page.

The header (whatever its contents) replaces the footer on the first page only if the **—rN1** option 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 (it may be *B-2*, for example); the default page header is blank.

Strings and Registers in Header and Footer Macros

String and register names may be placed in arguments to header and footer macros. If the value of the string or register is to be computed when the respective header or footer is printed, invocation must be escaped by four backslashes. This is because string or register invocation will be processed three times:

1. As the argument to the header or footer macro,
2. In a formatting request within the header or footer macro, or
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, the user specifies the following:

```
.PF '' ' ' '- \\\\n(H1-\\\\nP -' ''
```

If the user arranges for the string *a]* to contain the current section heading which is to be printed at the bottom of each page, the **.PF** macro call would appear as such:

```
.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 (for example, the top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page (nothing is specified for the center):

```
.PH ""
.PF ""
.EH '' '\\\\nP' 'Revision 3' ''
.OH '' 'Revision 3' '\\\\nP' ''
```

Generalized Top-of-Page Processing

NOTE

This part is intended only for users accustomed to writing formatter macros.

During header processing, MM invokes two user-definable macros:

- The **.TP** (top of page) macro is invoked in the environment (refer to **.ev** request) of the header.
- The **.PX** is a page header user-exit macro that is invoked (without arguments) when the normal environment has been restored and with the "no-space" mode already in effect.

The effective initial definition of **.TP** (after the first page of a document) is as follows:

```
.de TP
.sp 3
.tl \\*(}t
.if e 'tl \\*(}e
.if o 'tl \\*(}o
.sp 2
..
```

The string *}t* contains the header, the string *}e* contains the even–page header, and the string *}o* contains the odd–page header as defined by the **.PH, .EH,** and **.OH** macros, respectively. To obtain more specialized page titles, the user may redefine the **.TP** macro to cause the desired header processing. Formatting done within the **.TP** macro is processed in an environment different from that of the body. For example, to obtain a page header that includes three centered lines of data (such as document number, issue date, and revision date), the user could define the **.TP** call as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The **.PX** macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns to 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 will be printed at the bottom of each page after the footnotes (if any) but before the page footer. This block of text is removed by specifying an empty block in the following way:

.BS
.BE

The bottom block will appear on the table of contents, pages, and the cover sheet for memorandum for file, but not on the technical memorandum or released–paper cover sheets.

Top and Bottom (Vertical) Margins

.VM [*top*] [*bottom*]

The **.VM** (vertical margin) macro allows the user to specify additional space at the top and bottom of the page. This space precedes the page header and follows the page footer. The **.VM** macro takes two unscaled arguments that are treated as *v*'s. For example:

```
.VM 10 15
```

This adds 10 blank lines to the default top of page margin and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by redefining **.TP**).

Proprietary Marking

.PM [*code*]

The **.PM** (proprietary marking) macro appends to the page footer a proprietary disclaimer. The *code* argument may be any one of those listed here:

code	**DISCLAIMER**
none	turn off previous disclaimer, if any
P	PRIVATE
N	NOTICE
BP	TEK PRIVATE
BB (or BR)	TEK PROPRIETARY–PRIVATE
BPN	TEK–NOTICE
ILL	"RENDERED ILLEGIBLE" message
CI–II	Computer Inquiry II message

These disclaimers are in a form approved for use by Tektronix, Inc. The user may alternate disclaimers by use of the **.BS/.BE** macro pair. Markings are underlined (italic in **troff**). The *CI-II* marking may be used with any other message by two separate **.PM** requests. For example:

```
.PM CI–II
.PM N
```

produces a *CI-II* and *NOTICE* mark.

Private Documents

.nr Pv *value*

The word *PRIVATE* may be printed, centered, and underlined on the second line of a document (preceding the page header). This is done by setting the **Pv** register value:

value	**MEANING**
0	do not print PRIVATE (default)
1	PRIVATE on the first page only
2	PRIVATE on all pages

If *value* is 2, the user-definable **.TP** macro may not be used because the **.TP** macro is used by MM to print *PRIVATE* on all pages except the first page of a memorandum (on which **.TP** is not invoked).

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 because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet may not be desired by a user and is therefore produced at the end.

Table of Contents

.TC [*s-level*] [*spacing*] [*t-level*] [*tab*]
[*h1*] [*h2*] [*h3*] [*h4*] [*h5*]

The **.TC** macro generates a table of contents containing heading levels that were saved for the table of contents as determined by the value of the **Cl** register. Arguments to **.TC** control spacing before each entry, placement of associated page number, and additional text on the first page of the table of contents before the word *CONTENTS*.

Spacing before each entry is controlled by the first and second arguments (*s-level* and *spacing*). Headings whose level is less than or equal to *s-level* will have *spacing* blank lines (halves of a vertical space) before them. Both *s-level* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (one-half a vertical space). The *s-level* argument does *not* control what levels of heading have been saved; saving of headings is the function of the **CI** register.

The third and fourth arguments (*t-level* and *tab*) control placement of associated page number for each heading. Page numbers can be justified at the right margin with either blanks or dots (called leaders) separating the heading text from the page number; an alternative way would be with the page numbers following the heading text.

- For headings whose level is less than or equal to *t-level* (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, as leaders, are used. If *tab* is greater than 0, spaces are used.
- For headings whose level is greater than *t-level*, page numbers are separated from heading text by two spaces (such that page numbers are "ragged right", not right justified).

Additional arguments (*h1* . . . . *h5*) are horizontally centered on the page and precede the table of contents.

If the **.TC** macro is invoked with at most four arguments, the user-exit macro **.TX** is invoked (without arguments) before the word *CONTENTS* is printed. The user-exit macro **.TY** may be invoked instead, causing the word *CONTENTS* to not be printed.

By defining **.TX** or **.TY** and invoking **.TC** with at most four arguments, the user can specify what needs to be done at the top of the first page of the table of contents.

For example:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +5n
Approved:  \l'2.5i'
.in
.sp
..
.TC
```

```
                                        Special Application
                                Message Transmission

        Approved:_____________________________________________

                                        CONTENTS
yields the following output when the file is formatted:
                                                .
                                                .
```

If the **TX** macro were defined as **.TY**, the word *CONTENTS* would be suppressed. Defining **.TY** as an empty macro will suppress *CONTENTS* with no replacement:

```
.de TY
..
```

By default, the first level headings will appear in the table of contents left justified. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the *Ci* string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the **Cl** register. Arguments must be scaled.

For example, with "Cl equal to 5":

```
.ds Ci .25i .5i .75i 1i 1i          \"troff
```

or

```
.ds Ci 0                            \"nroff
```

Two other registers are available to modify the format of the table of contents. These are **Oc** and **Cp**.

- By default, table of contents pages will have lowercase Roman numeral page numbering. If the **Oc** register is set to 1, the **.TC** macro will not print any page number but will instead reset the **P** register to 1. It is the user's responsibility to give an appropriate page footer to specify the placement of the page number. Ordinarily, the same **.PF** macro (page footer) used in the body of the document will be adequate.
- The list of figures, tables, and other such pages will be produced separately unless **Cp** is set to 1, which causes these lists to appear on the same page as the table of contents.

Cover Sheet

.CS [*pages*] [*other*] [*total*] [*figs*] [*tbls*] [*refs*]

The **.CS** macro generates a cover sheet in either the released paper or technical memorandum style (see the earlier subsection, *Abstract*, for details of the memorandum for file cover sheet). All other information for the cover sheet is obtained from data given before the **.MT** macro call. If the technical memorandum style is used, the **.CS** macro generates the *Cover Sheet for Technical Memorandum*. The data that appear in the lower left corner of the technical memorandum cover sheet (counts of: pages of text, other pages, total pages, tables, figures, and references) are generated automatically.

NOTE

0 is used for "other pages".

These values may be changed by supplying the corresponding arguments to the **.CS** macro. If the released-paper style is used, all arguments to **.CS** are ignored.

References

There are two macros (**.RS** and **.RF**) that delimit the text of references, a string that automatically numbers the sebsequent references, and an optional macro (**.RP**) that produces reference pages within the document.

Automatic Numbering of References

Automatically numbered references may 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. The number register actually used to print the reference number for each reference call ***(Rf** in the text is **:R**. The **:R** register may have its format or value changed to affect the reference marks, without affecting the total count of references.

Delimiting Reference Text

.RS [*string-name*]
.RF

The **.RS** and **.RF** macros are used to delimit text of each reference as shown in the following example:

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 may be used later in the document as the string call, *(*aA*, to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets one–half line above the text to be referenced. No **.RS/.RF** pair is needed for subsequent references.

Reference Page

.RP [*arg1*] [*arg2*]

A reference page, entitled by default *REFERENCES*, will be generated automatically at the end of the document (before the table of contents and cover sheet) and will be listed in the table of contents. This page contains the reference items (the reference text enclosed within the **.RS/.RF** pairs). Reference items will be separated by a space (one–half a vertical space) unless the **Ls** register is set to 0 to suppress this spacing. The user may change the reference page title by defining the *Rp* string:

.ds Rp *"New Title"*

The **.RP** (reference page) macro may be used to produce reference pages anywhere else within a document. For instance, the user may want reference pages after each major section. It is not needed to produce a separate reference page with default spacings at the end of the document.

Two **.RP** macro arguments allow the user to control resetting of reference numbering and page skipping.

arg1	**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** macro is issued by the **.RP** macro, a single blank line will separate the reference from the following/preceding text. The user may wish to adjust spacing. For example, to produce references at the end of each major section, the following input is introduced:

.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, The **.B** macro changes the font to bold and the **.I** macro changes to underlining (italic). This condition continues until the occurrence of the **.R** macro which cause the Roman font to be restored.
Thus, the following yields underlined text via the **nroff** and italic text via the **troff** formatter:

.I
here is some text.
.R

If the **.B** or **.I** macro is called with one argument, that argument is printed in the appropriate font (underlined in the **nroff** formatter for **.I**). The previous font is restored (and underlining is turned off in the **nroff** formatter). If two or more arguments (with a maximum of six) are given with a **.B** or **.I** macro call, the second argument is concatenated to the first with no intervening space (1/12 space if the first font is italic) but is printed in the previous font. Remaining pairs of arguments are similarly alternated. For example:

```
.I italic  "<sp>text<sp>" right-justified
```
(*<sp>* indicates a space)

produces

italic `text right-justified`

The **.B** and **.I** macros alternate with the prevailing font at the time the macros are invoked. To alternate specific pairs of fonts, the following macros are available:

.IB .BI .IR .RI .RB .BR

Each macro takes a maximum of six arguments and alternates arguments between specified fonts.

When using a terminal that cannot underline, the following can be inserted at the beginning of the document to eliminate all underlining:

.rm ul
.rm cu

NOTE
Font changes in headings are handled separately.

Justification of Right Margin

.SA [*arg*]

The **.SA** macro is used to set right–margin justification for the main body of text. Two justification options are used, these being *current* and *default*. Initially, both options are set for no justification in the **nroff** formatter and for justification in the **troff** formatter. The argument causes the following action:

arg	**MEANING**
0	Sets both options to no justification. It acts like the **.na** request.
1	Sets both options to cause both right and left justification, the same as the **.ad** request.
omitted	Causes the current option to be copied from the default flag, thus performing either a **.na** or **.ad** depending on the default condition.

In general, the no adjust request (**.na**) can be used to ensure that justification is turned off, but **.SA** should be used to restore justification, rather than the **.ad** request. In this way, justification or no justification for the remainder of the text is specified by inserting **.SA 0** or **.SA 1** once at the beginning of the document.

