System 8000™
ZEUS Utilities Manual

Ø3-325Ø-Ø1

May, 1983

ZEUS UTILITIES MANUAL

Software Release 3.2

10/14/83

Preface

This manual and the related manuals below provide the complete technical documentation for the System 8000 and the ZEUS operating system.

Title	Zilog Part Number
Zeus Software Documents:	
Zeus Reference Manual	03-3255
Zeus Languages/Programming Tools Manual	03-3249
Zeus Utilities Manual	03-3250
Zeus Administrator Documents:	
Zeus Administrator Manual (Model 11)	03-3254
Zeus Administrator Manual (Model 21/31)	03-3246
System 8000 Hardware Documents:	
System 8000 Hardware Reference Manual (Model 11)	03-3227
System 8000 Hardware Reference Manual (Model 21/31)	03-3237

TABLE OF CONTENTS

SECTION 1
INTRODUCTION TO ZEUS UTILITIES

This volume contains manuals and tutorials describing the basic non-programming utility programs of ZEUS. For a description of the Language and Programming utilities, refer to the ZEUS Languages/Programming Tools Manual (part number 03-3249).

The entries in this volume are:

1.1. Utilities Sections by Topic

Introductory Material

Intro -- An Introduction to the ZEUS System
ZEUS for Beginners -- A Basic Introduction

The Shells

Cshell -- The C Shell from UC Berkeley
Shell -- The Bourne Shell from Bell Laboratories

Interactive Editors

Ed -- The Basic line editor
Ex -- An expanded line editor
Vi -- The visual mode of the Ex editor

Non-interactive Editors

Awk -- A pattern scanning and processing language
Sed -- A non-interactive stream editor

Text Formatters

Nroff -- For terminal and line printer output
Troff -- For CAT phototypesetter output

Text Formatting Macro Packages

Me -- The package from UC Berkeley
Mm -- A package from Bell Laboratories
Ms -- A package from Bell Laboratories

Text Formatting Pre-processors

Tbl -- For formatting tables

Other Extended Programs

Acct -- The system accounting package
Comm -- The Zeus communications package
Fsck -- A File System Checking package
Learn -- Computer aided learning program
Sccs -- Source Code Control System package
Uucp -- Unix to Unix Communication package

Basic ZEUS Interaction

ZEUS for Beginners describes the basics of logging in, running programs, creating and modifying files, etc.

Learn is an computer-aided instruction program for practice in using ZEUS.

The ZEUS mechanism for running programs is itself a user program called a shell. Commonly used under ZEUS is csh, described in **An Introduction to the C Shell.** An alternative is sh (known simply as The Shell, or The Bourne Shell); it is described in **The ZEUS Shell.**

Text Entry and Editing

There are three utilities for maintaining text files. They are the command-line oriented editors ed, and ex and the screen oriented editor vi. They are described in **The ZEUS Line-oriented Text Editor, ed; The Ex Reference Manual** and **Introduction to Display Editing with vi.**

Text Formatters and Macro Packages

Troff is a macro-oriented typesetting program; nroff approximates troff on typewriter-like devices. The **Nroff/Troff Reference Manual** describes these programs. These text processing programs are used with a macro packages such as those described in **Typing Documents on the ZEUS System using the -ms Macros with Troff and Nroff, Writing Papers with Nroff using Me,** and the **MM - Memorandum Macros.**

A Troff Tutorial describes problems of typesetting documents. **Tbl -- A Program to Format Tables** provides an introduction to creating tables with Nroff.

Non-Interactive Editors

SED: A Non-interactive Text Editor describes a program which edits input of indefinite length; commands are similar to those of ed.

AWK: A Pattern Scanning and Processing Language describes a stream editor with a powerful command language.

Inter-System Communications

ZEUS Communication Package describes a communications path between ZEUS and remote systems.

UUCP Installation describes a program that links to other ZEUS systems (or any other system that can run UUCP) via tty port-to-port connections or transient telephone connections.

File System Integrity

File System Integrity Program (FSCK) Reference Manual describes how file systems can be protected against corruption upon reboot.

Source Code Control

Source Code Control System (SCCS) describes a method of controlling the various versions of a file. Each time a change is made to the file, the changes are recorded so that any version of the file since its creation can be reconstructed.

ZEUS Accounting System

The **ZEUS Accounting System** provides a method to collect information about the system; who uses it, what gets used, and how much.

THE PWB/ZEUS ACCOUNTING SYSTEM

The information in this section is based on an article originally written by Henry S. McCreary of Bell Laboratories.

Table of Contents

SECTION 1
THE PWB/ZEUS ACCOUNTING SYSTEM

1.1. Abstract

The PWB Accounting System provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C programs and shell procedures is provided to reduce this accounting data into summary files and reports. This memorandum describes the structure, implementation, and management of this accounting system.

1.2. Introduction

The PWB/ZEUS accounting system was originally designed by John Mashey. Several modifications and additions have been made to make the system easier to manage, and to make it less susceptible to corrupted data or system errors. The following list is a synopsis of the actions of the accounting system:

- At process termination the **ZEUS** Kernal writes one record per process in /usr/adm/pacct in the form of acct.h. See Attachment 2 for a description of data files.

- The **login** and **init** programs record connect sessions by writing records into /usr/adm/wtmp. Date changes, reboots, and shutdowns are also recorded in this file.

- The disk utilization program **acctdusg**, breaks down disk usage by login.

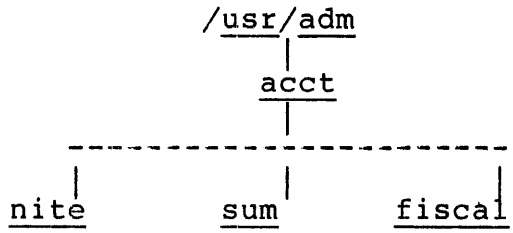
- Fees for file restores, etc, can be charged to specific logins with the **chargefee** shell procedure.

- Each day the **runacct** shell procedure is executed via **cron** to reduce accounting data, produce summary files and reports. See Attachment 3 for a sample report output.

- The **monacct** procedure can be executed on a monthly or fiscal period basis. It saves and restarts summary files, generates a report, and cleans up the sum directory. These saved summary files could be used to charge users for ZEUS usage.

1.3. Files and Directories

The /usr/lib/acct directory contains all of the C programs and shell procedures necessary to run the accounting system. The **adm** login (UID 6) is used by the accounting system and has the following directory structure:



The /usr/adm directory contains the active data collection files. For a complete explanation of the files used by the accounting system, see Attachment 1. The nite directory contains files that are re-used daily by the **runacct** procedure. The sum directory contains the cumulative summary files updated by **runacct**. The fiscal directory contains periodic summary files created by **monacct**.

1.4. Daily Operation

When ZEUS is switched into multi-user mode, **/usr/lib/acct/startup** is executed which does the following:

- The **acctwtmp** program adds a "boot" record to /usr/adm/wtmp. This record is signified by using the system name as the login name in the wtmp record.
- Process accounting is started via **turnacct**. **Turnacct on** executes the **accton** program with the argument /usr/adm/pacct.
- The **remove** shell procedure is executed to cleanup the saved pacct and wtmp files left in the sum directory by **runacct**.

The **ckpacct** procedure is run via **cron** every hour of the day to check the size of /usr/adm/pacct. If the file grows past 1000 blocks (default), **turnacct switch** is executed. While **ckpacct** is not absolutely necessary, the advantage of having several smaller pacct files becomes apparent when trying to restart **runacct** after a failure processing these records.

The **chargefee** program can be used to bill users for file restores, etc. It adds records to /usr/adm/fee which are picked up and processed by the next execution of **runacct** and

merged into the total accounting records.

Runacct is executed via **cron** each night. It processes the active accounting files, /usr/adm/pacct?, /usr/adm/wtmp, /usr/adm/acct/nite/disktacct, and /usr/adm/fee. It produces command summaries and usage summaries by login.

When the system is shut down using **/etc/down**, the **shutacct** shell procedure is executed. It writes a shutdown reason record into /usr/adm/wtmp and turns process accounting off.

This is also executed in /etc/rc_csh before accounting is started is case the system was not brought down using **/etc/down.**

The system administrator can execute **/usr/lib/acct/prdaily** to print the previous day's accounting report.

1.5. Setting up the Accounting System

In order to automate the operation of this accounting system, several things need to be done:

1. If not already present, add this line to the /etc/rc_csh file after the line that zeros out the utmp file.

   /bin/su - adm -c /usr/lib/acct/startup

2. If not already present, add this line to **/etc/down** to turn off the accounting before the system is brought down:

   /usr/lib/acct/shutacct

   This should also be added to /etc/rc_csh right after file systems are mounted.

3. Three entries should be made in /usr/lib/crontab so that **cron** will automatically start some shell procedures.

```
Ø 4 * * 1-6 /bin/su - adm -c
"/usr/lib/acct/runacct  2>  /usr/adm/acct/nite/fd2log"

Ø 2 * * 4 /bin/su - adm -c "/usr/lib/acct/sdisk"

5 * * * * /bin/su - adm -c "/usr/lib/acct/ckpacct"
```

4. The PATH shell variable in **adm's** .cshrc should be set to:

   PATH=/usr/lib/acct:/bin:/usr/bin

5. Make an entry in the /etc/passwd file for user "adm". This user has to have uid of 6 and belong to group 4.

6. Make an entry in the /etc/group file for the "adm" group. This group has a gid of 4. "zeus" and "daemon" need to belong to this group.

   example: "adm::4:zeus, adm, daemon"

1.6. Runacct

Runacct is the main daily accounting shell procedure. It is normally initiated via **cron** during non-prime time hours. **Runacct** processes connect, fee, disk, and process accounting files. It also prepares daily and cumulative summary files for use by **prdaily** or for billing purposes. The following files produced by **runacct** are of particular interest.

nite/lineuse — Produced by **acctcon1**, which reads the wtmp file, and produces usage statistics for each terminal line on the system. This report is especially useful for detecting bad lines. If the ratio between the number of logoffs to logins exceeds about 3/1, there is a good possibility that the line is failing.

nite/daytacct — This file is the total accounting file for the previous day in tacct.h format.

sum/tacct — This file is the accumulation of each day's nite/daytacct, which can be used for billing purposes. It is restarted each month or fiscal by the **monacct** procedure.

sum/daycms — Produced by the **acctcms** program, it contains the daily command summary. The ASCII version of this file is nite/daycms.

sum/cms — The accumulation of each day's command summaries. It is restarted by the execution of **monacct**. The ASCII version is nite/cms.

sum/loginlog Produced by the **lastlogin** shell procedure, it maintains a record of the last time each login was used.

sum/rprt.MMDD Each execution of **runacct** saves a copy of the output of **prdaily**.

Runacct takes care not to damage files in the event of errors. A series of protection mechanisms are used that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that **runacct** can be restarted with minimal intervention. It records its progress by writing descriptive messages into the file active. Files used by **runacct** are assumed to be in the nite directory unless otherwise noted.

All diagnostic output during the execution of **runacct** is written into fd2log. To prevent multiple invocations, in the event of two crons or other problems, **runacct** will complain if the files lock and lock1 exist when invoked. The lastdate file contains the month and day **runacct** was last invoked, and is used to prevent more than one execution per day.

If **runacct** detects an error, a message is written to /dev/console, mail is sent to **root** and **adm**, the locks are removed, diagnostic files are saved, and execution is terminated.

In order to allow **runacct** to be restartable, processing is broken down into separate reentrant states. This is accomplished by using a **case** statement inside an endless **while** loop. Each state is one case of the **case** statement. A file is used to remember the last state completed. When each state completes, statefile is updated to reflect the next state. In the next loop through the **while**, statefile is read and the **case** falls through to the next state. When **runacct** reaches the **CLEANUP** state, it removes the locks and terminates. States are executed in the following order:

SETUP The command **turnacct switch** is executed. The process accounting files, /usr/adm/pacct?, are moved to /usr/adm/Spacct.MMDD. The /usr/adm/wtmp file is moved to /usr/adm/acct/nite/wtmp.MMDD with the current time added on the end.

WTMPFIX The wtmp file in the nite directory is checked for correctness by the **wtmpfix** program. Some date changes will cause **acctcon1** to fail, so **wtmpfix** attempts to adjust the

time stamps in the wtmp file if a date change record appears.

CONNECT Connect session records are written to ctmp in the form of ctmp.h. The lineuse file is created, and the reboots file is created showing all of the boot records found in the wtmp file.

CONNECT Ctmp is converted to ctacct.MMDD which are connect accounting records. Accounting records are in tacct.h format.

PROCESS The **acctprc1** and **acctprc2** programs are used to convert the process accounting files, /usr/adm/Spacct.MMDD, into total accounting records in ptacct?MMDD. The Spacct and ptacct files are correlated by number so that if **runacct** fails, the unnecessary reprocessing of Spacct files will not occur. One precaution should be noted; when restarting **runacct** in this state, remove the last ptacct file because it will not be complete.

MERGE Merge the process accounting records with the connect accounting records to form daytacct.

FEES Merge in any ASCII tacct records from the file fee into daytacct.

DISK On the day after the **sdisk** procedure runs, merge disktacct with daytacct.

MERGETACCT Merge daytacct with sum/tacct, the cumulative total accounting file. Each day, daytacct is saved in sum/tacctMMDD, so that sum/tacct can be recreated in the event it becomes corrupted or lost.

CMS Merge in today's command summary with the cumulative command summary file sum/cms. Produce ASCII and internal format command summary files.

USEREXIT Any installation dependent (local) accounting programs can be included here.

CLEANUP Clean up temporary files, run **prdaily** and save its output in sum/rprtMMDD, remove the locks, then exit.

1.7. Recovering from Failure

The **runacct** procedure can fail for a variety of reasons; usually due to a system crash, /usr running out of space, or a corrupted wtmp file. If the activeMMDD file exists, check it first for error messages. If the active file and lock files exist, check fd2log for any mysterious messages. The following are error messages produced by **runacct**, and the recommended recovery actions:

ERROR: locks found, run aborted

The files lock and lock1 were found. These files must be removed before runacct can restart.

ERROR: acctg already run for date :
check /usr/adm/acct/nite/lastdate

The date in lastdate and today's date are the same. Remove lastdate.

ERROR: turnacct switch returned rc=?

Check the integrity of **turnacct** and **accton**. The **accton** program must be owned by **root**, and have the setuid bit set.

ERROR: Spacct?.MMDD already exists
file setups probably already run

Check status of files, then run setups manually.

ERROR: /usr/adm/acct/nite/wtmp.MMDD already exists,
run setup manually

Self-explanatory.

ERROR: wtmpfix errors see /usr/adm/acct/nite/wtmperror

Wtmpfix detected a corrupted wtmp file. Use **fwtmp** to correct the corrupted file.

ERROR: connect acctg failed: check /usr/adm/acct/nite/log

The **acctcon1** program encountered a bad wtmp file. Use **fwtmp** to correct the bad file.

ERROR: Invalid state, check /usr/adm/acct/nite/active

The file, statefile, is probably corrupted. Check statefile and read active before restarting.