SCCS Release Identification

The *RE* string contains the SCCS release and the MM text formatting macro package current version level. For example:

```
This is version \*(RE of the macros.
```

produces

```
This is version 10.129 of the macros.
```

This information is useful in analyzing suspected bugs in MM. The easiest way to have the release identification number appear in the output is to specify **—rD1** 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 two-column processing which continues until a **.1C** macro (one-column processing) is encountered. In two-column processing, each physical page is thought of as containing two-column "pages" of equal (but smaller) "page" width. Page headers and footers are not affected by two-column processing. The **.2C** macro does not balance two-column output.

Footnotes and Displays for Two-Column Output

It is possible to have full-page width footnotes and displays when in two-column mode, although default action is for footnotes and displays to be narrow in two-column mode and wide in one-column mode. Footnote and display width is controlled by the **.WC** (width control) macro, which takes the following arguments:

arg	**MEANING**
N	Default mode (*—WF, -FF, -WD, FB*).
WF	Wide footnotes (even in two-column mode).
—WF	DEFAULT: Turn off *WF*. Footnotes follow column mode; wide in one-column mode (1C), narrow in two-column mode (2C), unless *FF* is set.
FF	First Footnote. All footnotes have same width as first footnote encounteded for that page.
—FF	DEFAULT: Turn off *FF*. Footnote style follows settings of *WF* or *-WF*.
WD	Wide displays (even in two-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 FF** *command will cause all displays to be wide and all footnotes on a page to be the same width, while* **.WC N** *will reinstate default actions. If conflicting settings are given to* **.WC**, *the last one given is used. For instance, a* **.WC WF** *—***WF** *command has the effect of a* **.WC** *—***WF**.

Column Headings for Two–Column Output

NOTE

This section is intended only for users accustomed to writing formatter macros.

In two–column processing output, it is sometimes necessary to have headers over each column, as well as headers over the entire page. This is accomplished by redefining the **.TP** macro to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP'OVERALL"
.tl ''TITLE''
.sp
.nf
.ta 16C 31R 34 50C 65R
left◯center◯right◯left◯center◯right
◯first column◯◯◯second column
.fi
.sp 2
..
```

In the preceding example, ◯ stands for the tab character.

This example will produce two lines of page header text plus two lines of headers over each column. Tab stops are for a 65–en overall line length.

Vertical Spacing

.SP *[lines]*

Several ways of obtaining vertical spacing exist, all with different effects. The **.sp** request spaces the number of lines specified unless the no space (**.ns**) mode is on; then, the **.sp** request is ignored. The no space mode is set at the end of a page header to eliminate spacing by a **.sp** or **.bp** request that happens to occur at the top of a page. This mode can be turned off by the **.rs** (restore spacing) request.

The **.SP** macro is used to avoid the accumulation of vertical space by successive macro calls. Several **.SP** calls in a row will not produce the sum of the arguments but only the maximum argument. For example, the following produces only three blank lines:

```
.SP 2
.SP 3
.SP
```

Many MM macros utilize **.SP** for spacing. For example, **.LE 1** immediately followed by **.P** produces only a single blank line (one–half a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (one vertical space). Negative arguments are not permitted. The argument must be unscaled but fractional amounts are permitted. The **.SP** macro (as well as **.sp**) is also inhibited by the **.ns** request.

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 af a page (then it does nothing). A **.SK** *n* command skips *n* pages. A **.SK** positions text that follows it at the top of a page. **.SK 1** leaves one page blank except for the header and footer.

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 it), 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 prevailing point size and vertical spacing may be changed by invoking the **.S** macro. In the **troff** formatter, the default point size (obtained from the MM register

S) is 10 points, and the vertical spacing is 12 points (six lines per inch). The mnemonics *D* (default value), *C* (current value), and *P* (previous value) may be used for both arguments.

- If an argument is *negative*, current value is decremented by the specified amount.
- If an argument is *positive*, current value is incremented by the specified amount.
- If an argument is *unsigned*, it is used as the new value.
- If there are no arguments, the **.S** macro defaults to *P*.
- If the first argument is specified but the second is not, then (default) *D* is used for the vertical spacing.

Default value for vertical spacing is always two points greater than the current point size. Footnotes are two points smaller than the body, with an additional three-point space between footnotes. A null ('' '') value for either argument defaults to *C* (current value). Thus, if *n* is equal to a defined numeric value, the following applies:

```
.S          =  .S P P
.S"" n      =  .S C n
.S n""      =  .S n C
.S n        =  .S n D
.S""        =  .S C D
.S"" ""     =  .S C C
.S n n      =  .S n n
```

If the first argument is greater than 99, the default point size (10 points) is restored. If the second argument is greater than 99, the default vertical spacing (current point size plus two points) is used. For example:

.S 100 = .S 10 12
.S 14 111 = .S 14 16

Reducing Point Size of a String

.SM *string1* [*string2*] [*string3*]

The **.SM** macro allows the user to reduce by one point the size of a string. If the third argument (*string3*) is omitted, the first argument (*string1*) is made smaller and is concatenated with the second argument (*string2*) if specified. If all three arguments are present (even if any are null), the second argument is made smaller and all three arguments are concatenated. For example:

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

Strings may be used to produce accents for letters as shown in the following examples:

	INPUT	OUTPUT
Grave accent	c*‘	ċ
Acute accent	e*’	ė
Circumflex	o*^	ô
Tilde	n*	ñ
Cedilla	c*,	ç
Lower-case umlaut	u*:	ü
Upper-case umlaut	U*;	Ü

Inserting Text Interactively

.RD [*prompt*] [*diversion*] [*string*]

The **.RD** (read insertion) macro allows a user to stop the standard output of a document and to read text from the standard input until two consecutive newline characters are found. When newline characters are encountered, normal output is resumed.

- The *prompt* argument will be printed at the terminal. If not given, **.RD** signals the user with a **BEL** on terminal output.
- The *diversion* argument allows the user to save all text typed in after the prompt in a macro whose name is that of the diversion.
- The *string* argument allows the user to save for later reference the first line following the prompt in the named string.

The **.RD** macro follows the formatting conventions in effect. Thus, the following examples assume that the **.RD** call is invoked in no–fill mode (**.nf**):

```
.RD Name aA bB
```

This input produces the following:

```
Name: J.Jones  (user types name)
16 Elm Rd.,
Piscataway
```

The diverted macro **.aA** will contain this information:

```
J.Jones
16 Elm Rd.,
Piscataway
```

The string *bB* (*(bB) contains *J.Jones*.

A newline character followed by an **EOF** (user specifiable <CTRL–D>) also allows the user to resume normal output.

Errors and Debugging

Error Terminations

When a macro detects an error, the following actions occur:

- A break occurs.
- The formatter output buffer (which may contain some text) is printed to avoid confusion regarding location of the error.
- A short message is printed giving the name of the macro that detected the error, type of error, and approximate line number in the current input file of the last processed input line. Error messages are explained in the last subsection of this section.
- Processing terminates unless register **D** has a positive value. In the latter case, processing continues even though the output is guaranteed to be nonsensical 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, 450**, or **hp** is being used to post–process the **nroff** formatter output, the message may be garbled by being intermixed with text held in that filter's output buffer.

NOTE

If either **neqn** *or* **tbl** *is being used, and if the* **—o***list option of the formatter causes the last page of the document not to be printed, a "broken pipe" message may result; this message is not indicative of a problem.*

Disappearance of Output

Disappearance of output usually occurs because of an unclosed diversion (such as a missing **.DE** or **.FE** macro). Fortunately, macros that use diversions were carefully designed; 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 prints related macros that are found in *files*...:

grep –n '^ \.**[EDFRT] [EFNQS]**' *files*...

Because of what was asked for in the brackets, the macros **.DF, .DS, .DE, .EQ, .EN, .FS, .FE, .RS, .RF, .TS,** and **.TE** are printed with each preceded by its filename 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 alpha–numeric character (n, a, or A, for example, letter or digit)
- s: Any nonalpha–numeric character (special character)

All other characters are literals (characters that stand for themselves).

Request, macro, and string names are kept by the formatters in a single internal table; therefore, there must be no duplication among such names. Number register names are kept in a separate table.

Names Used by Formatters

requests:	aa (most common) an (only one, currently: c2)
registers:	aa (normal) .x (normal) .s (only one, currently: .$) a. (only one, currently: c.) % (page number)

Names Used by MM

macros and strings:	A, AA, Aa (accessible to users; for example macros **P** and **HU**, strings *F*, *BU*, and *Lt*). nA (accessible to users; only two, currently: 1C and 2C). aA (accessible to users; only one, currently: nP). s (accessible to users; only the seven accents, currently).)x,}x,]x, >x, ?x (internal).

registers:	An, Aa (accessible to users; for example, **H1**, **Fg**).
	A (accessible to users; meant to be set on the command line; for example, **C**).
	:x, ;x, #x, ?x, !x (internal).

Names Used by neqn and tbl

The mathematical equation preprocessor, **neqn**, uses registers and string names of the form *nn*. The table preprocessor, **tbl**, uses *T*, *T#*, and *TW*, and names of the form:

a- a+ a¦ 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. The following is a possible naming convention:

macros:	aA (**bG**, **kW**, for example)
strings:	as (*c)*, *f]*, *p*, for example)
registers:	a (**f**, **t**, for example)

Sample Extensions

Appendix headings

The following is a way of generating and numbering appendix headings:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH " ´ ´ ´ Appendix \\na- \\\\\\\\nP ´ "
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form

.aH "*title*"

beings a new page (with the page header changed to *Appendix a-n*) and generates an unnumbered heading of *title*. This title, if desired, can be saved for the table of contents. To center appendix titles, the **Hc** register must be set to 1.

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 make to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*— It is not necessary to write complex sequences of commands to produce documents. Reasonable macro argument default values are provided where possible.
- *Parameterization*— There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt input text files to produce output documents to their respective needs over a wide range of styles.
- *Extension by moderately expert users*— A strong effort has been made to user mnemonic naming conventions and consistent techniques in construction of macros. Naming conventions are given so that a user can add new macros or redefine existing ones if necessary.
- *Device independence*— A common use of MM is to produce documents on hard copy via teletypewriter terminals using the **nroff** formatter. Macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be displayed on an appropriate video terminal. Macros have been constructed to allow compatibility with the **troff** formatter so that output can be produced on both a photoypesetter and a teletypewriter/video terminal.
- *Minimization of input*— The design of macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, the user need only set a specific parameter once at the beginning of a document rather than type a blank line after each such heading.
- *Decoupling of input format from output style*— There is one way to prepare the input text although the user may obtain a number of output styles by setting a few global options. For example, the **.H** macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or within a single document.

MM Macro Name Summary

The following listing shows all the MM macros and their usage. Each item in the list gives a definition of the macro followed by its normal format and arguments.

NOTE

Macros marked with an asterisk are not, in general, called (invoked) directly by the user. They are user exits defined by the user and called by the MM macros from inside header, footer, or other macros.

1C One–column processing
.1C

2C Two–column processing
.2C

AE Abstract end
.AE

AF Alternate format of *Subject/Date/From* block
.AF [*company-name*]

AL Automatically incremented list start
.AL [*type*] [*text-indent*] [1]

AS Abstract start
.AS [*arg*] [*indent*]

AT Author's title
.AT [*title*]

AU Author information
.AU *name* [*initials*] [*loc*] [*dept*] [*ext*] [*room*] [*arg*] [*arg*] [*arg*]

AV Approval signature
.AV [*name*] [1]

B Bold
.B [*bold-arg*] [*previous-font-arg*] [*bold*] [*prev*] [*bold*] [*prev*]

BE Bottom block end
.BE

BI Bold/Italic
.BI [*bold-arg*] [*italic-arg*] [*bold*] [*italic*] [*bold*] [*italic*]

BL Bullet list start
.BL [*text-indent*] [1]

BR Bold/Roman
.BR [*bold-arg*] [*Roman-arg*] [*bold*] [*Roman*] [*bold*] [*Roman*]

BS Bottom block start
.BS

CS Cover Sheet
.CS [*pages*] [*other*] [*total*] [*figs*] [*tbls*] [*refs*]

DE Display end
.DE

DF Display floating start
.DF [*format*] [*fill*] [*right-indent*]

DL Dash list start
.DL [*text-indent*] [1]

DS Display static start
.DS [*format*] [*fill*] [*right-indent*]

EC Equation caption
.EC [*title*] [*override*] [*option*]

EF Even–page footer
.EF [*arg*]

EH Even–page header
.EH [*arg*]

EX Exhibit caption
.EX [*title*] [*override*] [*option*]

FC Formal closing
.FC [*closing*]

FD Footnote default format
.FD [*arg*] [1]

FE Footnote end
.FE

FG Figure title
.FG [*title*] [*override*] [*option*]

FS Footnote start
.FS [*label*]

H heading—numbered
.H *level* [*heading-text*] [*heading-suffix*]

HC Hyphenation character
.HC [*hyphenation-indicator*]

HM Heading mark style (Arabic or Roman numerals, or letters)
.HM [*arg1*] . . . [*arg7*]

HU Heading—unnumbered
.HU *heading-text*

HX* Heading user exit X (before printing heading)
.HX *d-level r-level heading text*

HY* Heading user exit Y (before printing heading)
.HY *d-level r-level heading-text*

HZ* Heading user exit Z (after printing heading)
.HZ *d-level r-level heading-text*

I Italic (underline in the **nroff** formatter)
.I [*italic-arg*] [*previous-font-arg*] [*italic*] [*prev*] [*italic*] [*prev*]

IB Italic/Bold
.IB [*italic-arg*] [*bold-font-arg*] [*italic*] [*bold*] [*italic*] [*bold*]

IR Italic/Roman
.IR [*italic-arg*] [*Roman-arg*] [*italic*] [*Roman*] [*italic*] [*Roman*]

LB List begin
.LB *text-indent mark-indent pad type* [*mark*] [*LI-space*] [*LB-space*]

LC List–status clear
.LC [*list-level*]

LE List end
.LE [1]

LI List item
.LI [*mark*] [1]

ML Marked list start
.ML *mark* [*text-indent*]

MT Memorandum type
.MT [*type*] [*addressee*] or .MT [4] [1]

ND New date
.ND *new-date*

NE Notation end
.NE

NS Notation start
.NS [*arg*] [1] nP" Double–line indented paragraphs
.nP

OF Odd–page footer
.OF [*arg*]

OH Odd–page header
.OH [*arg*]

OK Other keywords for the Technical Memorandum cover sheet
.OK [*keyword*]

OP Odd page
.OP

P Paragraph
.P [*type*]

PF Page footer
.PF [*arg*]

PH Page header
.PH [*arg*]

PM proprietary marking
.PM [*code*]

PX* Page–header user exit
.PX

R Return to regular (Roman) font
.R

RB Roman/Bold
.RB [*Roman-arg*] [*bold-arg*] [*Roman*] [*bold*] [*Roman*] [*bold*]

RD Read insertion from terminal
.RD [*prompt*] [*diversion*] [*string*]

RF Reference end
.RF

RI Roman/Italic
.RI [*Roman-arg*] [*italic-arg*] [*Roman*] [*italic*] [*Roman*] [*italic*]

RL Reference list start
.RL [*text-indent*] [1]

RP Produce reference page
.RP [*arg*] [*arg*]

RS Reference start
.RS [*string-name*]

S set **troff** formatter point size and vertical spacing
.S [*size*] [*spacing*]

SA Set adjustment (right–margin justification) default
.SA [*arg*]

SG Signature line
.SG [*arg*] [1]

SK Skip pages
.SK [*pages*]

SM Make a string smaller
.SM *string1* [*string2*] [*string3*]

SP Space vertically
.SP [*lines*]

TB Table title
.TB [*title*] [*override*] [*option*]

TC Table of contents
.TC [*s-level*] [*spacing*] [*t-level*] [*tab*[] [*h1*] [*h2*] [*h3*] [*h4*] [*h5*]

TE Table end
.TE

TH Table header
.TH [*N*]

TL Title of memorandum
.TL [*charging-case*] [*filing-case*]

TM Technical Memorandum number(s)
.TM [*number*] . . .

TP* Top of page macro
.TP

TS Table start
.TS [*H*]

TX* Table of contents user exit
.TX

TY* Table of contents user exit (suppresses *CONTENTS*)
.TY

VL Variable-item list start
.VL *text-indent* [*mark-indent*] [1]

VM Vertical margins
.VM [*top*] [*bottom*]

WC Footnote and display width control
.WC [*format*]

MM String Name Summary

The following list shows the predefined string names used by the MM macro package.

BU Bullet
NROFF: ●
TROFF: ●

Ci Table of contents indent list
Up to seven scaled arguments for heading levels

DT Date
Current date, unless overridden
Month, day, year (for example, May 31, 1979)

EM Em dash string
Produces an em dash in the **troff** formatter and a double hyphen in **nroff**

F Footnote number generator
NROFF: \u\\n+(:p\d
TROFF: \v'-.4m'\s-3\\n+(:p\s0\v'.4m'

HF Heading font list
Up to seven codes for heading levels one through seven
2 2 2 2 2 2 2 (all levels underlined by **nroff** and italicized by **troff**)

HP Heading point size list
Up to seven codes for heading levels one through seven

Le Title for LIST OF EQUATIONS

Lf Title for LIST OF FIGURES

Lt Title for LIST OF TABLES

Lx Title for LIST OF EXHIBITS

RE SCCS Release and Level of MM
Release.Level (for example, 15.129)

Rf Reference number generator

Rp Title for References

Tm Trademark string
Places the letters *TM* one-half line above the text that is follows

Seven accent strings are also available.

NOTE

If the released-paper style is used, then (in addition to the above strings) certain **BTL** *location codes are defined as strings. These location strings are needed only until the .MT macro is called. Currently, the following codes 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.

MM Number Register Summary

The list that follows contains a description of all the predefined number registers used by MM. After each description is the normal value of the register followed by the range of allowable values, enclosed in brackets[ ]. The lower and upper limit of values are separated by a colon (:).

Note 1: *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.*

Note 2: *The Introduction subsection defines setting and referencing registers. Any register having a single-character name can be set from the command line. These are indicated by a dagger (†) in the following list.*

A * † Handles preprinted forms and logo
0, [0:2]

Au Inhibits printing of author information
1, [0:1]

C * † Copy type (original, DRAFT, etc.)
0 (original), [0:4]

Cl Level of headings saved for table of contents
2, [0:7]

Cp Placement of list of figures, etc.
1 (on separate pages), [0:1]

D * † Debug option
0, [0:1]

De Display eject register for floating displays
0, [0:1]

Df Display format register for floating displays
5, [0:5]

Ds Static display pre- and post-space
1, [0:1]

E * † Controls font of the Subject/Date/From fields
1 (**nroff**) 0 (**troff**), [0:1]

Ec Equation counter used by **.EC** macro
0, [0:?]
Incremented by one for each **.EC** call

Ej Page-ejection option for headings
0 (no eject), [0:7]

Eq Equation label placement
0 (right-justified), [0:1]

Ex Exhibit counter, used by **.EX** macro
0, [0:?]
Incremented by one for each **.EX** call.

Fg Figure counter, used by **.FG** macro
0, [0:?]
Incremented by one for each **.FG** call.

Fs Footnote space (for example, spacing between footnotes)
1, [0:?]

H1–H7 Heading counters for levels 1 through 7
0, [0:?]
Incremented by the **.H** macro of corresponding level or the **.HU** macro if at level given by the **Hu** register. The **H2** through **H7** registers are reset to 0 by any **.H** (**.HU**) macro at a lower-numbered level.

Hb Heading break level (after **.H** and **.HU**)
2, [0:7]

Hc Heading centering level for **.H** and **.HU**
0 (no centered headings), [0:7]

Hi Heading temporary indent (after **.H** and **.HU**)
1 (indent as paragraph), [0:2]

Hs Heading space level (after **.H** and **.HU**)
2 (space only after **.H 1** and **.H 2**), [0:7]

Ht Heading type (for **.H**: single or concatenated numbers)
0 (concatenated numbers: 1.1.1, etc.), [0:1]

Hu Heading level for unnumbered heading (**.HU**)
2 (**.HU** at the same level as **.H 2**), [0:7]

Hy Hyphenation control for body of document
0 (automatic hyphenation off), [0:1]

L * †
Length of page
66, [20:?]
(11i, [2i:?] in **troff** formatter)

Le
List of equations
0 (list not produced), [0:1]

Lf
List of figures
1 (list produced), [0:1]

Li
List indent
6 (**nroff**) 5 (**troff**), [0:?]

Ls
List spacing between items by level
6 (spacing between all levels), [0:6]

Lt
List of tables
1 (list produced), [0:1]

Lx
List of exhibits
1 (list produced), [0:1]

N * †
Numbering style
0, [0:5]

Np
Numbering style for paragraphs
0 (unnumbered), [0:1]

O * †
Offset of page
.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
0 (lowercase Roman), [0:1]

Of
Figure caption style
0 (period separator), [0:1]

P
Page number managed by MM
0, [0:?]

Pi
Paragraph indent
5 (**nroff**) 3 (**troff**), [0:?]

Ps
Paragraph spacing
1(one blank space between paragraphs), [0:?]

Pt
Paragraph type
0 (paragraphs always left–justified), [0:2]

Pv "PRIVATE" header
0 (not printed), [0:2]

Rf Reference counter, used by **.RS** macro
0, [0:?]
Incremented by one for each **.RS** call.

S * † The **troff** formatter default point size
10, {6:36]

Si Standard indent for displays
5 (**nroff**) 3 (**troff**), [0:?]

T * † Type of **nroff** output device
0, [0:2]

Tb Table counter, used by **.TB** macro
0, [0:?]
Incremented by one for each **.TB** call.

U Underlining style (**nroff**) for **.H** and **.HU**
0 (continuous underline when possible), [0:1]

W Width of page (line and title length)
6i, [10:1365]
(6i, [2i:7.54i] in the **troff** formatter)

MM and Formatter Error Messages

When processing text using the MM macro package, MM will report any errors it can detect. Since the macros are written in the basic formatter primitives, other errors may be found and reported by the formatter. The next two subsections contain descriptions of any error messages which may be received from MM or the formatter.

MM 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 descriptive message usually beginning with a macro name. They are listed below in alphabetical order by macro name, each with a more complete explanation.

Check TL, AU, AS, AE, MT sequence

The correct order of macros at the start of a memorandum is shown in subsection *Memorandum and Released-Paper Documents*. 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 the subsection *Memorandum and Released-Paper Documents*. 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; for example, 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; for example, more than 26 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; for example, a **.DE** had 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; for example, a **.DS** or **.DF** occurs without a matching **.DE**.

FS:missing FE

A previous **.FS** macro was not matched by a closing **.FE**; for example, 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 **.HU**) occurs inside a footnote.

HU:missing arg

The **.HU** macro needs one argument.

LB:missing arg(s)

The **.LB** macro requires at least four arguments.

LB:too many nested lists

Another list was started when there were already six active lists.

LE:mismatched

The **.LE** macro has occurred without a previous **.LB** or other list–initialization macro. This is not a serious error. The message is issued because there exists some problem in the preceding text.

LI:no lists active

The **.LI** macro occurred without a preceding list–initialization macro. The latter probably has been omitted or entered incorrectly.

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 **.RP** macro was not matched by a closing **.RF**.

RS:missing RF

A previous **.RS** macro was not matched by a closing **.RF**.

S:bad arg:*value*

The incorrect argument *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.

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.

Cannot do ev

Caused by:

1. Setting a page width that is negative or extremely short
2. Setting a page length that is negative or extremely short
3. Reprocessing a macro package (for example, performing a **.so** request on a macro package that was already requested on the command line)

4. Requesting the **troff** formatter *-s1* option on a document that is longer than ten pages.

Cannot execute *filename*

Given by the **.!** request if the *filename* is not found.

Cannot open *filename*

Indicates one of the files in the list of files to be processed cannot be opened.

Exception word list full

Indicates too many words have been specified in the hyphenation exception list (via **.hw** request).

Line overflow

Indicates output line being generated was too long for the formatter line buffer capacity. The excess was discarded. Likely causes for this message are very long lines or words generated through the misuse of \c of the **.cu** request, or very long equations produced by **eqn/neqn**.

Nonexistent font type

Indicates a request has been made to mount an unknown font.

Nonexistent macro file

Indicates the requested macro package does not exist.

Nonexistent terminal type

Indicates the terminal options refer to an unknown terminal type.

Out of temp file space

Indicates additional temporary space for macro definitions, diversion, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing **.FE** or **.DE**), unclosed macro definitions (for example, 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

Indicated a word being generated exceeded the formatter word buffer capacity. Excess characters were discarded. Likely causes for this messages are very long lines, words generated through the misuse of **\c** of the **.cu** request, or very long equations produced by **eqn/neqn**.

The ME Reference Guide

Introduction

This document describes in extremely terse form the features of the **–me** macro package for **nroff/troff**. Some familiarity is assumed with those programs, specifically, the reader should understand breaks, fonts, point sizes, the use and definition of number registers and strings, how to define macros, and scaling factors for ens, points, **v** 's (vertical line spaces), etc.

For a more casual introduction to text processing using **nroff**, refer to Section 4F, *The ME Text-Formatting Macros.*

There are a number of macro parameters that may be adjusted. Fonts may be set to a font number only. In **nroff** font 8 is underlined, and is set in bold font in **troff** (although font 3, bold in **troff**, is not underlined in **nroff**). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are *pseudo-fonts*; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

\f8

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one–en spaces is:

.nr pi 8n

and not

.nr pi 8

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form **$***x* may be used in expressions but should not be changed. Macros of the form **$***x* perform some function (as described) and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

All names in **–me** follow a rigid naming convention. The user may define number registers, strings, and macros, provided that s/he uses single character uppercase names or double character names consisting of letters and digits, with at least one uppercase letter. In no case should special characters be used in user–defined names.

On daisy wheel-type printers in 12 pitch, the *-rx1* flag can be stated to make lines default to 1/8 inch (the normal spacing for a newline in 12-pitch). This is normally too small for easy readability, so the default is to space 1/6 inch.

Paragraphing

These macros are used to begin paragraphs. The standard paragraph macro is **.pp**; the others are all variants to be used for special purposes.

The first call to one of the paragraphing macros defined in this section or the **.sh** macro (defined in the next session) *initializes* the macro processor. After initialization you cannot use any of the following requests: **.sc**, **.lo**, **.th**, or **.ac**. Also, the effects of changing parameters that have a global effect on the format of the page (notably page length and header and footer margins) are not well defined and should be avoided.

.lp Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to \n(pf [1], the type size is set to \n(pp [10p], and a \n(ps space is inserted before the paragraph [0.35v in **troff**, 1v or 0.5v in **nroff** depending on device resolution]. The indent is reset \n($i [0] plus \n(po [0] unless the paragraph is inside a display (see **.ba**). At least the first two lines of the paragraph are kept together on a page.

.pp Like **.lp** , except that it puts \n(pi [5n] units of indent. This is the standard paragraph macro.

.ip *T I* Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or \n(ii [5n] spaces if *I* is not specified) more than a nonindented paragraph (such as with **.pp**) is. The title *T* is exdented (opposite of indented). The result is a paragraph with an even left edge and *T* printed in the margin. Any spaces in *T* must be unpaddable. If *T* does not fit in the space provided, **.ip** does start a new line.

.np A variant of **.ip** that numbers paragraphs. Numbering is reset after a **.lp**, **.pp**, or **.sh**. The current paragraph number is in \n($p.

Section Headings

Numbered sections are similar to paragraphs except that a section number is automatically generated for each one. The section numbers are of the form *n.n.n*. The *depth* of the section is the count of numbers (separated by decimal points) in the section number.

Unnumbered section headings are similar, except that no number is attached to the heading.

.sh *+N T a b c d e f* Begin numbered section of depth *N*. If *N* is missing the current depth (maintained in the number register *\n($0)* is used. The values of the individual parts of the section number are maintained in *\n($1* through *\n($6*. There is a *\n(ss* [1v] space before the section. *T* is printed as a section title in font *\n(sf* [8] and size *\n(sp* [10p]. The *name* of the section may be accessed by using **($n*. If *\n(si* is nonzero, the base indent is set to *\n(si* times the section depth, and the section title is exdented (see **.ba**.) Also, an additional indent of *\n(so* [0] is added to the section title (but not to the body of the section). The font is then set to the paragraph font, so that more information may occur on the line with the section number and title. **.sh** insures that there is enough room to print the section head plus the beginning of a paragraph (about three lines total). If *a* through *f* are specified, the section number is set to that number rather than incremented automatically. If any of *a* through *f* are a hyphen that number is not reset. If *T* is a single underscore (_) then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.

.sx *+N* Go to section depth *N* [−1], but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.

.uh *T* Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as for **.sh**.

.$p *T B N* Print section heading. May be redefined to get fancier headings. *T* is the title passed on the **.sh** or **.uh** line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, **.sh** passes all three, **.uh** passes only the first, and **.sx** passes three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.

.$0 *T B N* This macro is called automatically after every call to **.$p**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similar function. *T* is the section title for the section title that was just printed, *B* is the section number, and *N* is the section depth.

.$1–.$6 Traps called just before printing that depth section. May be defined (for example) to give variable spacing before sections. These macros are called from **.$p**, so if you redefine that macro, you may lose this feature.

Headers and Footers

Headers and footers are put at the top and bottom of every page automatically. They are set in font *\n(tf* [3] and size *\n(tp* [10p]. Each of the definitions apply as of the *next* page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers is controlled by three number registers. *\n(hm* [4v] is the distance from the top of the page to the top of the header, *\n(fm* [3v] is the distance from the bottom of the page to the bottom of the footer, *\n(tm* [7v] is the distance from the top of the page to the top of the text, and *\n(bm* [6v] is the distance from the bottom of the page to the bottom of the text (nominal). The following macros are also supplied for compatibility with **roff** documents:

.m1 **.m2**
.m3 **.m4**

.he *'l'm'r'* Define three-part header, to be printed at the top of every page.

.fo *'l'm'r'* Define footer, to be printed at the bottom of every page.

.eh *'l'm'r'* Define header, to be printed at the top of every even-numbered page.

.oh *'l'm'r'* Define header, to be printed at the top of every odd-numbered page.

.ef *'l'm'r'* Define footer, to be printed at the bottom of every even-numbered page.

.of *'l'm'r'* Define footer, to be printed at the bottom of every odd-numbered page.

.hx Suppress headers and footers on the next page.

.m1 *+N* Set the space between the top of the page and the header [4v].

.m2 *+N* Set the space between the header and the first line of text [2v].

.m3 *+N* Set the space between the bottom of the text and the footer [2v].

.m4 *+N* Set the space between the footer and the bottom of the page [4v].

.ep End this page, but do not begin the next page. Useful for forcing out footnotes, but other than that rarely used. Must be followed by a **.bp** or the end of input.

.$h Called at every page to print the header. May be redefined to provide fancy (for example, *multi-line*) headers, but doing so loses the function of the **.he**, **.fo**, **.eh**, **.oh**, **.ef**, and **.of** requests, as well as the chapter-style title feature of **.+c**.

.$f Print footer; same comments apply as in **.$h**.

.$H A normally undefined macro that is called at the top of each page (after outputting the header, initial saved floating keeps, etc.). In other words, this macro is called immediately before printing text on a page. It can be used for column headings and the like.

Displays

All displays except centered blocks and block quotes are preceded and followed by an extra \n(bs [same as \n(ps] space. Quote spacing is stored in a separate register; centered blocks have no default initial or trailing space. The vertical spacing of all displays except quotes and centered blocks is stored in register \n($R instead of \n($r.

.(l *m f* Begin list. Lists are single spaced, unfilled text. If *f* is **F**, the list is filled. If *m* [**I**] is **I** the list is indented by \n(bi [4n]; if **M** the list is indented to the left margin; if **L** the list is left justified with respect to the text (different from **M** only if the base indent (stored in \n($i and set with **.ba**) is not zero); and if **C** the list is centered on a line-by-line basis. The list is set in font \n(df [0]. Must be matched by a **.)l**. This macro is almost like **.(b** except that no attempt is made to keep the display on one page.

.)l End list.

.(q Begin major quote. These are single-spaced, filled, moved in from the text on both sides by \n(qi [4n], preceded and followed by \n(qs [same as \n(bs] space, and are set in point size \n(qp [one point smaller than surrounding text].