1.8. Restarting runacct

Runacct called without arguments assumes that this is the first invocation of the day. The argument MMDD is necessary if **runacct** is being restarted, and specifies the month and day for which **runacct** will rerun the accounting. The entry point for processing is based on the contents of statefile. To override statefile, include the desired state on the command line.
Examples:

To start **runacct**:
nohup runacct 2> /usr/adm/acct/nite/fd2log&

To restart **runacct**:
nohup runacct 0601 2> /usr/adm/acct/nite/fd2log&

To restart **runacct** at a specific state:
nohup runacct 0601 WTMPFIX 2> /usr/adm/acct/nite/fd2log&

1.9. Fixing Corrupted Files

Unfortunately, this accounting system is not entirely foolproof. Occasionally a file will become corrupted or lost. Some of the files can simply be ignored or restored from the filesave backup. However, certain files must be fixed in order to maintain the integrity of the accounting system.

The wtmp files seem to cause the most problems in the day to day operation of the accounting system. When the date is changed when ZEUS is in multi-user mode, a set of date change records is written into /usr/adm/wtmp. The **wtmpfix** program is designed to adjust the time stamps in the wtmp records when a date change is encountered. Some combinations of date changes and reboots, however, will slip through **wtmpfix** and cause **acctcon1** to fail. The following steps show how to patch up a wtmp file.

cd /usr/adm/acct/nite

fwtmp < wtmp.MMDD **> xwtmp**

vi xwtmp

(delete corrupted records or
delete all records from the
beginning up to the date change)

fwtmp -ic < xwtmp > wtmp.MMDD

If the wtmp file is beyond repair, create a null wtmp file. This will prevent any charging of connect time. **Acctprc1** won't be able to determine which login owned a particular process, but it will be charged to the login that is first in the password file for that userid.

If the installation is using the accounting system to charge users for system resources, the integrity of sum/tacct is quite important. Occasionally, mysterious tacct records will appear with negative numbers, duplicate userids, or a userid of 65535. First check sum/tacctprev with **prtacct**. If it looks ok, the latest sum/tacct.MMDD should be patched up, then recreate sum/tacct. A simple patchup procedure is:

```
cd /usr/adm/acct/sum

acctmerg -v < tacct.MMDD > xtacct

vi xtacct

   (  remove the bad records
    write duplicate uid records
    to another file  )

acctmerg -i < xtacct > tacct.MMDD

acctmerg tacctprev < tacct.MMDD > tacct
```

Remember that the **monacct** procedure removes all the tacct.MMDD files; therefore, sum/tacct can be recreated by merging these files together.

1.10. Editing the Holidays File

Every year on the day after Christmas, the following message will appear in **log**:

```
*** EDIT /usr/lib/acct/holidays with NEW HOLIDAYS
```

Edit /usr/lib/acct/holidays, change the year in the first line to the new year. Change the holiday dates to the new ones for the year. The file must end with a '-1' as the last line.

1.11. Summary

The PWB accounting system was designed from a ZEUS system administrator's point of view. Every possible precaution

has been taken to ensure that the system will run smoothly and without error. It is important to become familiar with the C programs and shell procedures. The manual pages should be studied, and it is advisable to keep a printed copy of the shell procedures handy. This accounting system should be easy to maintain, provide valuable information for the administrator, and provide accurate breakdowns of the usage of system resources for charging purposes.

APPENDIX A
ATTACHMENT 1

Files in the /usr/adm Directory

diskdiag	diagnostic output during the execution of disk accounting programs
dtmp	output from the acctdusg program
fee	output from the chargefee program, ASCII tacct records
pacct	active process accounting file
pacct?	process accounting files switched via turnacct
Spacct?.MMDD	process accounting files for MMDD during execution of runacct
wtmp	active wtmp file for recording connect sessions

Files in the /usr/adm/acct/nite Directory

active	used by runacct to record progress and print warning and error messages. **active**MMDD same as **active** after runacct detects an error
cms	ASCII total command summary used by prdaily
ctacct.MMDD	connect accounting records in tacct.h format
ctmp	output of acctcon1 program, connect session records in ctmp.h format
daycms	ASCII daily command summary used by prdaily
daytacct	total accounting records for one day in tacct.h format
disktacct	disk accounting records in tacct.h format, created by dodisk procedure
fd2log	diagnostic output during execution of runacct (see cron entry)

lastdate last day runacct executed in **date +%m%d** format

lock lock1 used to control serial use of runacct

lineuse tty line usage report used by prdaily

log diagnostic output from acctcon1

logMMDD same as **log** after runacct detects an error

reboots contains beginning and ending dates from **wtmp**, and a listing of reboots

statefile used to record current state during execution of runacct

tmpwtmp wtmp file corrected by wtmpfix

wtmperror place for wtmpfix error messages

wtmperrorMMDD same as **wtmperror** after runacct detects an error

wtmp.MMDD previous day's wtmp file

Files in the /usr/adm/acct/sum Directory

cms total command summary file for current fiscal in internal summary format

cmsprev command summary file without latest update

daycms command summary file for yesterday in internal summary format

loginlog created by lastlogin

pacct.MMDD concatenated version of all pacct files for MMDD, removed after reboot by remove procedure

rprt.MMDD saved output of prdaily program

tacct cumulative total accounting file for current fiscal

tacctprev same as **tacct** without latest update

tacct.MMDD total accounting file for MMDD

wtmp.MMDD saved copy of wtmp file for MMDD, removed after reboot by remove procedure

Files in the /usr/adm/acct/fiscal Directory

cms? total command summary file for fiscal ? in internal summary format

fiscrpt? report similar to prdaily for fiscal ?

tacct? total accounting file for fiscal ?

APPENDIX B
ATTACHMENT 2

Format of wtmp files (utmp.h)

```
/*
/* Format of /etc/utmp and /usr/adm/wtmp
*/

struct utmp {
        char    ut_line[8];     /* tty name /*
        char    ut_name[8];     /* user id /*
        long    ut_time;        /* time on /*
};
```

Definitions (acctdef.h)

```
/*
 * defines, typedefs, etc. used by acct programs
*/

/*
 * following taken from (or modified versions of) <sys/types.h>
*/
typedef unsigned short  dev_t;
typedef unsigned int    ino_t;
typedef long            off_t;
typedef long            time_t;

/*
 *    acct only typedefs
*/
typedef unsigned short  uid_t;

#define LSZ     8       /* sizeof line name */
#define NSZ     8       /* sizeof login name */
#define P       0       /* prime time */
#define NP      1       /* nonprime time */

/*
 * limits which may have to be increased if systems get larger
/*
#define SSIZE 1000 /* max number of sessions in 1 acct run */
#define TSIZE 100 /* max number of line names in 1 ac/t run */
#define USIZE 500 /* max number of distinct login names
                     in 1 acct run                          */
```

```
#define EQN(s1, s2)     (strncmp(s1, s2, sizeof(s1)) == Ø)
#define CPYN(s1, s2)    strncpy(s1, s2, sizeof(s1))

#define SECS(tics)      ((double) tics)/6Ø.
#define MINS(secs)      ((double) secs)/6Ø.
#define MINT(tics)      ((double) tics)/36ØØ.
#define KCORE(clicks)   ((double) clicks/16)
#define SECSINDAY       86400L
```

Format of pacct files (acct.h)

```
/*
 * Accounting structures
/*

typedef unsigned short comp_t;  /* "floating point" */

struct  acct
{
char    ac_flag;        /* Accounting flag */
char    ac_stat;        /* Exit status */
short   ac_uid;         /* Accounting user ID */
short   ac_gid;         /* Accounting group ID */
dev_t   ac_tty;         /* control typewriter */
time_t  ac_btime;       /* Beginning time */
comp_t  ac_utime;       /* Accounting user time */
comp_t  ac_stime;       /* Accounting system time */
comp_t  ac_etime;       /* Accounting elapsed time */
comp_t  ac_mem;         /* memory usage */
comp_t  ac_io;
comp_t  ac_rw;
char    ac_comm[8];     /* command name */
};

extern  struct  acct  acctbuf;
extern  struct  inode *acctp;   /* inode of accounting file */

#define AFORK   Ø1      /* has executed fork, but no exec */
#define ASU     Ø2      /* used super-user privileges */
#define ACCTFØ3ØØ       /* record type: ØØ = acct */
```

Format of tacct files (tacct.h)

```
/*
 *      total accounting (for acct period), also for day
*/
```

```
struct tacct    {
    uid_t  ta_uid;          /* userid */
    char   ta_name[8];      /* login name */
    float  ta_cpu[2];       /* cum. cpu time, p/np (mins) */
    float  ta_kcore[2];     /* cum kcore-minutes, p/np */
    float  ta_con[2];       /* cum. connect time, p/np, mins */
    float  ta_du;           /* cum. disk usage */
    long   ta_pc;           /* count of processes */
    unsigned short ta_sc;   /* count of login sessions */
    unsigned short ta_dc;   /* count of disk samples */
    unsigned short ta_fee;  /* fee for special services */
};
```

Format of ctmp file (ctmp.h)

```
/*
 *      connect time record (various intermediate files)
/*
struct ctmp {
       dev_t   ct_tty;          /* major minor */
       uid_t   ct_uid;          /* userid */
       char    ct_name[8];      /* login name */
       long    ct_con[2];       /* connect time (p/np) secs */
       time_t   ct_start;       /* session start time */
};
```

APPENDIX C
ATTACHEMENT 3

```
Jun  8 Ø4:14 1979  DAILY REPORT FOR pwba Page 1
from Thu Jun  7 Ø6:ØØ:48 1979
to   Fri Jun  8 Ø4:ØØ:28 1979
2        shutdown
2        pwba

TOTAL DURATION IS 132Ø MINUTES
```

LINE	MINUTES	PERCENT	# SESS	# ON	# OFF
tty04	479	36	9	9	3Ø
tty47	341	26	4	4	33
tty44	298	23	3	3	29
tty46	336	25	9	9	33
console	11ØØ	83	14	14	21
tty05	448	34	3	3	22
tty06	439	33	9	9	31
tty07	421	32	6	6	24
tty42	53	4	5	5	2Ø
tty09	385	29	11	11	33
tty1Ø	336	25	1Ø	1Ø	31
tty08	464	35	2	2	19
tty26	544	41	6	6	24
tty12	252	19	5	5	25
tty13	258	2Ø	3	3	21
tty14	156	12	6	6	26
tty17	145	11	1	1	16
tty18	39	3	5	5	24
tty15	228	17	5	5	25
tty25	7Ø4	53	6	6	25
tty21	Ø	Ø	Ø	Ø	16
tty19	1Ø	1	1	1	17
tty2Ø	25	2	2	2	18
tty22	Ø	Ø	Ø	Ø	15
tty23	Ø	Ø	Ø	Ø	15
tty24	Ø	Ø	Ø	Ø	16
tty27	481	36	3	3	2Ø
tty28	426	32	5	5	24
tty29	3Ø2	23	6	6	25
tty3Ø	257	2Ø	11	11	28
tty4Ø	38Ø	29	5	5	21
tty41	343	26	3	3	21
tty45	Ø	Ø	Ø	Ø	15
tty11	365	28	7	7	25
tty43	3	Ø	1	1	17

tty16	213	16	3	3	20
tty31	250	19	4	4	18
tty02	62	5	1	1	3
TOTALS	10544	--	174	174	846

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 1

UID	LOGIN NAME	CPU (MINS) PRIME	CPU (MINS) NPRIME	KCORE-MINS PRIME	KCORE-MINS NPRIME	CONNECT (MINS) PRIME	CONNECT (MINS) NPRIME	DISK BLOCKS	# OF PROCS	# OF SESS	# DISK SAMPLES	FEE
0	TOTAL	388	103	12414	2934	9251	1056	0	16164	174	0	0
0	root	47	41	1003	924	67	30	0	2360	8	0	0
4	adm	7	19	48	652	0	0	0	842	0	0	0
19	games	0	0	4	0	0	0	0	28	0	0	0
22	mhb	0	0	1	1	1	1	0	14	2	0	0
37	abs	0	0	4	0	0	0	0	3	0	0	0
37	absjrk	14	0	284	0	423	0	0	1588	4	0	0
68	rje	3	3	24	21	0	0	0	179	0	0	0
71	?	0	0	0	0	0	0	0	12	0	0	0
150	jac	7	0	156	5	281	2	0	510	13	0	0
173	?	0	0	0	0	0	0	0	16	0	0	0
180	?	0	0	0	0	0	0	0	4	0	0	0
185	?	0	0	0	0	0	0	0	2	0	0	0
217	denise	0	0	2	0	31	0	0	32	3	0	0
217	kof	0	0	2	0	1	0	0	7	1	0	0
219	?	0	0	0	0	0	0	0	12	0	0	0
1001	hsm	5	0	189	0	179	0	0	92	2	0	0
2001	systst	0	1	5	28	476	64	0	99	5	0	0
2002	mfp	1	0	7	5	270	62	0	93	3	0	0
2003	als	1	0	23	0	100	0	0	99	3	0	0
2005	eric	0	0	3	0	13	0	0	21	1	0	0
2006	hoot	0	0	2	0	16	0	0	8	1	0	0
2009	agp	47	0	2040	0	444	0	0	492	2	0	0
2009	fsrepl	2	0	60	0	36	0	0	95	1	0	0
2011	pdw	0	0	1	0	4	0	0	11	1	0	0
2012	pwbst	0	0	1	0	28	0	0	9	1	0	0
2014	cath	0	0	1	0	1	0	0	7	1	0	0
2022	rem	32	1	1227	91	576	4	0	226	3	0	0
2025	fld	55	23	2176	862	336	98	0	750	7	0	0
2027	krb	14	2	365	51	547	24	0	372	8	0	0
2028	text	0	0	1	0	3	0	0	13	1	0	0
2030	arf	8	0	288	0	317	0	0	315	3	0	0
2031	dp	12	0	480	3	459	6	0	220	6	0	0
2032	graf	2	0	49	0	23	0	0	118	1	0	0
2033	ecp	3	0	74	0	355	0	0	115	4	0	0
2040	leap	15	0	308	0	513	1	0	505	2	0	0
2041	dan	3	0	93	3	149	2	0	117	8	0	0
2051	ds52	2	2	19	40	375	601	0	611	8	0	0
2055	nuucp	0	0	15	9	17	1	0	10	3	0	0
2057	ech	1	0	28	0	63	0	0	68	2	0	0
2061	jcw	4	3	99	70	37	34	0	869	4	0	0
2064	mjr	18	0	443	0	176	0	0	2065	3	0	0
2065	rrr	0	0	6	0	7	0	0	23	1	0	0
2068	trc	0	0	7	0	10	0	0	29	1	0	0

2075	herb	29	0	1178	1	384	2	0	249	5	0	0
2086	paul	1	0	14	0	152	0	0	28	1	0	0
2087	pris	0	0	0	10	0	2	0	13	1	0	0
2111	pwbcs	2	3	60	85	64	86	0	185	4	0	0
2116	rbj	1	0	16	0	408	0	0	222	1	0	0
2121	teach	0	0	3	0	53	0	0	50	2	0	0
2123	msb	0	0	3	0	5	0	0	24	1	0	0
2124	rmt	2	0	42	0	66	0	0	260	3	0	0
2126	dal	0	0	5	0	121	0	0	17	1	0	0
2127	m2	15	0	495	11	390	2	0	602	10	0	0