.)q End major quote.

.(b *m f* Begin block. Blocks are a form of *keep*, where the text of a keep is kept together on one page if possible. Keeps are useful for tables and figures that should not be broken over a page. If the block does not fit on the current page, a new page is begun, unless that would leave more than \n(bt [0] white space at the bottom of the text. If \n(bt is 0, the threshold feature is turned off. Blocks are not filled unless *f* is **F**, when they are filled. The block is left-justified if *m* is **L**, indented by \n(bi [4n] if *m* is **I** or absent, centered (line-for-line) if *m* is **C**, and left-justified to the margin (not to the base indent) if *m* is **M**. The block is set in font \n(df [0].

.)b End block.

.(z *m f* — Begin floating keep. Like **.(b** except that the keep is *floated* to the bottom of the page or the top of the next page. Therefore, its position is relative to the text changes. The floating keep is preceded and followed by \n(zs [1v] space. Also, it defaults to mode **M**.

.)z — End floating keep.

.(c — Begin centered block. The next keep is centered as a block, rather than on a line–by–line basis as with **.(b C**. This call may be nested inside keeps.

.)c — End centered block.

Annotations

.(d — Begin delayed text. Everything in the next keep is saved for output later with **.pd**, in a manner similar to footnotes.

.)d *n* — End delayed text. The delayed text number register \n($d and the associated string *# are incremented if *# has been referenced.

.pd — Print delayed text. Everything diverted by **.(d** is printed and truncated. This might be used at the end of each chapter.

.(f — Begin footnote. The text of the footnote is floated to the bottom of the page and set in font \n(ff [1] and size \n(fp [8p]. Each entry is preceded by \n(fs [0.2v] space, is indented \n(fi [3n] on the first line, and is indented \n(fu [0] from the right margin. Footnotes line up underneath two columned output. If the text of the footnote does not all fit on one page, it is carried over to the next page.

.)f *n* — End footnote. The number register \n($f and the associated string ** are incremented if they have been referenced.

.$s — The macro to output the footnote separator. This macro may be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.

.(x *x* — Begin index entry. Index entries are saved in the index *x* [**x**] until called up with **.xp**. Each entry is preceded by a \n(xs [0.2v] space. Each entry is *undented* by \n(xu [0.5i]; this register tells how far the page number extends into the right margin.

.)x *P A* — End index entry. The index entry is finished with a row of dots with *A* (null) right justified on the last line (such as for an author's name), followed by *P* [\n%]. If *A* is specified, *P* must be specified; \n% can be used to print the current page number. If *P* is an underscore, no page number and no row of dots are printed.

.xp *x* — Print index *x* [**x**]. The index is formatted in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

Columned Output

.2c *+S N* — Enter two–column mode. The column separation is set to *+S* [4n, 0.5i in ACM mode] (saved in *\n($s)*. The column width, calculated to fill the single column line length with both columns, is stored in *\n($l*. The current column is in *\($c*. You can test register *\n($m* [1] to see if you are in single column or double column mode. Actually, the request enters *N* [2] columned output.

.1c — Revert to single–column mode.

.bc — Begin column. This is like **.bp** except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

Fonts and Sizes

.sz *+P* — The point size is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to point size is stored in *\n($r*. The ratio used internally by displays and annotations is stored in *\n($R* (although this is not used by **.sz**).

.r *W X* — Set *W* in Roman font, appending *X* in the previous font. To append different font requests, use *X* = *\c*. If no parameters, change to Roman font.

.i *W X* — Set *W* in italics, appending *X* in the previous font. If no parameters, change to italic font. Underlines in **nroff**.

.b *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. Underlines in **nroff**.

.rb *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. **.rb** differs from **.b** in that **.rb** does not underline in **nroff**.

.u *W X* — Underline *W* and append *X*. This is a true underlining, as opposed to the **.ul** request, which changes to *underline font* (usually italics in **troff**). It doesn't work correctly if *W* is spread or broken (including hyphenated). In other words, it is safe in no–fill mode only.

.q *W X* — Quote *W* and append *X*. In **nroff** this just surrounds *W* with double quote marks (") but in **troff** uses directed quotes.

.bi *W X* — Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in **nroff**. It doesn't work correctly if *W* is spread or broken (including hyphenated). In other words, it is safe in no–fill mode only.

.bx *W X* — Sets *W* in a box, with *X* appended. Underlines in **nroff**. It doesn't work correctly if *W* is spread or broken (including hyphenated). In other words, it is safe in no–fill mode only.

Roff Support

.ix +*N* — Indent, no break. Equivalent to **'in** *N*.

.bl *N* — Leave *N* contiguous white space, on the next page if not enough room on this page. Equivalent to a **.sp** *N* inside a block.

.pa +*N* — Equivalent to **.bp**.

.ro — Set page number in Roman numerals. Equivalent to **.af % i**.

.ar — Set page number in Arabic. Equivalent to **.af % 1**.

.n1 — Number lines in margin from one on each page.

.n2 *N* — Number lines from *N*, stop if *N* = 0.

.sk — Leave the next output page blank, except for headers and footers. This is used to leave space for a full–page diagram, that is produced externally and pasted in later. To get a partial–page paste–in display, say **.sv** *N*, where *N* is the amount of space to leave; this space is output immediately if there is room, and otherwise output at the top of the next page. However, be warned: if *N* is greater than the amount of available space on an empty page, no space is ever output.

Preprocessor Support

.TS *h* — Table start. Tables are single–spaced and kept on one page if possible. If you have a large table that does not fit on one page, use *h* = **H** and follow the header part (to be printed on every page of the table) with a **.TH**. See Section 4I, *Table Formatting Program (Tbl)*.

.TH — With **.TS H**, ends the header portion of the table.

.TE — Table end. Note that this table does not float; in fact, it is not even guaranteed to stay on one page if you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z**.

Miscellaneous

.re Reset tabs. Set to every 0.5i in **troff** and every 0.8i in **nroff**.

.ba +*N* Set the base indent to +*N* [0] (saved in *\n($i*). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are not affected. The **.sh** request performs a **.ba** request if *\n(si* [0] is not 0, and sets the base indent to *\n(si*\n($0*.

.xl +*N* Set the line length to *N* [6.0i]. This differs from **.ll** because it affects only the current environment.

.ll +*N* Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-columned output. The current line length is stored in *\n($l*.

.hl Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.

.lo This macro loads another set of macros (in */usr/lib/me/local.me*), which is intended to be a set of locally defined macros. These macros should all be of the form **.***X* where *X* is any letter (upper- or lower-case) or digit.

Standard Papers

.tp Begin title page. Spacing at the top of the page can occur, and headers and footers are suppressed. Also, the page number is not incremented for this page.

.th Set thesis mode. It double-spaces, defines the header to be a single-page number, and changes the margins to be 1.5 inch on the left and one inch on the top. **.++** and **.+c** should be used with it. This macro must be stated before initialization, that is, before the first call of a paragraphing macro or **.sh**.

.++ *m H* This request defines the section of the paper that we are entering. The section type is defined by *m* and can come from the following:

m	**SECTION TYPE**
C	entering chapter portion of paper
A	entering appendix portion of paper
P	material following should be preliminary portion (abstract, table of contents, etc.)
AB	entering abstract (numbered independently from 1 in Arabic numerals)
B	entering bibliographic portion at the end

Also, the variants **RC** and **RA** are allowed, which specify renumbering of pages from one at the beginning of each chapter or appendix, respectively. The *H* parameter defines the new header. If there are any spaces in it, the entire header must be quoted. If you want the header to have the chapter number in it, use the string *n(ch*. For example, to number appendixes *A.1* etc., enter **.++ RA '''\\\\n(ch.%'**. Each section (chapter, appendix, etc.) should be preceded by the **.+c** request. It is easier when using **troff** to put the front material at the end of the paper, so that the table of contents can be collected and output; this material can then be physically moved to the beginning of the paper.

.+c *T* — Begin chapter with title *T*. The chapter number is maintained in *n(ch*. This register is incremented every time **.+c** is called with a parameter. The title and chapter number are printed by **.$c**. The header is moved to the footer on the first page of each chapter. If *T* is omitted, **.$c** is not called; this is useful for doing your own *title page* at the beginning of papers without a title page proper. **.$c** calls **.$C** as a hook so that chapter titles can be inserted into a Table of Contents automatically. The footnote numbering is reset to one.

.$c *T* — Print chapter number (from *n(ch*) and *T*. This macro can be redefined to your liking. This macro calls **$C**, which can be defined to make index entries, or whatever.

.$C *K N T* — This macro is called by **.$c**. It is normally undefined, but can be used to automatically insert index entries, or whatever. *K* is a keyword, either *Chapter* or *Appendix* (depending on the .++ mode); *N* is the chapter or appendix number; and *T* is the chapter or appendix title.

.ac *A N* — This macro (short for **.acm**) sets up the **nroff** environment for photo-ready papers as used by the ACM. This format is 25% larger, and has no headers or footers. The author's name *A* is printed at the bottom of the page (but off the part that is printed in the conference proceedings), together with the current page number and the total number of pages *N*. Additionally, this macro loads the file */usr/lib/me/acm.me*, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro does not work correctly in **troff**, since it sets the page length wider than the physical width of the phototypesetter roll.

Predefined Strings

**	Footnote number, actually **[\n(*$f**]*. This macro is incremented after each call to **.)f**.
*#	Delayed text number. Actually *[\n(*$d*]*.
*[	Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character **[**. Extra space is left above the line to allow room for the superscript.
*]	Unsuperscript. Inverse to *[. For example, to produce a superscript, you might type **x*[2*]**, which produces **x[2]**.
*<	Subscript. Defaults to < if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.
*>	Inverse to *<
*(dw	The day of the week, as a word.
*(mo	The month, as a word.
*(td	Today's date, directly printable. The date is of the form *April 7, 1984*. Other forms of the date can be used by using **\(dy** (the day of the month; for example, 7, ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, April is 4), and **\n(yr** (the last two digits of the current year).
*(lq	Left quote marks. Double quote in **nroff**.
*(rq	Right quote.
*-	¾ em dash in **troff**; two hyphens in **nroff**.

Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through **-me**. To reference these characters, you must call the macro **.sc** to define the characters before using them.

.sc Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization.

The special characters available are listed below.

Name	Usage		Example
Acute accent	*'	a*'	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*	n*	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Ⓐ
There exists	*(,,,		EXITSTS
For all	*(qa		FORALL

4F

The ME Text-formattin Macros

Introduction

This document describes the text processing facilities available on the UTek operating system through **nroff** and the **me** macro package. It is assumed that you, the reader, are already familiar with the UTek operating system and a text editor such as **ex**. This is intended to be a casual introduction, and is not all material is covered. In particular, many variations and additional features of the **me** macro package are not explained. For a complete discussion of this and other issues, see Section 4E *The ME Reference Guide*, and section 4B *The Nroff/Troff Reference Guide*.

Nroff, a computer program that runs on the UTek operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the **nroff** program telling how to format the printed copy.

The first part describes the basics of text processing. The second describes the basic requests. The third introduces displays. Annotations, such as footnotes, are handled in the fourth part. The more complex requests that are not discussed in the second subsection are covered in the fifth. Finally, the sixth subsection discusses things you need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond the fourth subsection until you have tried some of the basic features.

When you have your text ready, call the **nroff** formatter by typing as a request to the UTek shell:

nroff -me -T*type filenames*

where *type* describes the type of terminal you are outputting to. If the **-T** flag is omitted, a *lowest common denominator* terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the **nroff** command can be found in Section 4B *The Nroff/Troff Reference Guide*.

The word *argument* is used in this manual to mean a word or number that appears on the same line as a request that modifies the meaning of that request. For example, the request

.sp

spaces one line, but

.sp 4

spaces four lines. The number **4** is an *argument* to the **.sp** request, says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

Basics of Text Proc

The primary function of **nroff** is to *collect* words from those words, *justify* the right margin by inserting ext the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...
```

is read, packed onto output lines, and justified to pr

```
Now is the time for all good men to come to t
party.  Four score and seven years ago,...
```

Sometimes you may want to start a new output line i is not yet full (for example, at the end of a paragrapl *break*, which starts a new output line. Some request as do blank input lines and input lines beginning wit

Not all input lines are text to be formatted. Some of describe how to format the text. Requests always h as the first character of the input line.

The text formatter also does more complex things, s pages, skipping over page folds, putting footnotes in

A few hints for preparing text for input to **nroff**: First Short input lines are easier to edit, and **nroff** packs anyway. In keeping with this, begin a new line after phrase, since common corrections are to add or dele Second, do not put spaces at the end of lines, since **nroff** processor. Third, do not hyphenate words at t that should have hyphens in them, such as mother–i hyphenate words for you as needed, but is not smar and join a word back together. Also, words such as broken over a line, since then you could get a space *mother- in-law*.

Headers and Fo

Arbitrary headers and footers requests of the form

.he *title*

and

.fo *title*

define the titles to put at the titles are called *three-part* titl and a right–justified part. To (whatever it may be) is used backslash (\) and double quo is replaced by the current pa the input:

```
.he ''%''
.fo 'Jane Jones''My Boo
```

results in the page number c left corner, and *My Book* in t

Double Spacing

Nroff double spaces output t done in this section. You ca

Page Layout

A number of requests let you called the *layout* of the outpu *white space* (blank lines or sp should be replaced with valu characters that should actual

The **.bp** request starts a r

The request **.sp** *n* leaves *n* li single line) or can be of the f example, the input:

```
.sp 1.5i
My thoughts on the subj
.sp
```

leaves one and a half inches *subject*, followed by a single

4F

The ME Text-formattin Macros

Introduction

This document describes the text processing facilities available on the UTek operating system through **nroff** and the **me** macro package. It is assumed that you, the reader, are already familiar with the UTek operating system and a text editor such as **ex**. This is intended to be a casual introduction, and is not all material is covered. In particular, many variations and additional features of the **me** macro package are not explained. For a complete discussion of this and other issues, see Section 4E *The ME Reference Guide*, and section 4B *The Nroff/Troff Reference Guide*.

Nroff, a computer program that runs on the UTek operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the **nroff** program telling how to format the printed copy.

The first part describes the basics of text processing. The second describes the basic requests. The third introduces displays. Annotations, such as footnotes, are handled in the fourth part. The more complex requests that are not discussed in the second subsection are covered in the fifth. Finally, the sixth subsection discusses things you need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond the fourth subsection until you have tried some of the basic features.

When you have your text ready, call the **nroff** formatter by typing as a request to the UTek shell:

nroff -me -T*type filenames*

where *type* describes the type of terminal you are outputting to. If the **-T** flag is omitted, a *lowest common denominator* terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the **nroff** command can be found in Section 4B *The Nroff/Troff Reference Guide*.

The word *argument* is used in this manual to mean a word or number that appears on the same line as a request that modifies the meaning of that request. For example, the request

.sp

spaces one line, but

.sp 4

spaces four lines. The number **4** is an *argument* to the **.sp** request, says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

Basics of Text Processing

The primary function of **nroff** is to *collect* words from input lines, *fill* output lines with those words, *justify* the right margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...
```

is read, packed onto output lines, and justified to produce:

```
Now is the time for all good men to come to the aid of their
party.  Four score and seven years ago,...
```

Sometimes you may want to start a new output line even though the line you are on is not yet full (for example, at the end of a paragraph). To do this, you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* that describe how to format the text. Requests always have a period or an apostrophe (') as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

A few hints for preparing text for input to **nroff**: First, keep the input lines short. Short input lines are easier to edit, and **nroff** packs words onto longer lines for you anyway. In keeping with this, begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the **nroff** processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as mother–in–law); **nroff** is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as mother–in–law should not be broken over a line, since then you could get a space where not wanted, such as *mother- in-law*.

Basic Requests

Paragraphs

Paragraphs are begun by using the **.pp** request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

produces a blank line followed by an indented first line. The result is:

```
          Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if you typed:

```
.pp
Now is the time for all good men
      to come to the aid of their party.
Four score and seven years ago,...
```

the output would be:

```
      Now is the time for all good men
       to come to the aid of their party.
Four score and seven years ago,...
```

A new line begins after the word *men* because the second line began with a space character.

There are many fancier types of paragraphs, which are described later.

Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form

.he *title*

and

.fo *title*

define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles; that is, there is a left–justified part, a centered part, and a right–justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash (\) and double quotation marks (") should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he ''%''
.fo 'Jane Jones''My Book'
```

results in the page number centered at the top of each page, *Jane Jones* in the lower left corner, and *My Book* in the lower right corner.

Double Spacing

Nroff double spaces output text automatically if you use the request **.ls 2** , as is done in this section. You can revert to single–spaced mode by typing **.ls 1**.

Page Layout

A number of requests let you change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of *white space* (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters that should actually be typed.

The **.bp** request starts a new page.

The request **.sp** *n* leaves *n* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *n***i** (for *n* inches) or *n***c** (for *n* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line *My thoughts on the subject*, followed by a single blank line.

The **.in** +*n* request changes the amount of white space on the left of the page (the *indent*). The argument *n* can be of the form +*n* (meaning leave *n* spaces more than you are already leaving), –*n* (meaning leave less than you do now), or just *n* (meaning leave exactly *n* spaces). *n* can be of the form *n***i** or *nc* also. For example, the input:

initial text
.in 5
some text
.in +1i
more text
.in —2c
final text

produces *some text* indented exactly five spaces from the left margin, *more text* indented five spaces plus one inch from the left margin (fifteen spaces on a standard lineprinter), and *final text* indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```
initial text
        some text
                  more text
          final text
```

The **.ti** +*n* (temporary indent) request is used like **.in** +*n* when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R.  The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

```
Ware, James R.  The Best of Confucius,
          Halcyon House, 1950.
           An excellent book containing translations of
           most of Confucius' most delightful sayings.
           A definite must for anyone interested in the early
           foundations of Chinese philosophy.
```

Text lines can be centered by using the **.ce** request. The line after the **.ce** is centered (horizontally) on the page. To center more than one line, use **.ce** *n* (where *n* is the number of lines to center), followed by the *n* lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines you want to center
.ce 0
```

The **.ce 0** request tells **nroff** to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use **.br**.

Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the **.ce** request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

underlines those eight words in **nroff**. (In **troff** they are set in italics.)

Displays

Displays are sections of text to be set off from the body of the paper. Major quotations, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

Major Quotes

Major quotations are quotations that are several lines long, and hence are set in from the rest of the text without quotation marks around them. These can be generated using the commands **.(q** and **.)q** to surround the quotation mark. For example, the input:

```
As Weizenbaum points out:
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming,...
.)q
```

generates as output:

```
As Weizenbaum points out:

        It is said that to explain is to explain away.
        This maxim is nowhere so well fulfilled
        as in the areas of computer programming,...
```

Lists

A *list* is an indented, single-spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests **.(l** and **.)l**. For example, type:

```
Alternatives to avoid deadlock are:
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

produces:

```
Alternatives to avoid deadlock are:
        Lock in a specified order
        Detect deadlock and back out one process
        Lock all resources needed before proceeding
```

Keeps

A *keep* is a display of lines that are kept on a single page if possible, such as in a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps are not.

Blocks are the basic kind of keep. They begin with the request **.(b** and end with the request **.)b**. If there is no room on the current page for everything in the block, a new page is begun. This sometimes results in blank space at the bottom of the page. When blank space is not appropriate, you can use the alternative, *floating keeps*.

Floating keeps move relative to the text. They are good for things referred to by name, such as "See Figure 3". A floating keep appears at the bottom of the current page if it fits; otherwise, it appears at the top of the next page. Floating keeps begin with the line **.(z** and end with the line **.)z**. For an example of a floating keep, see Figure 4F-1. The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1.  Example of a Floating Keep.
.hl
.)z
```

Figure 4F-1. Example of Floating Keep.

Fancier Displays

Keeps and lists are normally collected in *no-fill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type **.(l F** (throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This kind of display is indented from both margins. For example, the input:

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

is output as:

```
And now boys and girls, a newer, bigger, better toy than ever
before!  Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern data processing
devices.  You too can produce beautifully formatted papers
without even batting an eye!
```

Lists and blocks are also indented (floating keeps are normally left–justified). To get a left–justified list, type **.(l L**. To get a list centered line–for–line, type **.(l C**. For example, to get a filled, left–justified list, enter:

.(l L F
text of block
.)l

The input:

.(l
first line of unfilled display
more lines
.)l

produces the indented text:

first line of unfilled display
more lines

Typing the character **L** after the **.(l** request produces the left–justified result:

first line of unfilled display
more lines

Using **C** instead of **L** produces the line–at–a–time centered output:

first line of unfilled display
more lines

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this, use centered blocks, which are surrounded by the requests **.(c** and **.)c**. All the lines are centered as a unit, so that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

first line of unfilled display
more lines

If the block requests **.(b** and **.)b** had been omitted, the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

Footnotes

Footnotes begin with the request **.(f** and end with the request **.)f**. The current footnote number is maintained automatically, and can be used by typing `\**`, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

```
.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
```

1. Like this.

```
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright
> and at the same time is presumptuous;
> one who is not diligent and at the same time is ignorant;
> one who is untruthful and at the same time is incompetent;
> such men I do not count among acquaintances.[2]

It is important that the footnote appears *inside* the quotation, so that you can be sure that the footnote will appear on the same page as the quote.

Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# for delayed text instead of ** as with footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes. But you may want to reference them with special characters* rather than numbers.

Indexes

An *index* (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after an ellipsis.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request may have a argument, which is the value to print as the *page number*. It defaults to the current page number. If the page number given is an underscore (_) no page number or line of dots is printed at all. To get the line of dots without a page number, type **.)x ""**, which specifies an explicitly null page number.

---

2. James R. Ware. *The Best of Confucius*, Halcyon House, 1950. Page 77.

* Such as an asterisk.

The **.xp** request prints the index.

For example, the input:

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

```
Sealing wax.............................13
Cabbages and kings
Why the sea is boiling hot.....................2.5a
Whether pigs have wings.....................
        This is a terribly long index entry,
        such as might be used for a list of
        illustrations, tables, or figures;
        I expect it to take at least two lines.........13
```

The **.(x** request may have a single character argument, specifying the *name* of the index; the normal index is **x**. Thus, several indexes may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents). The pages may have to be physically rearranged after printing.

Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form *n.n.n* (such as used in this document), and multicolumn output.

More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified, block-style paragraphs by using **.lp** instead of **.pp**, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the **.ip** request. A word specified on the same line as **.ip** is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text . . .
```

produces as output:

```
one  This is the first paragraph.
     Notice how the first line of the resulting
     paragraph lines up with the other
     lines in the paragraph.

two  And here we are at the second paragraph already.
     You may notice that the argument to
     .ip appears in the margin.

We can continue text without starting a new indented
paragraph by using the .lp request.
```

If you have spaces in the label of a **.ip** request, you must use an *unpaddable space* instead of a regular space. This is typed as a backslash character \ followed by a space. For example, to print the label *Part 1*, enter:

.ip "Part\ 1"

If a label of an indented paragraph (that is, the argument to **.ip**) is longer than the space allocated for the label, **.ip** begins a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second
and subsequent lines, although they will line up
with each other.
```

produces:

```
longlabel
        This paragraph had a long label.
        The first character of text on the first line
        will not line up with the text on second
        and subsequent lines, although they will line up
        with each other.
```

It is possible to change the size of the label by using a second argument, which is the size of the label. For example, the above example could be done correctly by saying:

.ip *longlabel* **10**

which makes the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* **ii**. For example, to leave one inch of space for the label, type:

.nr ii 1i

somewhere before the first call to **.ip**.