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 2

2128	jel	14	0	492	9	422	14	0	523	8	0	0
2130	s1	0	0	5	1	16	0	0	42	2	0	0
2130	s3	0	0	0	0	0	2	0	9	1	0	0
2135	jfn	0	1	0	12	0	11	0	33	2	0	0
2136	m2class	0	0	5	0	2	0	0	18	1	0	0
2140	star	4	0	213	12	90	3	0	170	7	0	0
2141	reg	5	0	245	25	470	4	0	181	1	0	0
2199	llc	0	0	1	0	10	0	0	7	1	0	0
2999	stock	0	0	1	0	1	0	0	17	1	0	0
3001	whm	5	0	93	0	253	0	0	414	3	0	0
3332	vjf	0	0	4	0	8	0	0	39	1	0	0

Jun 8 04:07 1979 DAILY COMMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCOREMIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	16164	15332.89	490.72	37463.98	31.25	0.03	0.01	322183844	1097670
nroff	119	3958.68	93.21	569.83	42.47	0.78	0.16	67070052	130284
troff	26	2483.38	51.63	342.70	48.10	1.99	0.15	37869304	48989
xnroff	20	732.03	16.74	111.05	43.73	0.84	0.15	13885248	22659
a.out	31	623.53	10.52	142.77	59.26	0.34	0.07	382435	2758
egrep	185	574.83	13.96	34.53	41.18	0.08	0.40	170625	8249
m2find	232	555.79	9.93	155.11	55.96	0.04	0.06	6155937	30994
c1	150	519.04	13.57	48.89	38.25	0.09	0.28	4285724	16032
c0	165	413.10	9.19	35.16	44.93	0.06	0.26	3827309	12170
m2edit	33	340.92	4.63	148.27	73.62	0.14	0.03	1074914	14492
ld	87	317.38	7.94	38.48	39.97	0.09	0.21	17640896	45797
acctcms	17	294.75	6.49	14.15	45.41	0.38	0.46	2525427	5515
c2	112	289.69	9.13	34.61	31.72	0.08	0.26	3667050	9681
sh	1834	276.98	26.77	20444.24	10.35	0.01	0.00	3496613	71979
ed	524	253.13	14.46	2029.89	17.50	0.03	0.01	18058108	56039
acctprc1	3	231.28	6.67	19.45	34.67	2.22	0.34	2577344	2926
du	145	219.35	19.91	39.08	11.02	0.14	0.51	716389	23695
diff	49	175.53	6.04	25.78	29.05	0.12	0.23	3740887	11351
get	151	152.96	4.28	25.23	35.74	0.03	0.17	3634042	24917
adb	22	148.10	4.07	202.35	36.37	0.19	0.02	2313718	9813
tbl	24	143.43	2.44	210.65	58.71	0.10	0.01	1536210	3433
dd	9	139.24	10.15	51.05	13.72	1.13	0.20	26006848	294
as2	155	129.33	9.82	42.25	13.17	0.06	0.23	10500835	30165
sed	597	115.46	4.19	36.23	27.57	0.01	0.12	783825	24497
ps	51	109.69	5.92	41.55	18.54	0.12	0.14	2278056	8310
make	89	102.94	2.87	203.32	35.81	0.03	0.01	1018461	8664
delta	25	90.23	2.27	17.80	39.70	0.09	0.13	2909269	9321
cpp	172	89.37	2.69	11.32	33.19	0.02	0.24	3519054	12155
fsck	16	86.94	1.30	10.57	66.85	0.08	0.12	27671849	2927
find	52	86.64	5.05	63.87	17.15	0.10	0.08	565125	11161
ls	706	82.47	5.78	62.85	14.26	0.01	0.09	1811882	29659
xck	2	79.44	10.49	47.89	7.57	5.25	0.22	198016	21995
awk	22	78.83	1.37	5.24	57.72	0.06	0.26	355466	3769
uucico	60	75.55	1.42	632.50	53.27	0.02	0.00	398693	6377
acctcom	9	75.21	2.81	11.49	26.75	0.31	0.24	1283776	3771
echo	2814	66.10	7.08	91.80	9.33	0.00	0.08	168651	24253
ged	3	57.27	0.82	7.51	70.16	0.27	0.11	51832	426
dc	284	56.92	2.42	9.43	23.48	0.01	0.26	15283	20329
450	7	48.03	6.80	84.45	7.06	0.97	0.08	279451	1700
cat	749	45.49	5.69	478.54	8.00	0.01	0.01	8959500	27903
ntd	6	41.52	1.55	7.55	26.87	0.26	0.20	59888	478
mail	202	39.95	2.05	532.98	19.53	0.01	0.00	427217	14377

acctprc2	3	38.95	1.43	19.45	27.24	0.48	0.07	587336	87
sort	94	38.72	1.09	9.73	35.41	0.01	0.11	375876	4433
pr	104	34.89	2.47	214.50	14.10	0.02	0.01	1060989	6572
haspmain	7	33.20	5.28	1244.54	6.29	0.75	0.00	63064	36635
ex	17	31.69	0.62	41.04	50.97	0.04	0.02	514624	3593
grep	213	28.73	2.98	21.01	9.64	0.01	0.14	2100229	14297

Jun 8 04:07 1979 MONTHLY TOTAL COMMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCOREMIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	553286	297698.78	10916.09	742924.94	27.27	0.02	0.01	820472546	26253312
nroff	1687	44681.55	995.92	5737.25	44.86	0.59	0.17	613403153	1089180
troff	1351	25692.15	583.69	4356.05	44.02	0.43	0.13	413163589	646243
spellpro	6466	17298.41	294.16	1893.79	58.81	0.05	0.16	334572640	853901
m2edit	654	13526.69	164.62	4238.58	82.17	0.25	0.04	54940426	427924
xnroff	397	10408.44	203.72	1496.32	51.09	0.51	0.14	215221419	301967
sort	7983	9292.34	226.01	2298.05	41.11	0.03	0.10	80108304	355963
c1	6139	8949.86	236.45	861.09	37.85	0.04	0.27	79897995	489661
ld	3244	8852.96	223.19	1128.09	39.67	0.07	0.20	493701995	1278119
sed	53134	8126.71	313.85	2241.78	25.89	0.01	0.14	23035033	1692990
m2find	2982	7984.45	140.18	1698.25	56.96	0.05	0.08	111330040	449604
c0	6586	7866.42	185.16	725.47	42.49	0.03	0.26	72595655	389426
ed	20083	7822.78	425.90	41898.18	18.37	0.02	0.01	483425634	1541326
tbl	660	7766.69	113.95	2458.55	68.16	0.17	0.05	50760094	83887
sh	40476	7499.67	635.00	383786.53	11.81	0.02	0.00	70525236	1421194
du	1941	6730.54	553.04	1128.44	12.17	0.28	0.49	20848359	628324
a.out	1483	5658.46	126.87	1868.87	44.60	0.09	0.07	16158675	80260
egrep	4801	5573.51	139.86	460.25	39.85	0.03	0.30	6823696	237298
lint1	793	5325.66	71.23	425.67	74.76	0.09	0.17	9599001	131592
cat	21170	4657.53	236.59	4354.24	19.69	0.01	0.05	239180412	1023965
acctprc1	42	3837.84	110.88	291.34	34.61	2.64	0.38	43954136	61123
c2	4067	3807.25	144.86	477.28	26.28	0.04	0.30	57519376	213521
grep	21212	3204.86	300.44	2727.87	10.67	0.01	0.11	139340583	899415
cpp	7469	3060.72	94.12	647.79	32.52	0.01	0.15	91471956	459882
getty	35556	2948.71	853.53	101107.45	3.45	0.02	0.01	34704751	263866
m2editD	83	2707.27	28.79	361.84	94.02	0.35	0.08	2852202	33949
as2	6454	2698.74	218.96	910.59	12.33	0.03	0.24	213336016	705690
make	1858	2449.10	64.69	4388.86	37.86	0.03	0.01	24116259	175544
ps	1034	2384.14	128.29	1207.87	18.58	0.12	0.11	54873792	204172
acctcms	294	2288.36	51.99	116.06	44.01	0.18	0.45	36124940	80523
uucico	815	2226.75	40.42	11729.01	55.08	0.05	0.00	11086105	162558
ls	18876	2170.01	152.76	1538.09	14.20	0.01	0.10	32418106	691028
find	1705	2114.18	114.35	920.75	18.49	0.07	0.12	94631199	338600
ged	72	2026.43	28.54	317.21	71.01	0.40	0.09	1648636	10374
echo	84710	2018.23	190.14	1138.49	10.61	0.00	0.17	2926992	649200
cpio	127	1956.60	77.03	391.45	25.40	0.61	0.20	190822346	296302
maze	8	1620.42	44.80	128.25	36.17	5.60	0.35	120399	212
mail	4735	1474.38	76.92	14262.62	19.17	0.02	0.01	25719618	463748
get	1085	1358.03	37.59	234.97	36.13	0.03	0.16	31540008	178623
acctcom	165	1253.99	47.06	339.34	26.64	0.29	0.14	57405662	68949
yacc	58	1187.17	15.36	36.90	77.31	0.26	0.42	4096070	12093
col	638	1064.40	49.01	2199.00	21.72	0.08	0.02	23835395	16903

line	27184	1036.03	93.14	1941.33	11.12	0.00	0.05	925447	296142
nroff1.2	29	909.83	17.71	56.97	51.38	0.61	0.31	11459920	18802
delta	264	904.54	23.07	254.06	39.21	0.09	0.09	24219141	87164
td	175	886.19	25.74	159.73	34.43	0.15	0.16	1990177	15792
ar	1434	872.65	61.87	309.07	14.11	0.04	0.20	189858731	428871
m2findD	144	864.29	12.54	344.13	68.94	0.09	0.04	1184947	28576
rm	15319	857.97	85.65	754.20	10.02	0.01	0.11	453479	433903
acctdusg	1	819.77	39.30	170.10	20.86	39.30	0.23	1812480	39744
f77pass1	155	779.13	7.97	29.09	97.70	0.05	0.27	990027	34702
diff	786	767.31	32.77	260.27	23.41	0.04	0.13	22940094	97214

Jun 8 04:07 1979 LAST LOGIN Page 1

00-00-00	dii	00-00-00	rudd	79-06-08	adm
00-00-00	absadm	00-00-00	s10	79-06-08	agp
00-00-00	absafr	00-00-00	s2	79-06-08	als
00-00-00	abscas	00-00-00	s4	79-06-08	arf
00-00-00	absjcw	00-00-00	s5	79-06-08	cath
00-00-00	abspvg	00-00-00	s6	79-06-08	dal
00-00-00	abstbm	00-00-00	s8	79-06-08	dan
00-00-00	adm94	00-00-00	s9	79-06-08	denise
00-00-00	apb	00-00-00	scbsa	79-06-08	dp
00-00-00	archive	00-00-00	sjm	79-06-08	ds52
00-00-00	asc	00-00-00	srb	79-06-08	ech
00-00-00	badt	00-00-00	sys	79-06-08	ecp
00-00-00	btb	00-00-00	tgp	79-06-08	eric
00-00-00	bvl	00-00-00	tld	79-06-08	fld
00-00-00	bwk	00-00-00	ussc	79-06-08	fsrepl
00-00-00	chicken	00-00-00	uucpa	79-06-08	games
00-00-00	class	00-00-00	uvac	79-06-08	graf
00-00-00	cleary	00-00-00	vav	79-06-08	herb
00-00-00	cs	00-00-00	wdr	79-06-08	hoot
00-00-00	dbs	00-00-00	willa	79-06-08	hsm
00-00-00	deby	00-00-00	zooma	79-06-08	jac
00-00-00	dec	79-06-04	dws	79-06-08	jcw
00-00-00	demo	79-06-04	ewb	79-06-08	jel
00-00-00	dlt	79-06-04	kas	79-06-08	jfn
00-00-00	dmr	79-06-04	satz	79-06-08	kof
00-00-00	docs	79-06-04	uucp	79-06-08	krb
00-00-00	dug	79-06-05	bcm	79-06-08	leap
00-00-00	ellie	79-06-05	lprem	79-06-08	llc
00-00-00	fsrep2	79-06-05	s7	79-06-08	m2
00-00-00	gas	79-06-05	sccs	79-06-08	m2class
00-00-00	graphics	79-06-06	conv	79-06-08	mfp
00-00-00	hjg	79-06-06	dck	79-06-08	mhb
00-00-00	hlb	79-06-06	dmt	79-06-08	mjr
00-00-00	inst	79-06-06	emp	79-06-08	msb
00-00-00	jfm	79-06-06	pah	79-06-08	nuucp
00-00-00	jrh	79-06-06	sync	79-06-08	paul
00-00-00	ken	79-06-06	tad	79-06-08	pdw
00-00-00	lco	79-06-07	ams	79-06-08	pris
00-00-00	learn	79-06-07	bin	79-06-08	pwbcs
00-00-00	lppdw	79-06-07	dgd	79-06-08	pwbst
00-00-00	lrbb	79-06-07	haigh	79-06-08	rbj
00-00-00	maj	79-06-07	hasp	79-06-08	reg
00-00-00	mar	79-06-07	jgw	79-06-08	rem
00-00-00	mash	79-06-07	leb	79-06-08	rje
00-00-00	meq	79-06-07	ljk	79-06-08	rnt
00-00-00	mifi	79-06-07	mep	79-06-08	root
00-00-00	mlc	79-06-07	nhg	79-06-08	rrr
00-00-00	mmr	79-06-07	nws	79-06-08	sl

00-00-00	mpf	79-06-07	qtrof	79-06-08	s3
00-00-00	plan	79-06-07	tbm	79-06-08	star
00-00-00	plum	79-06-07	train	79-06-08	stock
00-00-00	pvg	79-06-07	whr	79-06-08	systst
00-00-00	rakesh	79-06-07	wwe	79-06-08	teach
00-00-00	rfg	79-06-08	?	79-06-08	text
00-00-00	rlc	79-06-08	abs	79-06-08	trc
00-00-00	rrc	79-06-08	absjr	79-06-08	vjf
79-06-08	whm				

Awk - A Pattern Scanning and Processing Language *

This information is based on an article
originally written by Alfred V. Aho,
Brian W. Kernighan, and Peter J. Weinberger,
Bell Laboratories.