If **.ip** is used with no argument at all, no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

```
[a]  This is the first paragraph of the example.
     We have seen this sort of example before.

     This paragraph is lined up with the previous paragraph,
     but it has no tag in the margin.
```

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
that are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

```
(1)  This is the first point.

(2)  This is the second point. Points are
     just regular paragraphs that are given
     sequence numbers automatically by the .np
     request.

     This paragraph will reset numbering by .np.

(1)  For example, we have reverted to numbering
     from one now.
```

Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the **.sh** request. You must tell **.sh** the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number **4.2.5** has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

1. The Preprocessor
1.1. Basic Concepts
1.2. Control Inputs
1.2.1.
1.2.2.
2. Code Generation
2.1.1.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

.sh 3 "*Another section*" **7 3 4**

begins the section numbered **7.3.4**; all subsequent **.sh** requests will number relative to this number.

There are more complex features that cause each section to be indented proportionally to the depth of the section. For example, if you enter:

.nr si *n*

each section is indented by an amount *n*. *n* must have a scaling factor attached, that is, it must be of the form *nx*, where *x* is a character telling what units *n* is in. Common values for *x* are **i** for inches, **c** for centimeters, and **n** for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

.nr si 0.5i

After this, sections are indented by one-half inch per level of depth in the section number. For example, this document was produced using the request

.nr si 3n

at the beginning of the input file, giving three spaces of indent per section depth.

Section headers without automatically generated numbers can be done using:

.uh "*Title*"

which does a section heading, but puts no number on the section.

Parts of the Basic Paper

There are some requests which assist in setting up papers. The **.tp** request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(1 C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)1
.bp
```

The request **.th** sets up the environment of the **nroff** processor. It defines the correct headers and footers (a page number in the upper–right corner only), sets the margins correctly, and double spaces.

The **.+c** *T* request can be used to start chapters. Each chapter is automatically numbered from one, and a heading is printed at the top of each chapter with the chapter number and the chapter name *T*. For example, to begin a chapter called *Conclusions*, use the request:

.+c "CONCLUSIONS"

which produces, on a new page, the lines

CHAPTER 5
CONCLUSIONS

with appropriate spacing. Also, the header is moved to the foot of the page on the first page of a chapter.

If the title parameter *T* is omitted from the **.+c** request, the result is a chapter with no heading. This can also be used at the beginning of a paper.

Although papers traditionally have the abstract, table of contents, and so forth at the front of the paper, it is more convenient to format and print them last when using **nroff**. This is so that index entries can be collected and then printed for the table of contents (or whatever). At the end of the paper, issue the **.++** *P* request, which begins the preliminary part of the paper. After issuing this request, the **.+c** request begins a preliminary section of the paper. Most notably, this prints the page number restarted from 1 in lower–case Roman numerals. **.+c** may be used repeatedly to begin different parts of the front material, for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request **.++** *B* may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in Figure 4F–2. (In this figure, comments begin with the sequence \".)

```
.th      \" set for thesis mode
.fo ''DRAFT''      \" define footer for each page
.tp      \" begin title page
.(l C    \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l      \" end centered part
.+c INTRODUCTION  \" begin chapter named "INTRODUCTION"
.(x t    \" make an entry into index `t'
Introduction
.)x      \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"         \" begin another chapter
.(x t    \" enter into index `t' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B    \" begin bibliographic information
.+c BIBLIOGRAPHY  \" begin another `chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P    \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t    \" print index `t' collected above
.+c PREFACE       \" begin another preliminary section
text of preface
```

Figure 4F–2. Outline of a Sample Paper.

Tables

A special UTek program exists to format special types of material. **Tbl** arranges to print good-looking tables in a variety of formats. This document only describes the embellishments to the standard features; consult Section 4G, *Tbl — A Table Formatting Program* for a full description of its use.

The **tbl** program produces tables. Tables begin with the **.TS** request and end with the **.TE** request. Tables are normally kept on a single page. If you have a table that is too big to fit on a single page, begin the table with the request **.TS H** and put the request **.TH** after the part of the table that you want duplicated at the top of every page the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

Two-Column Output

You can get two-column output automatically by using the request **.2c**. This causes everything after it to be output in two-column form. The request **.bc** starts a new column; it differs from **.bp** in that **.bp** may leave a totally blank column when it starts a new page. To revert to single column output, use **.1c**.

Defining Macros

A *macro* is a collection of requests and text that may be used by stating a simple request. Macros begin with the line **.de** *xx* (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line *.xx* is the same as stating all the other lines. For example, to define a macro that spaces three lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
..
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names can be one or two characters. In order to avoid conflicts with names in **me**, always use upper–case letters as names. The only names to avoid are **TS**, **TH**, **TE**, **EQ**, and **EN**.

Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a *list of figures* you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you hope will give you a figure with a label and an entry in the index *f* (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string \! at the beginning of all the lines dealing with the index. In other words, you should use:

.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x *f*
\!Figure 5
\!.)x
.)z

which defers the processing of the index until the figure is output. This guarantees that the page number in the index is correct. The same comments apply to blocks (with **.(b** and **.)b**) as well.

Troff and the Phototypesetter

With a little care, you can prepare documents that print nicely on either a regular terminal or when phototypesetting using the **troff** formatting program.

Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Times Roman. Text which would be underlined in **nroff** with the **.ul** request is set in italics in **troff**.

There are ways of switching between fonts. The requests **.r**, **.i**, and **.b** switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

.i *word*

which will set *word* in italics but does not affect the surrounding text. In **nroff**, italic and bold text are underlined.

Notice that if you are setting more than one word in a font, you must surround that word with quotation marks (") so that it appears to the **nroff** processor as a single word. The quotation marks do not appear in the formatted text. If you want a quotation mark to appear, you should quote the entire string (even if a single word), and use *two* quotation marks where you want one to appear. For example, if you want to produce the text:

"Master Control"

in italics, you must type:

.i """Master Control\¦"""

The \¦ produces a very narrow space so that the *l* does not overlap the quote sign in **troff**, like this:

"Master Control"

There are also several *pseudofonts* available:

- **.u** Underlines the text that follows it on an input line
- **.bi** Creates a word (or words) that is both boldface and italic
- **.bx** Places the words that follow it on the input line in a box

In **nroff** these all just underline the text. Notice that pseudofont requests set only the single parameter in the pseudofont; ordinary font requests begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way **troff** justifies text. For example, if you were to issue the requests:

.bi *"some bold italics"*
and
.bx *"words in a box"*

in the middle of a line, **troff** would produce words in a special bold and italic font, and words encased in a box. This would not look good in the middle of text.

The second parameter of all font requests is set in the original font. For example, the font request:

.b bold face

generates *bold* in bold font, but sets *face* in the font of the surrounding text, resulting in:

bold *face*

To set the two words *bold* and *face* both in boldface, type:

.b "bold face"

You can mix fonts in a word by using the special sequence **\c** at the end of a line to indicate *continue text processing*. This allows input lines to be joined together without a space in between them. For example, the input:

.b bold \c
.i italics

generates **bold***italics*. But if we had typed:

.b bold
.i italics

the result would have been

bold *italics*

as two separate words.

Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the point size, type:

.sz +*n*

where *n* is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

NOTE

Changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotations, rather than the double quotation character ("). It looks better to use grave and acute accents; for example, compare "quote" to ``quote''.

In order to make quotations compatible between the typesetter and terminals, you may use the sequences *(**lq** and *(**rq** to stand for the left and right quotation mark, respectively. These both appear as " on most terminals, but are typeset as `` and '' respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

```
"Some things aren't true even if they did happen."
```

As a shorthand, the special font request:

.q *"quoted text"*

generates ``quoted text''. Notice that you must surround the material to be quoted with double quotation marks if it is more than one word.

Tbl — A Table Formatting Program

Introduction

The **tbl** program is a document formatting preprocessor for the **nroff** and **troff** formatters that makes fairly complex tables easy to specify and enter. Tables consist of columns that can be independently centered, right–aligned, left–aligned, or aligned by decimal points. It can also:

- place headings over single columns or groups of columns
- contain text
- draw horizontal or vertical lines in the table
- enclose a table or its subparts in a box

The **tbl** program translates your input into a list of **nroff/troff** formatter requests that produce the table. The **tbl** program isolates a portion of the job that it can successfully handle (text between the **.TS** and **.TE** macros), and leaves the remainder for other programs.

Usage

On the UTek system, the **tbl** program can be run on a sample table with the command:

tbl *filename* ¦ **troff**

When you have several input files containing tables and **mm** macros requests, the normal command is:

tbl *file1 file2* ¦ **troff –mm**

The usual options can be used on the **troff** formatter. Using the **nroff** formatter is similar to that of **troff**. Instead of a filename you can enter — and the standard input is read.

If you use a line printer without adequate driving tables or post–filters, there is a special **–TX** option to **tbl**. This option produces output that does not have fractional line motions.

The **tbl** program accepts up to 35 columns; the actual number that can be processed may be smaller depending on the availability of **troff** formatter number registers. Names of number registers used by **tbl** must be avoided within tables.

These include two–digit numbers from 31 to 99 and strings of the form 4*x*, 5*x*, #*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 (discussed later) share a register. Therefore, you cannot use them in the same column.

As an aid in writing layout macros, **tbl** defines a number register TW that defines the table width. The TW number register is defined by the time that the **.TE** macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multipage boxed tables, the macro **T#** is defined to produce the bottom lines and side lines of a boxed table. By using this macro in the page footer, a multipage table can be boxed. In particular, the **mm** macros can be used to print a multipage boxed table with a repeated heading by giving the argument *H* to the **.TS** macro. If the table start macro is written:

```
.TS H
```

then, a line of the form:

```
.TH
```

must be given in the table after any table heading. If there are no table headings, place it at the start of the table. Material up to the **.TH** is placed at the top of each page of the table. The remaining lines in the table are placed on several pages as required. This is not a feature of **tbl**, but of the **mm** macros.

Input Commands

Input to **tbl** is text for a document, including the data you want to put in tables. The table portion begins with the command **.TS** and ends with the command **.TE**. The **tbl** program processes the tables, generates table formatting requests, and leaves the text outside **.TS** and **.TE** unchanged. The text and the **tbl** commands within the table are expanded into **troff** formatter layout codes.

The general format of the input is:

text
.TS
table
.TE
text

The format of each *table* is:

.TS
options;
format.
data
.TE

Each table is independent and contains:

- Global options
- A format section describing columns and rows of the table
- Data to fill in the table

These three items are discussed in the following sections. The format and data sections are always required, and the global options are optional.

Global Options

The single line of options can affect the whole table. If present, the options line must immediately follow the **.TS** line. It contains a list of option names separated by spaces, tabs, or commas and ends in a semicolon. Available options include:

- **center** — center table (default is left–aligned)
- **expand** — make table as wide as current line length
- **box** — enclose table in a box (see figure 2–1)
- **allbox** — enclose each item of table in a box (see figure 2–2)
- **doublebox** — enclose table in two boxes
- **tab** (*x*) — separate data items by using *x* instead of <TAB>
- **linesize** (*n*) — set lines or rules in *n*–point type

The **tbl** program tries to keep boxed tables on one page by issuing the appropriate **.ne** (need) requests. These requests are calculated from the number of lines in the tables. If there are spacing requests embedded in the input, the **.ne** requests may be inaccurate. Normal **troff** formatter procedures, such as keep–release macros, are used in that case. If a multipage boxed table is required, macros designed for this purpose (**.TS H** and **.TH**) should be used.

INPUT:

```
.TS
box;
ccc
lll.
Language(T)Authors(T)Runs on
.sp
Fortran(T)Many(T)Almost anything
C(T)BTL(T)11/45,H6000,370
BLISS(T)Carnegie-Mellon(T)PDP-10,11
IDS(T)Honeywell(T)H6000
Pascal(T)Stanford(T)370
.TE
```

OUTPUT:

Language	Authors	Runs on
Fortran	Many	Almost anything
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Figure 4G-1. Table Using box Option.

INPUT:

```
.TS
allbox;
css
ccc
nnn.
AT&T Common Stock
Year⒯Price⒯Dividend
1971⒯41-54⒯$2.60
2⒯41-54⒯2.70
3⒯46-55⒯2.87
4⒯40-53⒯3.24
5⒯45-52⒯3.40
6⒯51-59⒯.95*f
.TE
*(first quarter only)
```

OUTPUT:

AT&T Common Stock

Year	Price	Dividend
1971	41–54	$2.60
2	41–54	2.70
3	46–55	2.87
4	40–53	3.24
5	45–52	3.40
6	51–59	.95*f

*(first quarter only)

Figure 4G–2. Table Using allbox Option.

Format Section

The format section of the table specifies the layout of the columns. Each line in the format section corresponds to one line of table data (except the last format line, which corresponds to all following data lines up to any **.T&** command line). Each format line contains a *key letter* for each column of the table. Key letters can be separated by spaces or tabs for readability purposes. Key letters for column entries include:

L *or* **l** Indicates a left-aligned column entry.

R *or* **r** Indicates a right-aligned column entry.

C *or* **c** Indicates a centered column entry.

N *or* **n** Indicates a numerical column entry. Numerical entries are aligned so that the units digits of numbers line up.

A *or* **a** Indicates an alphabetic subcolumn. All corresponding entries are aligned on the left and positioned so that the widest entry is centered within the column.

S *or* **s** Indicates a spanned heading. The entry from the previous column continues across this column (not allowed for the first column of the table).

^ Indicates a vertically spanned heading. The entry from the previous row continues down through this row (not allowed for the first row of the table).

When numerical column alignment (**n**) is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point. If there is no dot adjoining a digit, the rightmost digit is used as a unit digit. If no alignment is necessary, the item is centered in the column. However, you can use the special nonprinting character string \& to override decimal points and digits or to align alphabetic data. This aligns the decimal points and the \& disappears from the final output.

In Example 4G–1, items shown in the INPUT column are aligned numerically as shown in the OUTPUT column:

INPUT	OUTPUT
`.TS`	
`center;`	
`n.`	
`13`	13
`4.2`	4.2
`26.4.12`	26.4.12
`abcdefg`	abcdefg
`abcd\&efg`	abcd\&efg
`abcdefg\&`	abcdefg\&
`43\&3.22`	43\&3.22
`749.12`	749.12
`.TE`	

Example 4G–1. Numerically Aligned Table.

If you use numerical data in the same column with **L** (the capital L is used instead of lowercase for readability) or **r** table type entries, the widest number is centered relative to the wider **L** or **r** items. Alignment within numerical items is preserved. This is similar to the behavior of **a** type data. Alphabetic subcolumns (requested by the **a**) 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.

NOTE

The **n** *and* **a** *items should not be used in the same column.*

The end of the format section is indicated by a period (.). The layout of key letters in the format section resembles the layout of the actual data in the table. Thus, a simple three-column format might appear as:

```
css
lnn.
```

The first line of the table contains a heading centered across all three columns. Each remaining line contains a left-aligned item in the first column, followed by two columns of numerical data. Example 4G-2 illustrates a table in this format:

OVERALL	FORMAT	
Item-a	34.22	9.1
Item-b	12.65	.02
Item-c	23	5.8
Total	69.87	14.92

Example 4G-2. A Table Using Simple Three-Column Format.

Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be give on the same line, separated by commas. The format for the above example could be written:

```
css,lnn.
```

The order of the features is not important. You do not need to separate them with spaces, except as indicated to avoid ambiguities involving point size and font changes. So a numerical column entry in italic font and 12-point type with a minimum of 2.5 inches and separated by 6 ens from the next column could be specified as:

```
np12w(2.5i)fl 6
```

Additional features of the key letter system are:

- **Horizontal lines** — An underscore (_) indicates a horizontal line in place of the column entry; an equal sign (=) indicates a double horizontal line. If an adjacent column contains a horizontal line or if there are vertical lines adjoining this column, the horizontal line is extended to meet nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed. See Figure 4G–3.
- **Vertical lines** — A vertical bar (|) placed between column key letters causes a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key letters, a double vertical line is drawn. See Figure 4G–4.

INPUT:

```
.TS
box;
LLL
LL_
LL|LB
LL_
LLL.
january⊤february⊤march
april⊤may
june⊤july⊤Months
august⊤september
october⊤november⊤december
.TE
```

OUTPUT:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Figure 4G–3. Table Using Horizontal Lines in Place of Key Letters.

INPUT:

```
.TS
box;
css
c|c|c|
l|l|n.
Major New York Bridges
_
Bridge⓪TDesigner⓪TLength
_
Brooklyn⓪TJ.A. Roebling⓪T1595
Manhattan⓪TG. Lindenthal⓪T1470
Williamsburg⓪TL.L. Buck⓪T1600
_
Queensborough⓪TPalmer⓪T1182
⓪THornbostel
_
⓪T⓪T1380
Triborough⓪TO.H. Ammann⓪T_
⓪T⓪T383
_
Bronx Whitestone⓪TO.H. Ammann⓪T2300
Throgs Neck⓪TO.H. Ammann⓪T1800
.TE
```

OUTPUT:

Major New York Bridges

Bridge	Designer	Length
Brooklyn	J.A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L.L. Buck	1600
Queensborough	Palmer Hornbostel	1182
Triborough	O.H. Ammann	1380 383
Bronx Whitestone	O.H. Ammann	2300
Throgs Neck	O.H. Ammann	1800

Figure 4G-4. Table Using vertical bar Key Letter Feature.

- **Space between columns** — A number can follow the key letter, indicating the amount of separation between this column and the next column. The number specifies the separation in *ens*. One *en* is about the width of the letter n. More precisely, an *en* is the number of points (1 point = 1/72 inch) equal to half 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, the largest space requested becomes the default.
- **Vertical spanning** — Items spanned 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 begins at the top line of its range.
- **Font changes** — A key letter, followed by the letter **f** or **F**, then a string containing the font name or number, indicates that the corresponding column should be in a different font from the default. All font names are one or two letters. A one–letter font name is 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 requests given with the table entries overrides these specifications.
- **Point size changes** — A key letter followed by **p** or **P** and a number indicates the point size of the corresponding table entries. If the number is a signed digit, it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.
- **Vertical spacing changes** — A key letter followed by **v** or **V** and a number indicates the vertical line spacing used within a multi–line table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block.
- **Column width indication** — A key letter followed by **w** or **W** and a width value in parentheses indicates minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, is width is used. The width is also used as a default line length for included text blocks. You can use the **troff** units **c** (centimeters) and **i** (inches) to scale the width value. The default value is *ens* if none are used. If the width specification is a single integer, the parentheses can be omitted. If another width value is given in a column, the last one controls the width.
- **Equal–width columns** — A key letter followed by **e** or **E** indicates equal–width columns. All columns whose key letters are followed by **e** or **E** are made the same width. This permits 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 can be useful in allowing headings to run across adjacent columns where spanned headings are appropriate.
- **Default** — Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table. Extra columns in the data are ignored.

Data to be Printed

Data for the table is input after the format section. Each table line is typed as one line of data. Very long input lines can be broken. Any line whose last character is a backslash (\) is combined with the following line, that is, the backslash joins the lines and is not output. Data for different columns (table entries) are separated by tabs or by whatever character has been specified in the **tab** global option. See the earlier discussion on *Global Options*.

There are a few special cases of when you are entering the data for the table:

- **troff commands within tables** — An input line beginning with a dot and followed by anything but a number (.*xx*) is assumed to be a request to the formatter and is passed through unchanged retaining its position in the table. For example, a blank line within a table can be produced by putting the **.sp** command in the data.
- **Full with horizontal lines** — An input line containing only the underscore character (_) or the equal sign (=) is taken to be a single or double line, respectively, extending the full width of the table.
- **Single column horizontal lines** — An input table entry containing only the underscore character or the equal sign is taken to be a single or double line extending the full width of the column. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, precede them with a \& or follow them with a space, before the usual tab or newline character.
- **Short horizontal lines** — An input table entry containing only the string _ produces 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 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 of ^.

- **Text blocks** — In order to include a block of text as a table entry, precede it by **T{** and follow it with **T}** on lines by themselves. So if you cannot conveniently type a string between tabs, you can make a single table entry by framing it in these commands. You can enter following data on the **T}** line, if you precede that data with a tab character. These text blocks are pulled out from the table, processed separately by the formatter, and replaced in the table as a solid block. See Figure 4G–5.

INPUT:

```
.TS
allbox;
cfI ss
cw(1i)cw(1.75i)cw(1.75i)
lll.
New York Area Rocks
.sp
EraⓉFormationⓉAge (years)
PrecambrianⓉReading ProngⓉ>1 billion
PaleozoicⓉManhattan ProngⓉ400 million
MesozoicⓉt{
Neward Basin, incl.
Stockton,Lockatong, and Brunswick
formations
.ad
T}Ⓣ200 million
CenozoicⓉCoastal PlainⓉ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	Neward Basin, incl. Stockton,Lockatong, and Brunswick formations	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation

Figure 4G–5. Table Using Text Blocks.

Various limits on the **troff** program are likely to be exceeded if 30 or more text blocks are used in a table. This produces diagnostic messages like *too many string/macro names* or *too many number registers.*

If no line length is specified in the block of text or in the table format, the default is to use:

$$L \times C / (N + 1)$$

In this case, 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, and so on) used in typesetting this block of text are:

- those in effect at the beginning of the table (including the effect of the **.TS** macro
- any table format specifications of size, spacing, and font using the **p**, **v**, and **f** modifiers to the column key letters
- **troff** requests within the text block itself (requests within the table data, but not within the text block, do not affect that block)

Although any number of lines can be present in a table, only the first 200 lines are used in setting up the table. See Figure 4G–6 A multipage table should be arranged as several single–page tables if this proves to be a problem.

When calculating column widths, all table entries are assumed to be in the font and size being used when the **.TS** command was encountered. This is true except for font and size changed indicated in the table format section or within the table data (as in the entry **\s+3Data\s0**). Because arbitrary **troff** requests can be sprinkled in a table, care must be taken to avoid confusing width calculations. It is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made in equal width.

Additional Command Lines

If you want to change the format of a table in the middle, perhaps to make subheadings or summaries, use the **.T&** command to change column parameters. It is not recognized after the first 200 lines of a table.

The overall format of such a table is:

.TS
options;
format.
data
.T&
format.
data
.TE

INPUT:

```
.TS
box;
cfB sss.
Composition of Foods
_
.T&
c|css
c|css
c|c|c|c.
Food(T)Percent(T)Weight
\^(T)_
\^(T)Protein(T)Fat(T)Carbo-
\^(T)\^(T)\^(T)hydrate
_
.T&
l|n|n|n.
Apples(T).4(T).5(T)13.0
Halibut(T)18.4(T)5.2(T)...
Lima beans(T)7.5(T).8(T)22.0
Milk(T)3.3(T)4.0(T)5.0
Mushrooms(T)3.5(T).4(T)6.0
Rye bread(T)9.0(T).6(T)52.7
.TE
```

OUTPUT:

Composition of Foods

Food	Percent		
	Protein	Fat	Carbo-hydrate
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

Figure 4G–6. Table Using Additional Command Lines.