Table of Contents

SECTION 1
INTRODUCTION

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of awk is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the ZEUS program grep (see ZEUS Reference Manual, Section 1) will recognize the approach, although in awk the patterns may be more general than in grep, and the actions allowed are more involved than merely printing the matching line. For example, the awk program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev      { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk  program  [files]
```

executes the awk commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

```
awk  -f pfile  [files]
```

1.2. Program Structure

An awk program is a sequence of statements of the form:

```
pattern { action }
pattern { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default awk processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields". Fields are normally separated by white space - blanks or tabs - but the input field separator may be changed, as described below. Fields are referred to as **$1**, **$2**, and so forth, where **$1** is the first field, and **$Ø** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument **-fc** may also be used to set **FS** to the character c.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the awk command **print**. The awk program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $Ø }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"file.1"; print $2 >"file.2" }
```

writes the first field, **$1**, on the file **file.1** and the second field on file **file.2** The **>>** notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo.** (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process for instance,

```
print | "mail bwk"
```

mails the output to **bwk**.

The variables OFS and ORS may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

Awk also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints **$1** as a floating point number 8 digits wide, with two after the decimal point, and **$2** as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C.

SECTION 2
PATTERNS

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN   { FS = ":" }
```

Or the input lines may be counted by

```
END  { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete awk program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the ZEUS text editor ed (see ZEUS Reference Manual, Section 1) and grep (without back-referencing). In addition, awk allows parentheses for grouping, || for

alternatives, + for "one or more", and ? for "zero or one", all as in lex. Character classes may be abbreviated: **[a-zA-Z0-9]** is the set of all letters and digits. As an example, the awk program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names

Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in ed and sed. Within a regular expression, blanks and the regular expression meta-characters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

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

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and **!~**. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John". Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign **$** refers to the end.

2.3. Relational Expressions

An awk pattern can be a relational expression involving the usual relational operators <, <=, ><==, **!=**, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u** etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators **||** (or), **&&** (and), and **!** (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selcts lines where the first field begins with "s", but is not "smith". **&&** and **||** guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2      { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

SECTION 3
ACTIONS

An awk action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base e logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string s2 occurs in s1, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2.**

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to **x.** Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

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

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **-**, *****, **/**, and **%** (mod). The C increment **++** and decrement **--** operators are also available, and so are the assignment operators **+=**, **-=**, ***=**, **/=**, and **%=**, These operators may all be used in expressions.

3.3. Field Variables

Fields in awk share essentially all of the properties of variables; they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

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

or assign a string to a field:

```
{ if ($3 > 1000)
        $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

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

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string **s** into **array[1], ..., array[n]** . The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by "is". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $Ø
```

assigns the current input record to the **NR-th** element of the array **x.** In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the awk program

```
        { x[NR] = $Ø }
END     { ... program ... }
```

The first action merely records each input line in the array **x.**

Array elements may be named by non-numeric values, which gives awk a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple,** etc. Then the program

```
/apple/         { x["apple"]++ }
/orange/        { x["orange"]++ }
END             { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else, while, for,** and statement grouping with braces, as in C. We showed the **if** statement in Section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
        print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
        statement
```

does statement with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if** or **while**, or **for** can include relational operators like **<**, **<=**, **>**, **>=**, **==** ("is equal to"), and **!=** ("not equal to"); regular expression matches with the match operators **~** and **!~**; the logical operators **||**, **&&**, and **!**; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes awk to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in awk programs: they begin with the character **#** and end with the end of the line, as in

```
print x, y      # this is a comment
```

SECTION 4
DESIGN

The ZEUS system already provides several programs that operate by passing input through a selection mechanism. Grep, the first and simplest, merely prints all lines which match a single specified pattern. Egrep provides more general patterns, i.e., regular expressions in full generality; fgrep searches for a set of keywords with a particularly fast algorithm. Sed provides most of the editing facilities of the editor ed, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of lex, however, requires a knowledge of C programming, and a lex program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, awk provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing awk went into deciding what awk should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. The syntax is powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, awk usage seems to fall into two broad categories. One is what might be called ``report generation'' - processing an input to extract counts, sums, subtotals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

SECTION 5
IMPLEMENTATION

The actual implementation of awk uses the language development tools available on the ZEUS operating system. The grammar is specified with yacc; the lexical analysis is done by lex; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An awk program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

As might be expected, awk is not as fast as the specialized tools wc, sed, or the programs in the grep family, but is faster than the more general tool lex. The tasks are about as easy to express as awk programs as programs in these other languages; tasks involving fields are considerably easier to express as awk programs.

ZEUS COMMUNICATIONS PACKAGE

Preface

This document describes the ZEUS Communications Package, a communication path between ZEUS and Zilog development tools.

In this document, the term "development system" refers to a standard Z8(TM) or Z8000(TM) Development Module or to Z-SCAN 8000(TM). The term "remote system" refers to a System 8000(TM) executing the ZEUS Operating System. The term "local system" refers to an MCZ(TM) or a ZDS system executing the RIO Operating System.

The LOAD/SEND function in ZEUS is analogous to the MCZ/ZDS LOAD/SEND function. Refer to the Z8000 Development Module Hardware Reference Manual (03-3080) for specific information.

Table of Contents

SECTION 1
INTRODUCTION

The ZEUS Communications Package gives the ZEUS user a communication path between ZEUS and the development tools offered by Zilog (the Z8 and Z8000 Development Modules and Z-SCAN 8000).

The upload/download capability includes the LOAD command, which loads a ZEUS file to development tool memory, and the SEND command, which transfers the contents of development tool memory to a ZEUS file. These facilities also interface with existing PROM programming products, giving the user PROM programming capability.

The package also provides a general-purpose file transfer capability for transferring files between a local system and a remote system. This includes software that executes under both ZEUS and the RIO Operating System.

NOTE

This software package is not designed for communication between two ZEUS systems. For this capability, use the programs uucp, uux, and uulog.

SECTION 2
FUNCTIONAL DESCRIPTION

2.1. Upload/Download Functional Description

The LOAD command downloads a Z8000 program to a development system from a ZEUS file. The binary data in the file is converted to Tektronix format and is transmitted to the development system. An acknowledgment from the development system causes the next record to be downloaded from ZEUS. If an acknowledgment is not received, the current record is retransmitted up to ten times. After continued nonacknowledgment, a record with an error message is sent, and the program aborts.

Possible error messages are:

/ABORT
/UNABLE TO OPEN FILE
/FILENAME ERROR
/INCORRECT FILE TYPE
/ERROR IN READING FILE
/CHECKSUM ERROR

The SEND command transfers the contents of development system memory to a ZEUS file. The SEND program opens the file and sends an acknowledgment to the development system to start transmission. If the file cannot be opened, an abort-acknowledgment is sent, and the program aborts. An acknowledgment is sent after each good record received. If the ASCII code double slash (//) is received from the development system, the program aborts.

Possible error messages are:

/ABORT
/OPEN FILE ERROR
/FILE WRITE ERROR
/CHECKSUM ERROR

2.2. File Transfer Functional Description

The file transfer software copies files residing on the remote system to files residing on the local system, and vice versa. On invocation of the file transfer command (Section 3), the remote system transmits a sequence of

characters to the local system to initiate the file transfer. A file is transferred one record at a time, along with a checksum to guarantee the accuracy of the data. For each successful transmission, an acknowledgment is sent, and a period (.) is displayed on the terminal to inform the user that the transfer is proceeding. If a nonacknowledgment is sent, the record is retransmitted up to ten times, after which the program proceeds to the next file. An error message is displayed for each retransmission that is necessary, unless the nonfatal error messages are suppressed in the command invocation (Section 3). A message is printed after each successful transmission that includes the file name. At the conclusion of the program, a message informs the user of the number of successful and unsuccessful transmissions. A control-x causes the current file transfer to terminate, and the program proceeds to the next file on the list. The termination message counts that file as an unsuccessful transfer (Section 4.2). Pressing the escape key (ESC) aborts the program.

Possible messages are:

Normal transmission:
<filename>
. (one . for every record for positive feedback)

Error messages:
checksum error ... retry
<filename> ... transmission aborted

ZEUS file names cannot be longer than 14 characters, but RIO file names can be as long as 32 characters. For file transfers from the local system to the remote system, only the first 14 characters of the file name are used. Path names can be specified; they apply only to the file name on the remote system. On the local system, all files to be uploaded must be in the working directory, and all downloaded files are created in the working directory (this does not apply to the MCZ/ZDS systems).

NOTE

If a duplicate file name exists on the target system, the contents of pre-existing files are automatically overwritten unless the [-q] option is specified as part of the command (Section 3). If the [-q] option is specified, the user is queried for a replacement name.

Possible message is:

replace <filename> (y/n)?

SECTION 3
INVOCATION AND OPERATION

3.1. Upload/Download Invocation and Operation

The LOAD command is given to the development system as follows:

LOAD file

The development system Monitor program transmits the command line to ZEUS exactly as it is entered, and the ZEUS program (LOAD) opens the file specified by <filename>. The Monitor on a Z8000 Development Module or Z-SCAN requires that <filename> be all uppercase on the remote system. If "load prog" is entered, the remote system searches for the file PROG. The binary data in the file is transmitted to the development system. Pressing ESC aborts the LOAD command.

The SEND command is given to the development system as follows:

SEND file start.addr end.addr [entry.addr]

This command transfers the contents of development system memory to a ZEUS file specified by <filename>. The development system transmits the command to ZEUS exactly as input, causing execution of the SEND program. SEND opens the file <filename> and stores in it the binary data received from the development system. Pressing ESC aborts the SEND command.

3.2. File Transfer Invocation and Operation

File transfer is accomplished in three steps. In the first step, control is transferred from the local system to the remote system by entering the following command to the local system.

remote [rate]

This command starts a program on the local system, which places the user in remote mode. In this mode, all characters entered from the keyboard are sent to the S8000, and all characters from the S8000 (except for character sequences that initiate file transfers and the return to local mode) are sent to the terminal screen. Therefore, the

terminal is essentially operating as an S8000 terminal, and any ZEUS command can be executed. The default communication rate is 9600 baud. Standard baud rates that can be specified for the MCZ/ZDS are 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19,200, and 38,400.

The second step in file transfer involves two commands: putfile and getfile, which are invoked as follows:

putfile [-qfbB] file.1 [[-b]file.2]
getfile [-qfbB] file.1 [[-b]file.2]

The command putfile transfers files from the remote system to the local system; getfile transfers files from the local system to the remote system.

The [-q] option specifies that transfer of a file to the target system where a file of the same name already exists causes a query to the user (Section 2.2). If this option is not given, the file is automatically overwritten.

The [-f] option suppresses the nonfatal error message "checksum error ... retry."

The [-b] option preceding a file name indicates a binary file and suppresses translation of ZEUS new line characters into RIO's carriage returns (and vice versa) for that file only. The type defaults to ASCII for the next file. This differs from the [-q] and [-f] options, which apply to the remainder of the line following the point at which they are invoked.

The [-B] option specifies that every file that follows is binary.

A list of files can be specified on the command line. A control-x aborts the transfer of a single file and proceeds to the next file. Pressing ESC aborts the entire transfer at any point.

The third step returns the user to the local system from the remote system. The command is:

local [-l]

The [-l] option causes a logout to be given to the remote system. It is necessary to log in after the next remote command.

SECTION 4
TERMINATION

4.1. Upload/Download Termination

After completion of the loading process, the program's entry point is displayed on the terminal, and the development system returns to Monitor mode. The LOAD program terminates and returns control to the ZEUS Operating System.

After completion of the sending process, the program's entry point is stored in the ZEUS file, and the development system returns to Monitor mode. The SEND program terminates and returns control to the ZEUS Operating System.

If there is a user or program abort during either the loading or sending process, an error message is printed (Section 2), the development system returns to Monitor mode, and the program returns control to the ZEUS Operating System.

4.2. File Transfer Termination

After completion of the file transfer, the local system returns to remote mode, enabling the user to continue to execute ZEUS commands. One of the following messages is printed on the terminal:

```
putfile:<n1> successful transfers <n2> unsuccessful transfers
getfile:<n1> successful transfers <n2> unsuccessful transfers
```

An unsuccessful file transfer does not cause the program to terminate abnormally. If the program is aborted via the escape key, it does not transfer any more files, and terminates in a normal fashion.

THE C SHELL

This information is a summary and expansion of information found in the ZEUS Reference Manual and other sources.

Table of Contents

List of Tables

List of Illustrations

SECTION 1
AN INTRODUCTION TO THE C SHELL

1.1. What is a Shell?

A shell is an interactive program that interprets and executes commands. It is the software interface between commands typed at the terminal, and the functions of the computer.

The shell also determines the qualities of each operating environment. The options and variables set for the shell become the options and variables established for each command.

Upon login, the operating system (ZEUS) initiates a C Shell process for the user. This interactive process is the user's login shell, it is the parent process for all subsequent processes (known as child processes). The login shell is an environment that defines the basic parameters of interaction between the user and the operating system.

The login shell defines the home directory for the user, it defines the path to any commands that may be used, the prompt given to the user to indicate that the system is ready for another command, the shell to be used, and the type of terminal in use.

If no modifications are made to that environment, its parameters default to a limited set of qualities.

Each user can customize this environment to suit individual taste, need, and/or application. The qualities of the shell environment can be altered by establishing new shell variables, or by changing existing shell variables.

ZEUS supports two shell programs each with its own set of commands and variables. The default shell program for ZEUS is the C Shell (written by William Joy at the University of California, Berkeley).

The second shell program is the Bourne Shell (also known as "the shell" because it was the first). The Bourne Shell was written by S. R. Bourne of Bell Laboratories. (see **sh**(1) and The Bourne Shell in the Zeus Utilities Manual).

The C shell is moderately more powerful than the Bourne Shell because of its enhanced command structure and its use

of conventional C programming syntax. Either shell program will serve as the login shell, and either program can be called interactively from the terminal.

1.2. Conventions in this Document

Bold

Information in **Bold** is typed literally -- commands such as **ls** are typed into the terminal just as they appear on paper. In the following example, the command **date** is typed exactly as it is shown:

date

Underlined

In the examples, underlined words are sample words only -- not the literal word. They can or must be replaced by words of the user's own choosing. In the following example:

date > filename

the filename must be replaced by the name of a file created by the user, or to which the user has access.

Outside of examples, words are underlined to illustrate that they have special meaning in the ZEUS operating system, but are not necessarily commands. For example, the term pipe refers to both the vertical bar " | " character and the process of feeding the output of one command into the input of the next command. However the word pipe is not a command. The names of pre-defined C shell variables are also highlighted in underlining.

Underlining is also used to highlight words in a **bold** expression, as in a section header.

command(1)

A word followed by a single number in parentheses as in **ls**(1) is a command, the word in bold is the command and the number in parenthesis " (1) " refers to the documentation section in the ZEUS Reference Manual.

In this case, refer to Section 1 of the ZEUS Reference Manual. The commands in the Reference Manual are arranged alphabetically within each section.

quotes

Special characters are printed in bold and within quotes as in "**?**" to distinguish them from document text.

Examples:

The examples used in this document are drawn from the Zilog Systems Publications System 8000. In many cases the file names, user names, and file system configuration will be different from any other installation.

Each user may obtain different data from the examples, depending upon the installation.

An example of a command takes the following format:

command **operator** filename

It is indented from the body of the text and separated from the text above and below by a blank line. In some cases a simple example may occur within the body of the text, e.g. to show the **ls**(1) command.

Variable Names

When variables are called, either at the terminal or in a shell script, they are preceded by a dollar sign "**$**" as in the command:

echo $PATH

when variables are explained in the text, they are referred to without the dollar sign prefix. See **echo**(1) and Section 10.

SYNOPSIS:

The synopsis line(s) demonstrates the syntax of a given command illustrating where the options, flags, or keys are placed (if any), and where the filename is placed (if any). The following example demonstrates a "SYNOPSIS:" section:

SYNOPSIS:

date > filename

If more than one line appears, it means that the item being explained is used more than one way.

Square brackets [ ]

Within the SYNOPSIS line, square brackets indicate that the material they enclose is optional, it can appear in the command, but is not mandatory. In the example:

echo [-n] string

the " **-n** " flag is optional. It can appear, but can be omitted. In the acutal command typed into the terminal, only the options are typed, not the braces.

Ellipsis ...

Three dots in a row " ... " indicates that the preceding element can be repeated any number of times. In the following example:

command {item1,item2,...}

the ellipsis indicates that there can be any number of items between the braces.

DEFAULT:

In the DEFAULT section, the value that exists for a variable (unless another is specified) is shown.

ALSO SEE:

If reference is made to Section 3 -- Filename Substitution it refers to Section 3 of this document. The "ALSO SEE:" section also refers to other documents in the ZEUS Utilities Manual, and other manuals in the ZEUS and System 8000 collection.

Capital Letters

Capital letters are used in proper names, and at the beginning of new sentences. If a sentence begins with a command name that command name will be capitalized even though all built in ZEUS commands must be typed in lower case letters.

The names of environment variables such as PATH are named in all capital letters by convention.

Split.Words

Some expressions are really two words, but must be written (typed into the computer) as one word because of the way the computer interprets blank spaces. In such cases, the expression can be presented as two words separated by a dot or an underbar (instead of a space character) as in:

```
command.1
 or
READ_ME
```

this conforms to the convention for naming files that is popular with experienced ZEUS users.

SECTION 2
TYPING COMMANDS TO THE C SHELL

Commands are typed into the computer in response to a "prompt".

2.1. The ZEUS Prompt

A prompt is a signal from the computer that it is ready to accept user input. It prompts the user to type a command.

On the ZEUS system, the default user prompt is a percent sign " % ".

2.2. Command Syntax

Many commands in the ZEUS system consist of a single word followed by a "RETURN" character. These are known as simple commands, and are entered with the following syntax:

command

An example of a simple command is the **date** command, which is entered as:

date

and produces results similar to:

Tue Nov 23 14:14:35 PST 1982

Most commands can be modified to supply more or better information. Modification takes the form of one or more arguments in the form of options, flags, keys, or filenames.

Commands with arguments are typed into the terminal keyboard with the following syntax:

command option flag key and/or filename

Each command determines its own syntactic requirements, i.e. the author of the program writes the syntactic requirements into the body of the program.

Some programs require that a minus sign " - " begin an option, flag, or key. The **ls**(1) program requires that options begin with a minus sign. The " l " option provides a long listing of the files, it is typed as:

ls -l

Other programs make the minus sign optional. The **tar**(1) (tape archiver) program options don't use the initial minus sign. The " t " option provides the table of contents for the files in the tape archive, it is typed as:

tar t

Some programs, like the **tar** program require that an option, flag or key be the second argument to the command; others, like the **ls** program, make the arguments optional.

Multiple words in a command are separated by blanks (spaces or tabs), or semi-colons, with the first word indicating the action and the remaining words serving as arguments as in:

ls -l

where the " **-l** " flag is an argument to the **ls** command instructing the computer to provide a long list.

2.3. Simple Commands

SYNOPSIS:

command

A command is an instruction to the computer. A simple command consists of one or more characters typed into a computer terminal at the "prompt". The command is terminated with a "RETURN" character.

A command consists of at least one word that specifies an action to be taken. For example:

ls

is the command requesting a list of the files and directories in the current working directory. See **ls**(1).

The **ls** command produces results in the following format:

```
csh.Ø1   csh.Ø3   csh.Ø5   csh.Ø7   csh.9A   csh.9T
csh.Ø2   csh.Ø4   csh.Ø6   csh.Ø8   csh.9B   temp
```

each name refers to a file or a directory in the current directory.

2.4. Compound Commands

SYNOPSIS:

command.1; command.2

Sequences of commands can be separated by a semi-colon, and are then executed sequentially, as in:

ls; who ; pwd; date

producing results in the format:

```
csh.Ø1   csh.Ø3   csh.Ø5   csh.Ø7   csh.9A   csh.9T
csh.Ø2   csh.Ø4   csh.Ø6   csh.Ø8   csh.9B   temp
patty    ttyØ     Nov 23 Ø8:Ø4
deck     tty2     Nov 23 Ø9:38
carol    tty8     Nov 23 Ø8:17
craig    tty9     Nov 23 Ø8:36
/z/deck/Util/New.csh
Tue Nov 23 14:14:35 PST 1982
```

See **ls**(1), **who**(1), **pwd**(1), and **date**(1).

2.5. Running a Command in Background

SYNOPSIS:

command &

Because some commands take several minutes to complete the ZEUS system provides a mechanism for running several commands at once by detaching the commands from their dependency upon the terminal -- this is known as running the commands detached or in background.

Control of the terminal is returned to the user while the command continues to execute in another part of the computer. A command runs in background when it is followed with an ampersand (" & "). Error diagnostics, unless otherwise instructed, return to the standard error output device -- the terminal. For example a command to compile a C

program called test.c takes the following form:

```
cc test.c &
```

(See **cc**(1) in the Zeus Reference Manual (ZRM) for more information on the **C** compiler.)

Verification that the compile process is running comes from the **ps**(1) command. The full exchange would take the following form:

```
% cc test.c &
2999
% ps
   PID TTY TIME CMD
  1309 2   0:19 csh
  2999 2   0:00 cc
  3002 2   0:03 ps
```

The **cc** test.c **&** command starts a compile process. A process identification number appears on the screen, followed immediately by the next prompt. The **ps** command is entered as soon as the prompt appears, even if the previous process is still running and the current processes are displayed.

2.6. Running a Command in a Subshell

SYNOPSIS:

```
(command)
```

Commands in parentheses are always executed in a subshell. In the following example, running the command in a subshell prevents **cd** from affecting the current shell.

Thus the command:

```
(cd; pwd)
```

prints the name of the home directory, leaving the current working directory untouched, while the command:

cd ; pwd

changes the current working directory to the home directory and then prints the name of that directory, leaving the cursor in the home directory. This command structure is useful as a temporary escape from the current working directory.

ALSO SEE:

cd(1)

2.7. Conditional Command Operators

SYNOPSIS:

command.1 **&&** command.2
command.1 **||** command.2

An operator is a symbol that changes the way a command works. In mathematical commands (like **bc**(1), and **dc**(1) the on-line calculators), the standard math operators are "**+**", "**-**", "*****" and "**/**" for "add, subtract, multiply and divide" respectively.

The following two operators are "logical" operators, the logical "and" operator, (**&&**) and the logical "or" operator (**||**). These operators separate two commands on a single line, and determine whether one or the other, both or neither command is executed.

The determination is based on whether or not the first command executes successfully. If the first command executes without an error it is said to have executed successfully, and returns a status code of zero "Ø". If it executes unsuccessfully it returns a non-zero exit status, usually a "1".

Unfortunately, the "Ø" means "false" and the "1" means "true" to the C shell, and to these conditional operators, thus, the syntax of the operators seems somewhat reversed when used on commands.

The following table demonstrates the results of these operators.

Table 2-1 Conditional Command Operator Summary

First command executes	operator	Second command executes
yes	|| "or"	= yes
no	|| "or"	= no
no	&& "and"	= yes
yes	&& "and"	= no

In the command:

ls || date

the C shell executes both **ls** and **date**

In the command:

bogus.command || fake.command

the C shell tries to execute bogus.command and failing, does not attempt to execute fake.command.

In the command:

bogus.command && **ls**

the C Shell tries to execute bogus.command and failing, executes **ls**.

Finally, in the command:

ls && bogus.command

the C shell executes the **ls** command, and succeeding, does not attempt to execute bogus.command. ALSO SEE:

Section 7 -- Shell Variables -- the Status Variable
Section 8.4.1 -- C Shell Scripts -- And and Or Operators

2.8. Command Substitution

SYNOPSIS:

`command`

A command inside back quotes will be executed and the output of the command replaces the command itself.

For example, the command:

echo "Today is `date`"

produces output similar to:

Today is Fri Dec 1Ø 17:16:ØØ PST 1982

2.9. Input/Output Control

The ZEUS system has three channels of communication between the user and the computer, one standard input channel and two channels of output, the standard output and the error output.

By default (unless otherwise specified) input comes into the computer via the terminal keyboard. This is the standard input. Output goes from the computer to the terminal screen and is the standard output. Any errors resulting from the execution of a program produces an error message to the terminal screen. This is the standard error.

There are circumstances in which input must come into a program from some other sources (e.g. from a file). Likewise, there may be a need to redirect the output and the error messages.

Although the standard input, output and error channels default to the keyboard and the terminal screen, they can be changed using greater than (">") and less than ("<") the following sections demonstrate the redirection syntax.

2.9.1. Input Redirection -- < :

SYNOPSIS:

command < com.list

The file com.list is opened and its contents are used as input for the command. As in:

wc < text.file

which uses text.file as the input for the word count command

wc(1). This produces results in the following format:

```
474   2055   12623
```

The first number is the number of lines, the second number is the number of words, and the last number is the number of characters in the file.

Another example is to create a file called com.file with **ex**(1) editor commands, e.g. a command to remove all the leading blank spaces from all the lines. The command:

```
ex test < com.file
```

invokes the **ex** editor on the file test, but instead of taking editor commands from the standard input (the keyboard), the commands are read from the file com.file.

ALSO SEE:

wc(1) **ex**(1) and The Ex Reference Manual in the ZEUS Utilities Manual.

2.9.2. Input Within a Script -- <<:

SYNOPSIS:

```
command << label
```

A shell script is a file of commands that are executed one at a time by the shell, just as if they were typed into the the terminal at the prompt. In most cases, commands within a shell script draw input either from the terminal or from other files, but in some cases it may be necessary to draw input from the shell script itself. (See Section 8 -- The Csh Command and C Shell Scripts)

The double less than symbol permits a shell script to take data from within its own text (rather than relying on data from external files).

This is most useful in the context of editor scripts; consider the following shell script:

```
# deblank -- remove blank lines
ex test << 'EOF'
g/^$/d
w
q
'EOF'
```

In the example, the line:

ex text << 'EOF'

means that the file test is edited with the **ex** editor, and that the command input for **ex** comes from the body of the shell script, rather than from some external source (like another file). The " **<< 'EOF'** " notation means that the input is taken "up to **'EOF'**". Single quotes around " **'EOF'** " prevents any variables from from being expanded to the contents they contain. (See Section 7 -- Shell Variables)

ALSO SEE:

ex(1) and The EX Reference Manual in the Zeus Utilities Manual.

2.9.3. Output Redirection -- > :

SYNOPSIS:

command > testl

The file testl is used as output. If the file does not exist, it is created; if the file exists, it is overwritten, and its previous contents are erased. The command:

ls -l > test2

puts the output of the **ls -l** command into a file named test2.

NOTE

A file is always erased (if it exists) before the new information is written into it.

The command

cat file1 > file2

erases any information in file2 before putting the contents of file1 into it.

NOTE

To prevent the accidental erasure of a file, the **noclobber** variable can be set with the command:

set noclobber

ALSO SEE:

Section 2.9.6 "Overriding noclobber" for examples, and Section 7.1 for more details on Predefined C Shell Variables (noclobber) and **cat**(1).

2.9.4. Adding to the End of A File -- >> :

SYNOPSIS:

command >> file

The double greater than sign construction (" >> ") adds the output of command to the end of file instead of erasing file first.

If file does not exist it is created automatically.

For example, if file1 consists of 3 lines:

```
Now is the time
for all good people
to come to the aid of their party
```

and file2 consists of 1 line:

```
The quick brown fox jumps over the lazy dog
```

Then the command:

cat file1 >> file2

produces a new file2 that contains 4 lines:

```
The quick brown fox jumps over the lazy dog
Now is the time
for all good people
to come to the aid of their party
```

Note that the contents of file1 have been appended to the

end of file2.

2.9.5. Standard Error Redirection -- >& :

SYNOPSIS:

command **>&** file

The greater than sign followed by an ampersand (" **>&** ") routes error messages into the specified file along with the standard output.

Given the command:

cat bogus.file **>** new.file

if bogus.file does not exist, an error returns:

cat: cannot open bogus.file

using the greater than sign, ampersand construction, the command:

cat bogus.file **>&** new.file

redirects any error messages to the file new.file, and can be examined as any other text file.

2.9.6. Overriding Noclobber -- >!:

SYNOPSIS:

command **>!** file

If file exists and the C shell variable noclobber is set, a command using the simple form of output redirection (" > ") fails and an error message results. In the command:

cat file1 **>** file2

if file2 exists, the command produces the error message:

file2**: File exists.**

The **noclobber** variable inhibits accidental destruction of files. In this case, the greater than sign, combined with the exclamation point (" **>!** ") form of redirection can be

used to suppress this check.

The command:

    cat file1 >! file2

succeeds, file2 is overwritten by file1 even if file2 exists and the **noclobber** is set. The command produces no error message.

2.9.7. Output, Error, and Noclobber -- >&! :

SYNOPSIS:

    command >&! file

This form combines the ampersand (" & ") and exclamation point (" ! ") constructions mentioned above, directing error output to the file, and overriding the noclobber variable if it is set.

2.9.8. Appending and Standard Error -- >>& :

SYNOPSIS:

    command >>& file

This form combines the "Adding to the End of a File" (double greater than construction) with the "Redirecting the Standard Error" (ampersand), appending the output of command and any error messages to the end of file.

2.9.9. Appending and Noclobber - >>! :

SYNOPSIS:

    command >>! file

This form combines the "Adding to the End of a File" (double greater than construction) with the "Overriding the Noclobber variable" (exclamation point), appending the output of command to the end of file without regard to the **noclobber** variable (if it is set).

2.9.10. Appending, Noclobber, Error -- >>&! :

SYNOPSIS:

command >>&! file

Appends the output at the end of file. If the variable **noclobber** is set, it is ignored, and the standard error is also appended.

2.10. Pipes

SYNOPSIS:

command | command

A sequence of simple commands separated by a vertical bar " | " also known as a pipe forms a pipeline. The output of each command in a pipeline becomes the input of the next. This example:

who | grep chuck

takes the output of the **who**(1) command and pipes it through the command **grep**(1) to extract the line with the word chuck if that line exists.

This command is equivalant to redirecting the output of the **who** command into a temporary file and then running the command **grep** chuck on that temporary file and removing the temporary file as in the sequence:

```
who > temp
grep chuck temp
rm temp
```

The command:

who

produces a list in the following format:

```
karen    tty0    Nov 23 08:04
chuck    tty2    Nov 23 09:38
mike     tty6    Nov 23 14:50
carol    tty8    Nov 23 08:17
george   tty9    Nov 23 08:36
```

The output is redirected to a file with the command:

who > temp

The line with the word chuck is extracted with the command:

grep chuck temp

to produce the output:

chuck tty2 Nov 23 Ø9:38

and the temporary file is removed with the command:

rm temp

All of this can be accomplished with the pipe mechanism as in the command:

who | grep chuck

which produces the desired output:

chuck tty2 Nov 23 Ø9:38

The following tables show summaries of the command structure and I/O redirection characters.

Table 2-2 Command Structure Summary

command	Simple command
command flag	Command with an option argument
command filename	Command with a filename argument
command; command	Compound command
command &	Running a command in background
(command)	Running a command in a subshell
`command`	Command substitution

Table 2-3 Input and Output Redirection Summary

Symbol	Meaning
<	Take input from
<<	Take input up to
>	Redirect output
>&	Redirect output and error
>!	Redirect output -- override noclobber if set
>&!	Redirect output and error; override noclobber
>>	Append output
>>&	Append output and error
>>!	Append output and override no clobber
>>&!	Append output and error; override noclobber
|	Output from first command is input for second

SECTION 3
FILENAME SUBSTITUTION
AN INTRODUCTION TO METACHARACTERS

3.1. Characters for Filenames

The C shell provides a method of shorthand communication. In the case of filenames, the shell provides a number of special characters (known as metacharacters, magic characters, or wild-card characters) that will expand to the names of files and directories according to specific rules.

The process is also referred to as pattern matching and filename expansion. When a metacharacter is used, names of the files and directories are scanned to see if the pattern set by the metacharacter is matched by any of those file and/or directory names.

The true qualities of a metacharacter are revealed with the **echo** command. The command:

echo metacharacter

will return the pattern the metacharacter stands for.

That pattern matching takes place under the following rules:

Asterisk -- *

SYNOPSIS:

command *

The asterisk is a very powerful character. It is shorthand for "any pattern" in file and directory names. For example, the command:

ls *

lists all files and directories. The command:

ls a*

lists all files and directories that begin with the letter "a". Lastly, the command:

ls /z/deck/U*/*

lists all the files and directories under the directory or directories in /z/deck that begin with the letter "U".

Question Mark -- ?

SYNOPSIS:

command ?

The question mark is shell shorthand for "any single character". Thus the command:

ls ???

lists all the files and directories with three (and only three) character names. For example, the files:

abc dog psl

match the string "**???**" while the following files do not:

file1 jj make.p test.c

The characters can be letters, numbers, or any other legitimate (non-metacharacter) filename character. Similarly, the command:

ls csh.??

will produce a list of the files and directories that start with "csh." and end with any two characters. For example, the files:

csh.Ø1 csh.Ø2 csh.Ø3

match, while the files:

csh.1 csh.test csh.A

do not.

Pattern/range -- [A-Z]

SYNOPSIS:

command [beginning of range-end of range]

Square brackets define a range of characters that match

any single character falling within that range (alphabetical or numeric).

ls csh.Ø[1-9]

lists any file starting with "csh.Ø" and ending with a number 1 through 9. e.g. csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.Ø6 csh.Ø7 csh.Ø8 csh.Ø9. The command:

ls csh.[1-3][1-9]

will list all the files from "csh.11" to "csh.39". Other characters can also be specified in the range -- the range proceeds along the ASCII numbering scheme. With the exception of special characters, the ASCII ordering sequence runs from Ø-9, A-Z, and a-z. Thus the full range of alpha-numeric characters (and some non-alpha-numeric characters) is covered with the expression Ø-z.

ALSO SEE:

ascii(7)

Abbreviation -- {A,B,C}

SYNOPSIS:

command {item.1,item.2,...}

The braces refer to a selection of characters or strings -- any one of which may or may not match a file or directory. The command:

```
ls file.a{b,c,d}e
```

lists the files:

```
file.abe file.ace file.ade
```

if they exist, similarly, the command:

```
ls /usr/man/man1/{csh,ls,dog}.1
```

matches the files:

```
/usr/man/man1/csh.1
/usr/man/man1/ls.1
/usr/man/man1/dog.1
```

The selection of characters or strings need not be in a range, or in any particular order. They need not be the same length. but they must be separated by commas with no spaces between them.

Tilde -- ~

SYNOPSIS:

command ~
command ~user.name

The tilde serves as an abbreviation and refers to the user's home directory reading the name of the directory from the HOME variable.

The command:

ls -l ~

expands to this user's home directory:

ls -l /z/deck

When followed by a name, the shell searches for a user with that name and substitutes their home directory; thus the command

ls -l ~carol

expands to

ls -l /z/carol

If the tilde "~" is followed by a name, other than a name in the password file, or a slant "/", it is taken as a literal tilde by the shell. For example, in the command:

cat ~filename

the shell looks for a file with the exact name "**~filename**".

A summary of the filename expansion characters appears in the table below.

ALSO SEE:

Section 1Ø.1 -- Environment Variables.

Table 3-1 Filename Substitution Character Summary

*	Any string
?	Any single character
[A-Z]	Any character in the range A to Z
{A,B,C}	Any element from the set A, B, or C
~	Home directory or user name.

3.2. The Full Metacharacter Set

Filename expansion is one example of the way the shell uses special characters. Each of the following characters has some special meaning to the shell and/or the ZEUS operating system.

The list below describes these characters in their order of appearance in the ASCII ordering scheme, and their special meaning in the C Shell, the History Function and the operating system.

Space ' '

SYNOPSIS:

command space arguments
command tab arguments

The space character delimits words in commands. When a compound command is typed, the Shell uses blank space -- the space or tab characters -- to distinguish the various components. This process of separating a command into specific words is known as "parsing" the command. The command:

ls -l /tmp /z /usr/spool

is understood by the Shell because the component parts are broken into recognizable bits by the delimiting spaces.

Exclamation point ' ! '

SYNOPSIS:

!character, number or string

The exclamation point is used in the Shell to initiate

a call to the shell's history mechanism (See Section 4). Previously typed commands are numbered from one, saved in a history list and can be re-invoked using this device. The command:

!ls

searches back through the history list to find and execute the most recent command that begins with the string **ls**. The command:

!3

searches back through the history list to find and execute command number "3". The exclamation point is also used in a variety of programs to call the shell.

ALSO SEE:

Section 4 -- The History Function

Double quote ' " '

SYNOPSIS:

command "string"

The double quotes character (") is used on either side of an expression to inhibit the expansion of various other special characters. The command:

echo *

expands the asterisk metacharacter "*" to all file and directory names in the current directory, while the command:

echo "*"

merely echos the asterisk.

ALSO SEE:

Section 3.3 -- Quoting -- Preventing Metacharacter Expansion.

Pound Sign ' # '

SYNOPSIS:

comment

The Pound Sign is used as the first character in a shell script to indicate that the C Shell is to be used to execute the script. The pound sign is also used inside the body of a shell script to begin a comment -- the pound sign tells the C shell to ignore the rest of the line. Inside the body of a shell script, the line:

 # this is a comment line

will be ignored by the shell.

ALSO SEE:

Section 8 -- Shell Scripts

Dollar Sign ' $ '

SYNOPSIS:

command **$variable**
command **!$**

The Dollar Sign has special meaning in a variety of circumstances. When used with a variable name, as in the command

echo $prompt

it refers to the shell variable "prompt". When used with the history mechanism, as in the command:

!$

it refers to the last element of the last command. If the last command is:

ls -l /z/joe/file.1

the command:

cat !$

produces the results as if the command had been:

cat /z/joe/file.1

ALSO SEE:

Section 7 -- Shell Variables
Section 4 -- The History Function

Ampersand ' & '

SYNOPSIS:

command &

!N:s/x/&/

command && command

In the first case, the Ampersand, used at the end of a command runs the command in background -- control of the terminal is returned to the terminal even if the command is not completed. The command:

cc test.c &

begins a compile process and immediately returns a prompt. At that point another command can be issued, even while the compile process is still running.

In the second case the Ampersand also stands for "the string just substituted" in a history substitution.

In the third case, the ampersand is used as the logical "and" operator in conditional commands.

ALSO SEE:

Section 2.5 -- Running a Command in Background
See Section 4 -- The History Function
See Section 2.7 -- Conditional Command Operators

Single quote "'"

SYNOPSIS:

command 'string'

The single quote is another quoting device used by the shell to inhibit or prevent expansion of special characters. The command:

echo $prompt

produces:

%

while using double quotes, as in the command:

echo "$prompt"

also produces:

%

To prevent the string "$prompt" from being expanded by the shell, single quotes must be used. The command:

echo '$prompt'

produces:

$prompt

ALSO SEE:

Section 3.3 -- Quoting -- Preventing Metacharacter Expansion.

Left Parenthesis ' ('

SYNOPSIS:

(command **)**
foreach variable **(** list **)**

In the first case, a command issued within parentheses is always executed in a subshell. It is rather like a temoporary escape into another working environment, for example, if the present working directory is **/tmp** and the following command is issued:

(cd ; pwd)

the C Shell creates a new C Shell and executes the command within that new shell. The subshell dies and control returns to the parent shell from which the command was issued. The present working directory remains **/tmp**. This is significantly different from the command:

cd ; pwd

which causes the present working directory to change to the home directory.

In the second case, parenthesis are used to delimit a word list in shell loops, as in the statement:

foreach i (1 2 3 4)

The parentheses indicate to the shell that the list "1 2 3 4" is to be used as the loop control mechanism. This also applies to the **if**, **while**, and **switch** statements, covered later.

ALSO SEE:

Section 2.6 -- Running a Command in a Subshell
Section 6 -- The C Shell Programming Language Structure

Right Parenthesis ') '

SYNOPSIS:

(command **)**
foreach variable **(** list **)**
while (expression **)**

The right parenthesis ends a loop control mechanism, or a subshell command.

Asterisk ' * '

SYNOPSIS:

command *

The Asterisk is a filename expansion character, it matches any pattern.

ALSO SEE:

Section 3.1 -- Characters for Filenames

Plus Sign ' + '

SYNOPSIS:

number + number
variable++

In the on-line calculators (**dc**(1) and **bc**(1)) and in the math functions of C Shell scripts, the plus sign is used in the addition function. In the body of a shell script, the line:

@ X=(6 + 6)

gives the variable "X" the value of 12. The plus sign

is also used to increment variables, as in the statement:

```
@ i++
```

which increments the value of i by 1 each time the statement is executed.

In the following shell script, the variable i is incremented within a loop:

```
# the name of this file is "test.file"
@ i=1
while ( 1 )
                echo $i
                @ i++
end
```

It is executed with the command:

```
csh test.file
```

and it produces the following output:

```
1
2
3
4
5
6
7
.
.
.
```

until an interrupt (the DELETE key) is hit.

ALSO SEE:

Section 5.3.9 -- The "At Sign" @

Comma ' , '

SYNOPSIS:

```
command {item1,item2}
```

The comma is used to delimit elements within braces. In the command:

```
cat csh.Ø{3,5,7}
```

the commas are necessary dividers between the digits 3, 5, and 7.

ALSO SEE:

bc(1), **dc**(1)
Section 7 -- Shell Variables.

Minus ' - '

SYNOPSIS:

number - number
@ variable--
command -(flag, option, or key)

Like the plus sign, the minus sign is used as the subtraction operator within shell scripts. It is also used as a variable decrementer. The following shell script:

```
# the name of this file is test.file.2
@ i=15
while ( 1 )
                echo $i
                @ i--
end
```

The shell script is executed with the **csh** command as in the example above, and it produces the following output:

```
15
14
13
12
11
1Ø
9
.
.
.
```

until an interrupt (the DELETE key) is encountered.

Of special importance to the shell, the minus sign is used to initialize flags, options, or keys for many Shell commands as in:

```
ls -l
```

ALSO SEE:

Section 5.3.9 -- The "At Sign" @
Section 2 -- Typing Commands to the C Shell

Dot ' . '

SYNOPSIS:

command .
.filename

Though not specifically a function of the C Shell, the period (always referred to as "dot") is used by the operating system to mark the current working directory. The command to copy (**cp**(1)) a file to the current directory takes the following syntax:

cp /tmp/karen .

This command is shorthand for the command

cp /tmp/karen (current working directory)

When used as the first character in a filename, the dot makes the filename transparent to a standard **ls** command ("dot" files will show up with the "-a" option, as in "**ls -a**").

In the second case, there are several "dot files" that are of special importance to the shell. The **.login** file is read by the Shell at login, the **.cshrc** file is read each time a new C Shell is invoked (forked), and the **.exrc** file is read by the **ex** editor to establish basic options, etc.

Filenames that begin with a dot are not listed with the standard **ls** command, but are listed with the **-a** (all) option as in the command **ls -a**.

ALSO SEE:

Section 9 -- C Shell Files

Dot-dot ' .. '

SYNOPSIS:

command ..

Like "dot", "dot-dot" is used by the operating system to mark a location in the file system, specifically, the parent directory to the present working directory.

If the present working directory is " /z/deck ", the command:

cd ..

changes the current working directory to " /z ". Both dot "." and dot-dot ".." appear as directory names with the command " **ls -a** ".

ALSO SEE:

Section 9 -- C Shell files.

Slant ' / '

SYNOPSIS:

command /path

The forward slash character, referred to as "slant" is used as a path delimiter to locate files. In the path,

/usr/spool/mail/user.name

the slants separate directory and filenames.

If the first character in a pathname to a file is a slant, the shell starts from the root of the file system (the directory named "/") to locate the file. For example, if the present working directory is /tmp, the command:

ls -l /z/paula/temp

will locate only one file with that exact path, while the command:

ls -l z/paula/temp

will look for a file named /tmp/z/paula/temp. If the first character in a path name is not a slant, the shell starts from the current working directory to match the file.

Colon ' : '

SYNOPSIS:

!identifier:modifier

In combination with the history mechanism, the colon is used to modify previous commands. The command:

!1:s/who/date/

will repeat command number 1, substituting the word date for the word who.

ALSO SEE:

Section 4 -- The History Function

Semi-colon ' ; '

SYNOPSIS:

command; command

The semi colon is a command separator. The command:

ls;who;pwd;date

can be typed on a single line and parsed into its individual parts by the shell.

Less Than ' < '

SYNOPSIS:

command < file
if (variable < variable **)**

In the first case, the "less than" sign is used to redirect input from file into the command.

In the second case, math operations use this character as the relational operator "less than", as in the expression:

```
if ( $a < $b ) then
...
```

ALSO SEE:

Section 2.9.1 -- Input Redirection and 2.9.3 -- Output Redirection

Equals ' = '

SYNOPSIS:

set variable=value
if (variable == variable) then

Shell variables are established with the **set** command using the syntax shown in the first case.

In the second case, math operations within shell scripts use the double "equals" character means "equal to" as in the expression:

```
if ( $a == $b ) then
...
```

ALSO SEE:

Section 7 -- Shell Variables

Greater Than ' > '

SYNOPSIS:

command > file
if (variable > variable) then

In the first case the "greater than" character redirects the output of command to file.

In the second case, as a math operator within a shell script, this character means "greater than" as in the line:

```
if ( $a > $b ) then
...
```

ALSO SEE:

Section 2.9.1 -- Input Redirection and 2.9.3 -- Output Redirection
Section 7 -- Shell Variables

Question Mark ' ? '

SYNOPSIS:

In the first case, the question mark is used as a filename substitution character, it matches any single character in a filename. In the example above, the command will affect filenames with a single character.

The question mark is also used to delimit strings in the history mechanism. The command:

!?string?

extracts the most recent command with string in it.

ALSO SEE:

Section 3.1 -- Characters for Filenames
Section 4 -- The History Function.

At Sign ' @ '

SYNOPSIS:

@ variable=number

This character sets variables with numeric values rather than string values, such that math operations can be performed on them. The command:

@ x=(6 + 6)

sets the variable "x" to the number 12, while the command

set x=(6 + 6)

sets x to the character string "6+6".

A space must separate the at sign from the rest of the variable assignment.

ALSO SEE:

Section 5.3.9 -- The "At Sign" @

Left Bracket ' ['

SYNOPSIS:

command [range]
$variable[subscript]

command [range]
$variable[subscript]

The left and right brackets are used to delimit a range of characters used for pattern matching in filename expansions.

They are also used to subscript a variable -- to isolate a component of a variable with multiple elements.'

If the variable "X" is set to the set of characters " a b c d e " with the command:

```
set X=(a b c d e)
```

the third element of the set, "c", is addressed with the command:

```
echo $X[3]
```

ALSO SEE:

Section 3.1 -- Characters for Filenames
Section 7.4 -- User-defined Variable Substitutions

Right Bracket '] '

SYNOPSIS:

command [range]
$variable[subscript]

The right bracket is used to close a range value or a subscript value.

ALSO SEE:

Section 3.1 -- Characters for Filenames
Section 7.4 -- User-defined Variable Substitutions

Backslash ' \ '

SYNOPSIS:

command \metacharacter

The backslash character escapes the magic qualities of the special characters in this section. The command:

translation.

*

ALSO SEE:

Section 3.3 -- Quoting -- Preventing Metacharacter Expansion.

Up Arrow ' ^ '

SYNOPSIS:

!identifier:^
^string1^string2^

The up arrow (sometimes referred to as a "caret" or a "hat") is used by the History function and is shorthand for "the first element". The command:

!5:^

refers to argument number 1 in the 5th command.

In the second case, the up arrow is also a substitution device in the history mechanism. Given a command:

ls -l /z/sisan

the next command:

^i^u^

produces the correct command:

ls -l /z/susan

This mechanism is much like the "**:s**" substitution mechanism in the history function.

ALSO SEE:

Section 4 -- The History Function

Back quotes ' ` '

SYNOPSIS:

command `command`

SYNOPSIS:

command `command`

Commands placed inside back quotes are executed and the output of the command replaces the statement in back quotes. For example, the command:

echo `date`

produces

Wed Dec 8 15:Ø1:48 PST 1982

ALSO SEE:

Section 2.8 -- Command Substitution

Left Braces ' { '

SYNOPSIS:

command {string1,string2}
command ${variable}word

The left and right braces delimit abbreviations in filename expansion and insulates variables when used in juxtaposition to other words.

If the variable "X" is set to the number "4" the following command:

echo ${X}9ers

results in:

49ers

as opposed to the command:

echo $X9ers

which results in the error:

X9ers: Undefined variable.

The braces prevent the surrounding text from affecting the "X" variable.

ALSO SEE:

Section 3.1 -- Characters for Filenames
Section 7 -- Shell Variables

Vertical Bar (pipe) ' | '

SYNOPSIS:

command | command
command || command

The single vertical bar acts as a pipeline, connecting the output of the command on the left side to the input of the command on the right side.

In the second case, the double bar mechanism is used as the logical "or" command operator.

ALSO SEE:

Section 2.10 -- Pipes
Section 2.7 -- Conditional Command Operators

Tilde ' ~ '

SYNOPSIS:

command ~
command ~user.name

The tilde is a filename expansion character. It expands to the home directory.

The following table summaries C shell metacharacters.

ALSO SEE:

Section 3.1 -- Characters for Filenames

Table 3-2 Metacharacter Summary

Char:	Meaning:	Context:
space	Delimits words	Commands
!	Accesses history	History
"	Quoting mechanism	Commands
#	Comment line	Shell scripts
$	Last element	History
%	String isolation	History
&	Background command	Prompt
&	Pattern substitution	History
&&	Logical "and" operator	Commands
(	Begins string	Command loops
(	Begin subshell	Commands
)	Ends string	Command loops
)	End subshell	Commands
*	All characters	Filenames
+	Addition	Shell scripts
++	Variable incrementer	Shell scripts
,	Range delimiter	Commands
-	Flag	Commands
-	Subtraction	Shell scripts
--	Variable decrementer	Commands
.	Current Directory	Filenames
.	"dot" files	Filenames
/	Path delimiter	Filenames
:	History modifier	History
;	Command delimiter	Commands
<	Input redirect	Commands
=	Equals	Shell scripts
>	Output Redirect	Commands
?	Any single character	Filenames
@	Math operations	Shell scripts
[	Begins range	Filenames
]	Ends range	Filenames
\	Escapes metacharacters	Commands
^	First argument	History
`	Quoting device	Commands
`	Command Substitution	Command
{	Begins abbreviations	Filenames
}	Ends abbreviations	Filenames
|	Pipe mechanism	Commands
||	Logical "or" operator	Commands
~	Home	Filenames

3.3. Quoting -- Preventing Metacharacter Expansion

There are situations in which metacharacters should not be expanded. For example, a shell script that includes both an editor command with dollar sign "$" and a variable name with a dollar sign.

In these cases, the dollar sign in the editor command must be quoted so that its significance is taken literally by the shell, not expanded.

There are 4 quoting devices available on the ZEUS system as the following table illustrates:

Table 3-3 Quoting Devices

The backslash	\
Double quotes	"
Right quote	'
The noglob option	**set noglob**

The following rules apply to quoting devices:

Table 3-4 Quoting Device Summary

Character:	Symbol:	Quotes Variables:	Quotes Filenames:
The backslash	\	yes	yes
Double quotes	"	no	yes
Right quote	'	yes	yes
Noglob	n/a	no	yes

The following table shows the effect of the available quoting devices.

Table 3-5 The Effect of Quoting Devices

Command:	\	"	'	noglob
echo *	*	*	*	*
echo $HOME	$HOME	/z/deck	$HOME	/z/deck

The following table shows which characters must be escaped when used in commands if their meaning is to be taken literally. Unescaped, they are used as command operators, or expanded to file and directory names:

Table 3-6 List of Metacharacters that Must Be Escaped

ampersand	&
asterisk	*
backslash	\
dollar sign	$
exclamation point	!
greater than sign	>
left brace	{
left bracket	[
left parenthesis	(
single quote mark	'
less than sign	<
question mark	?
quote mark	"
right brace	}
right bracket	]
right parenthesis	)
back quote	`
semi-colon	;
tilde	~
up arrow	^
vertical bar (pipe)	|

SECTION 4
THE HISTORY FUNCTION

4.1. Command History

Commands typed into the terminal are numbered sequentially from 1 and are saved in memory on a history list. The size of the history list is controlled by the history variable. See 7.1.4 -- history.

the command:

set history=15

which can be entered at the prompt or entered in the .cshrc or .login file will keep a list of the last 15 commands.

No commands are stored if the history variable is unset. The history variable is unset by default.

The command:

history

displays the current history list in the following format:

```
5     vi temp
6     ls
7     more temp
8     history
9     cat csh.Ø1
1Ø    cat csh.Ø1 > temp.2
11    more temp.2
12    who
13    vi temp.2
14    echo $prompt
15    who > temp.3
16    cat temp.3
17    ls -la
18    vi temp.3
19    history
```

Note that the list contains the last 15 commands. When the number of commands on the list exceeds 15, the oldest command drops (irretrievably) off the end of the list. In other words, if the history variable is set to 15, command number 16 pushes command number 1 off the list.

Commands from the history list can be recalled and manipulated at the prompt. The exclamation point "**!**" is used to initiate a call to the history function.

4.2. Common Forms of Use for the History Function

Table 4-1 below demonstrates the 7 most common uses of the history function. Since these commands fall into different categories, the explanation of each form is repeated in subsequent sections.

Table 4-1 Common Forms of History Manipulation

Syntax:	Explanation:	Example:
!!	Repeat the last command	!!
!n	Repeat command number n	!6
!string	Repeat command starting with string	!ls
!$	Last argument of previous command	ls !$
!*	All arguments except #Ø	ls !*
^x^y^	Substitute y for x	^wrong^right^
!!:n	Argument number n	ls !15:2

Double exclamation points -- !!

SYNOPSIS:

!!

The double exclamation point means "repeat the last command again". The last command (from the list above) is **history**, the command **!!** produces the following exchange:

```
% !!
history
      6     ls
      7     more temp
      8     history
      9     cat csh.Ø1
      1Ø    cat csh.Ø1 > temp.2
      11    more temp.2
      12    who
      13    vi temp.2
      14    echo $prompt
      15    who > temp.3
      16    cat temp.3
      17    ls -la
      18    vi temp.3
      19    history
      2Ø    history
```

Exclamation point; number -- !n

SYNOPSIS:

!n

An exclamation point and some number means "repeat command number". The command:

!6

repeats command number 6 from the history list. In this case (from the above list) the command **ls**.

Exclamation point; string -- !string

SYNOPSIS:

!string

An exclamation point and a string means "repeat the command that begins with string". The command:

!v

will search out the most recent command beginning with the letter "**v**". In this case, the command is:

vi temp.3

Exclamation point; Dollar sign -- !$

SYNOPSIS:

command **!$**

In addition to full command lines, portions of those lines, the individual words (arguments to the command) can be manipulated.

The exclamation point, dollar sign combination refer to the last argument of the previous command, thus if a command:

ls -l /z/hank/temp

the next command:

cat !$

produces the command:

cat /z/hank/temp

Exclamation point; Asterisk -- !*

SYNOPSIS:

command **!***

The exclamation point, asterisk combination means "the second argument to the last argument", thus given the command:

ls /z/joe/work /z/zubes/art

the command:

cat !*

produces the command:

cat /z/joe/work /z/zubes/art

Double up-arrows -- ^string1^string2^

SYNOPSIS:

^string1^string2^

The up-arrow works as a string substitution device for previous commands. Given the erroneous command:

cat /z/curl/letter

the command:

^curl^carol^

produces the command:

cat /z/carol/letter

The trailing up-arrow can be omitted if the last character of the string is a RETURN.

Double exclamation point; number -- !!:n

SYNOPSIS:

[command] !identifier:n

The call to the history function (an exclamation point followed by the identifier associated with the desired command), followed by a colon and a number is a call to the numbered argument of that command. For example, given the command:

ls /z | wc

which lists all the files and directories in the directory /z and pipes the results through the word count (**wc**(1)) command. The subsequent command:

cd !!:1

produces the command:

cd /z

4.3. Accessing Previous Commands

Saved commands (events) from the history list can be called up and executed again with a variety of commands, all of which begin with the exclamation point.

Commands, or events are recorded in order by their event number. Each event can be tracked by making the event number a part of the prompt. This is done by placing an

exclamation point (which must be escaped) "\!" in the prompt string.

Adding the line:

```
set prompt="%\! "
```

to the ~/.cshrc or ~/.login file produces the following prompt for the first command:

```
%1
```

The number will increment by one for each command.

!! repeats the immediately previous command. In the case below, the last command (command number 5) is **history**.

!n is short for "repeat command number n"

For example, if the history list is:

```
1  ls
2  who
3  pwd
4  date
5  history
```

The pwd(1) command can be executed again by typing the command:

```
!3
```

!-n means execute the command n commands back from the current command.

pwd (the third command back from the current command) can also be executed with the command:

```
!-3
```

!string

means "execute the most recent command with the prefix string."

The history function will search back through the history list and look for the most recent occurrence of a command beginning with the letter (or string of

letters) in string. The command

!p

will also produce the **pwd** command from the history list example shown above.

!?string**?**

will match a command with string in an event argument; trailing "?" can be omitted if nothing follows

!?wd?

will also produce the **pwd** from the sample history list.

Table 4-2 Accessing Previous Commands

Syntax	Explanation	Example
!!	Repeat the last command	!!
!n	Repeat command number n	!3
!-n	Repeat the command n commands back	!-4
!string	Repeat the command starting with the string string	!p
!?string?	Repeat the command containing the string string	!?wd?

4.4. Modifying Previous Commands

Portions of previous commands can be isolated and manipulated at the prompt. It is possible to access both an individual command, and individual arguments to that command.

Arguments within each event are numbered sequentially from zero and can be selected from an event by the sequence "!n:" followed by one of the argument designators listed in Table 4-3.

For example, given command number 1 as:

%1 ls; who; pwd; date

The arguments are numbered as follows:

Argument:	Ø	1	2	3	4	5	6
Word	**ls**	**;**	**who**	**;**	**pwd**	**;**	**date**

ls is the zero-th argument,
semi-colon is the first argument,
who is the second argument,
semi-colon is the third argument,
pwd is the fourth argument,
semi-colon is the fifth argument,
date is the sixth argument.

The following examples demonstrate the use of the history mechanism to access arguments from within that command.

!1:n refers to the nth argument from command number 1. The command:

```
!1:1
```

produces and tries to execute the first command argument (in this case the semi-colon):

```
;
```

!1:^ (the up arrow)
refers to the first argument in command number 1. The command:

```
!1:^
```

Produces and tries to execute the first semi-colon of the command string:

```
;
```

!1:$ refers to the last argument of command number 1. The command:

```
!1:$
```

produces and executes

```
date
```

!1:n-m
is a range of arguments from number n to number m. The command:

!1:2-4

produces and executes the second, third, and fourth command arguments; **who** the semi-colon, and **pwd:**

who; pwd

!1:-n
abbreviates the range of arguments from number Ø to number n. The command:

!1:-3

produces and executes the zero-th, first, second and third arguments from the first command:

ls; who;

!1:* abbreviates the range of arguments from number 1 to the last argument or nothing if there is only one argument in the event. The command:

!1:*

produces and executes the first, second, third, fourth, fifth and sixth arguments of command number 1:

; who; pwd; date

!1:n*
abbreviates the range of arguments from number n to the last argument. The command:

!1:2*

produces and executes the second, third, fourth, fifth, and sixth arguments.

who; pwd; date

!1:n-
like **!1:n*** but omitting the last argument (argument "$"). It abbreviates the range of arguments from number

n to the next-to-last argument. The command:

```
!1:2-
```

produces and executes the second, third, fourth, and fifth argument from command number 1:

```
who; pwd;
```

Table 4-3 Accessing Previous Command Words

Syntax	Explanation	Example
!N:n	Command N, argument n	!1:2
!N:^	Command N, first argument	!1:^
!N:$	Command N, last argument	!1:$
!N:n-m	Command N, argument n through m	!1:3-5
!N:-n	Command N, argument Ø through n	!1:-3
!N:*	Command N, argument 1 to the last	!1:*
!N:n*	Command N, argument n to the last	!1:3*
!N:n-	Command N, argument n to last-1	!1:3-

4.5. Modifying Previous Command Words

In addition to calling and modifying commands from the history list, and calling and modifying arguments within the individual commands, portions of arguments can also be called out and manipulated separately.

Given the command:

```
%1 ls -l /z/deck/Util/Cshell/csh.Ø1
```

the following modifiers are defined:

:h Remove a trailing path name component, leaving the head. The command:

```
!1:h
```

produces:

```
ls -l /z/deck/Util/Cshell
```

:r Remove a trailing .xxx component, leaving the root name. The command:

!l:r

produces:

ls -l /z/deck/Util/Cshell/csh

:t Remove all leading path name components, leaving the tail. The command:

!l:t

produces:

ls -l csh.Ø1

:p Print the new command but do not execute it. The command:

!l:p

produces

ls -l /z/deck/Util/Cshell/csh.Ø1

but does not execute it.

:s/string1/string2

Substitute string2 for string1; trailing "/" can be omitted if new line follows; "/" is not a unique delimiter

The command:

!l**:s/**Cshell/Shell/

produces and executes:

ls -l /z/deck/Util/Shell/csh.Ø1

:q Quote the substituted arguments, preventing further substitutions.

Given command number 1 as:

ls -l /z/deck/Util/Shell/csh.Ø1

and given the substitution:

!1:s/csh.Ø1/csh*/

The following command:

!!:q

produces:

ls -l /z/deck/Util/Shell/csh*

however, in this case the asterisk is taken literally, it is not expanded. Consider the following exchange (spaces have been added to the output to improve readability):

```
% 1 ls -l csh.Ø1

-rw-r--r-- 1 deck  system  15459 Oct 13 12:49 csh.Ø1

% 2 !1:s/csh.Ø1/csh.*/

ls -l csh.*

-rw-r--r-- 1 deck  system  15459 Oct 13 12:49 csh.Ø1
-rw-r--r-- 1 deck  system  18238 Oct 13 12:5Ø csh.Ø2
-rw-r--r-- 1 deck  system  13347 Oct 13 12:5Ø csh.Ø3
-rw-r--r-- 1 deck  system   3316 Oct 13 12:5Ø csh.Ø4
-rw-r--r-- 1 deck  system  3Ø814 Oct 13 12:51 csh.9A
-rw-r--r-- 1 deck  system   2395 Oct 13 12:51 csh.9T
-rw-r--r-- 1 deck  system  45153 Nov  9 17:59 csh.ref

% 3 !2:q

ls -l csh.*

csh.* not found
```

:x Like q, but break into arguments at blanks, tabs, and new lines.

:& Repeat the previous substitution.

!1:&

Table 4-4 Modifying Previous Command Words

Syntax:	Explanation:	Example:
!n:h	take the head of the pathname	**!l:h**
!n:r	leave the root of the filename	**!l:r**
!n:t	leave the tail of the pathname	**!l:t**
!n:p	print but don't execute	**!l:p**
!n:s/X/Y/	replace X with Y	**!l:s/unix/zeus/**
!n:q	quote substituted arguments	**!l:q**
!n:x	quote, break substituted arguments	**!l:x**
!n:&	repeat previous substitution	**!l:&**

Unless preceded by a "**:g**", the modification is applied only to the first modifiable argument.

A backslash character "\" must be used to escape a slash "/" character if it is used in the left side of the substitution string -- i.e. if it is part of stringl in the example below:

!l:s/stringl/string2/

4.6. Magic Characters in History Function

& The ampersand character "&" in the right side of the substitution statement is replaced by the text from the left side of the statement. For example, if the first command is

%l **ls** /z/deck/Util

a substitution using the ampersand can be used as follows:

!l:s/Util/&.plus/

resulting in the command:

ls /z/deck/Util.plus

null Nothing (a null) in the left string uses the previous string from a previous substitution command i.e.

!l:s//Memos/

substitutes the previous string "Util" for the string "Memos" producing the command:

ls /z/deck/Memos

The string used initially (in this case "Util") must be present in the command string called with the null or the error

Modifier failed

will result.

!$** A history reference can be given without an event specification; for example, "**!$" refers to the last argument in a command string.

Given the command:

ls -l /z/deck/Util/Cshell/csh.Ø1

The command:

cat !$

produces:

cat /z/deck/Util/Cshell/csh.Ø1

In this case, the reference is to the previous command.

Thus,

!?string**?^ !$**

gives the first and last arguments from the command matching ?string?.

^ Simple command substitutions are made at the prompt with the up arrow key (usually a shift or "upper-case" 6 on the keyboard). The erroneous command:

cat /usr/lab/news/zeus

Can be fixed with the command:

^lab^lib^

{} A history substitution can be surrounded with left and right brackets "{" and "}" to insulate it from the characters which follow. Thus, after

ls /z/cheryl

enter

!{l}/temp

to do

ls /z/cheryl/temp

as opposed to the command:

!l/temp

which looks for a command starting

l/temp

% produces the argument matched by the immediately preceding ?string? . Given the command:

cd /z/deck/Util/Shell

The command:

ls !?Shell?%

will produce the the command

ls /z/deck/Util/Shell

The following table shows a summary of metacharacters used in the history substitution function.

Table 4-5 Metacharacters in History Substitutions

Character:	Meaning:
\	Escapes magic qualities
&	The string just substituted
//	(null) the string just searched for
!$	The last element of the last command
?x?	String search for "x"
^x^y^	Substitution routine
{x}	Search insulators
%	Element search

SECTION 5
THE C SHELL BUILT-IN COMMAND STRUCTURE

The C Shell makes an excellent operating environment because it provides a number of improvements over previous shells. In particular, the C shell provides a built-in command language using the structure of the C Programming Language.

This section covers general purpose commands that are used either at the prompt (i.e. from the command line) or from within the body of a shell script. The next section covers the C Shell programming language.

5.1. Introduction to C Shell Commands

There are 38 built-in C Shell commands. These commands fall into four major categories.

The first ten are general purpose commands. The second ten deal with establishing or altering the working environment. Together these 20 commands form a set of commands that is both useful from within the body of a shell script, and when typed directly to the terminal.

The next section presents commands which deal specifically with programming. The 15 commands in this set are used to control the flow of operations within a shell script.

The last set of three commands are general purpose commands that are useful almost exclusively from within the body of a shell script.

For more detailed information on the C Programming Language, as a means of understanding the basic structure and syntax of the C shell programming language, refer to The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie. (1978, Prentice Hall Inc., Englewood Cliffs, New Jersey 07632).

5.2. General Purpose Commands from the Prompt

Built-in commands are a part of the C shell itself, not separate programs executed by the C shell. The following built-in commands are useful both in the body of a C Shell script and when typed at the prompt.

5.2.1. cd:

SYNOPSIS:

cd
cd name

Without an argument, **cd** changes the current working directory to the user's home directory. **Cd** reads the HOME variable to determine the user's home directory.

With a path name as an argument, **cd** changes the current working directory to the directory name. Error messages are produced if the target name is not a valid directory name, or if the user is not permitted access to that directory (see **chmod**(1)).

Verification of the current working directory is derived from the **pwd**(1) command.

ALSO SEE:

cd(1), **pwd**(1), **chmod**(1) and
Section 7.1.5 -- home

5.2.2. echo:

SYNOPSIS:

echo [-n] string

The words in string are printed (echoed) on the terminal. This is useful in generating output from a shell script. **Echo** can also be used to verify the true properties of the shell's variables and metacharacters. For example, the command:

echo *

expands the metacharacter asterisk (" * ") to all the file names in the current working directory (see Section 3--Filename Substitution) and prints the list in the following format:

csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T

The **-n** option inhibits a newline at the end of the output string. From within the body of a shell script, the

commands:

```
echo -n Hello
echo \ Roberta
```

produces the output:

```
Hello Roberta
```

Note that the "hard space" (the "\ " sequence) is needed to put a space between the strings. Normally **echo** ignores leading and trailing blank spaces (space or tab characters).

ALSO SEE:

echo(1) and **echo2**(1)
And the **echo** variable in Section 7.1 -- Predefined C Shell Variables.

5.2.3. glob:

SYNOPSIS:

glob string

The **glob** command is much like the **echo** command except that words are not separated by spaces, and no newline ends the string. This command is useful for programs which use the shell to expand a list of words.

The command:

```
glob *
```

produces the results:

```
csh.Ø1csh.Ø2csh.Ø3csh.Ø4csh.Ø5csh.9Acsh.9T
```

5.2.4. history:

SYNOPSIS:

history

The **history** command displays a record of past commands. The length of the history list is determined by the history variable (See Section 7.1 -- Predefined C Shell Variables).

The history variable can be set to 15 (for example) with the command:

set history=15

With the history variable set to 15 the command:

history

produces a history list of the last 15 commands. Results are produced in the following format:

```
1    alias
2    alias ll 'ls -l'
3    foreach i ( 1 2 3 4 )
     .
     .
     .
15   history
```

When command number 16 is reached the list will begin with command number 2, etc. There are no diagnostic messages. (See Section 4 -- The History Function).

5.2.5. nice:

SYNOPSIS:

nice
nice -number
nice command
nice -number command

The nice value sets the priority of a command in the system's Central Processing Unit (CPU). The nice value of each job is seen with the command:

ps -l

which produces results that look like:

```
...  CPU PRI   NI  ADDR  SZ  WCHAN TTY TIME COMD
...    0  30   20    f5  15   dd18 2   0:16 csh
...    0  30   20   5d3   9   debc 2   0:01 sh
...    0  30   20   4b1   9   ded8 2   0:02 csh
...    0  28   20   61a   8   e886 2   0:00 script
...    0  30   20    bb  14   df48 2   1:30 vi
...    9  30   20   77e   9   df64 2   0:02 csh
...    0  26   20   692   8   327a 2   0:00 script
...  101  56   20   706  11        2   0:02 ps
```

NOTE: The first 5 columns have been omitted to make the output fit the space. Refer to **ps**(1) for details.

The nice value (shown in column 3 in the example) can be increased, making the job run at a lower system priority, with the **nice** command.

Without an argument, **nice** increments the nice value for the current shell by 7. With an argument, in the form:

nice -N

The nice value is increased by N, which may be any number up to 20.

With a command as a second argument in the form:

nice -N command

The nice value of command is increased by N.

There is no way for a normal user (anyone but the superuser, **ZEUS**) to "un-nice" a process; only the superuser can set a negative nice number (increase the priority of a command). **ZEUS** can issue that command in the form:

nice --N

The first argument to **nice** must be either a minus sign - or a command.

ALSO SEE:

nice(1)

5.2.6. rehash:

SYNOPSIS:

rehash

The C Shell maintains a sorted list (a hash table) of all the commands available to the user. This list is created upon login, and when a new C Shell is invoked (forked). The hash table is stored in memory and contains a list of all the filenames (commands) in the directories named in the user's search path.

When a new command is created (as in the case of a new shell script), it does not appear in the hash table even though it is in a directory in the search path. This is true unless the new command (file) is in the user's current working directory.

A new command is installed into the hash table, when it is located in a directory named in the user's PATH, and the hash table is rehashed with the **rehash** command.

The message:

command.name: **command not found.**

will appear if the command is not installed into the hash table or is not in the current working directory.

ALSO SEE:

Section 7.1.11 -- path

5.2.7. repeat:

SYNOPSIS:

repeat N command

The specified command is repeated N times.

The command:

repeat 5 ls

produces results in the format:

```
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.9A csh.9T
```

Error diagnostics are repeated.

5.2.8. time:

SYNOPSIS:

time
time command

With no argument, a summary of time used by the current shell and its child processes is printed. The command

time

produces results that look like:

```
Ø.1u Ø.3s Ø:1Ø 3%
```

The first column reports the user seconds, the second column reports the system seconds, the third column is real time, and the last column reports the percentage of total system capacity used by the command.

If arguments are given, the specified simple command is timed and produces a time summary as described above.

The command:

```
time ls
```

produces results that look like:

```
csh.Ø1      csh.Ø4      csh.Ø7      csh.9B      junk
csh.Ø2      csh.Ø5      csh.Ø8      csh.9C      make.out
csh.Ø3      csh.Ø6      csh.9A      csh.9T      temp
Ø.1u Ø.2s Ø:Ø1 23%
```

The **time** command produces results even if the command being timed produces an error.

ALSO SEE:

time(1)

5.2.9. umask:

SYNOPSIS:

umask
umask N

The **umask** value determines the default file protection mode for new files. (See **chmod**(1))

Without an argument, the **umask** command displays the current umask value.

With a number argument, the umask value is set to N. The command:

umask Ø26

will result in new files with the following protection mode (as displayed with the **ls -l** command):

```
-rw-r----- 1 deck  system  Ø Dec  1 14:15 temp
```

Umask codes for new files are as follows:

```
ØØØ =          -rw-rw-rw-
111 =          -rw-rw-rw-
222 =          -r--r--r--
333 =          -r--r--r--
444 =          --w--w--w-
555 =          --w--w--w-
666 =          ----------
777 =          ----------
```

Note that the execution bit (the "x" bit) is never set. This is a protection against accidental execution of text files which can result in unintended (and potentially destructive) consequences.

Umask codes for directories will set the execution bit if desired. The specific codes are as follows:

```
ØØØ =          drwxrwxrwx
111 =          drw-rw-rw-
222 =          dr-xr-xr-x
333 =          dr--r--r--
444 =          d-wx-wx-wx
555 =          d-w--w--w-
666 =          d--x--x--x
777 =          d---------
```

If umask is unset it defaults to 002.

ALSO SEE:

chmod(1)

5.2.10. wait:

SYNOPSIS:

wait

The **wait** command causes the terminal to freeze until all background (child) processes terminate.

An interrupt will disrupt the wait. If a wait is interrupted the process identification number and the command name are displayed. Consider the following exchange:

```
sleep 200 &
27233
wait
                        (interrupt)
 27233  sleep
wait: Interrupted.
```

In the above example a command (**sleep 200**) is run in background (**&**). A process identification number returns (**27233**), then the **wait** command is issued.

The **wait** is interrupted (the interrupt character DELETE or RUB does not show), the process identification number is displayed along with the command name (**27233 sleep**) and the diagnostic message **wait: Interrupted.** on the next line.

ALSO SEE:

wait(1)

The following table shows the commands in Group 1.

Table 5-1 Built-in Command Summary -- Group 1

GROUP 1

GENERAL PURPOSE COMMANDS USEFUL FROM THE PROMPT:	
cd	Change working directory
echo	Print a string on the terminal
glob	Like echo -- but no spaces separate words
history	Print command history list
nice	Set the running priority of a command
rehash	Re-sort the search path for commands
repeat	Repeat a command
time	Time the execution of a command
umask	Set the execution bits on new files
wait	Wait for background jobs to finish

5.3. Environmental Commands from the Prompt

This set of commands is used to customize the working environment. These commands are used to create and remove shorthand commands (aliases) for long commands, set and unset C Shell variables, and shorthand words for file or directory names.

5.3.1. alias / unalias:

SYNOPSIS:

alias name long command
alias name
alias
unalias name

The **alias** command establishes user-defined shorthand for long commands. With the syntax **alias** name long command the command creates an alias name for the command long command. The command:

alias h history

creates the alias **h** which is used as the command **history.** The command:

h

now produces the same output as the command:

history

(**history** remains a valid command). In this context, the **alias** command produces no output. Verification that **h** is the alias for **history** can be derived from the next commands.

With one argument, **alias** displays the alias for that argument if one exists. With the **h** alias established, the command:

alias h

produces the output:

history

Without an argument, **alias** displays a list of the current aliases. With the alias established above, the command:

alias

produces a listing in the following format:

h history

unalias is used to remove an alias (see **unalias** below)

Looping can occur in an alias that calls itself -- as in the command:

alias ls 'pwd; ls'

Each call to **ls** attempts to execute **pwd** and then **ls** which calls the alias again. This is true unless the first word of the alias is the command itself, for example, the alias:

alias ls'ls; pwd'

works without a loop error.

The problem of looping is prevented by the C shell which produces the error message:

Alias loop

upon a call to a looped alias (although the alias can be established). The alias must be removed with the command:

unalias alias name

EXAMPLE 1:

To establish an alias called "**ls**" for the longer command "**ls -l**" the following alias command is used:

alias ls 'ls -l'

The command

ls /z/carol

will produce output as if the command typed is:

ls -l /z/carol

EXAMPLE 2:

Aliases accept input as well. For example, a file name can be passed to an aliased command.

The expression " **\!*** " like its counterpart in the history mechanism expression means "argument number 1 to the last word" (Refer to Section 4 -- The History Function). This expression substituted all the arguments typed at the command line (except for argument number Ø -- the command itself) into the aliased command. Thus, the command:

alias print 'pr \!* | lpr'

creates an alias called print that calls **pr**(1), accepts one or more arguments (file names) as input, and pipes the output of that command through **lpr**(1). The alias is used with the syntax:

print file.1

The results are the same as if the following command had been entered.

pr file.1 | lpr

the **\!*** expression is replaced by arguments 1 through the last (argument zero in this case is the word **print**).

5.3.2. exit:

SYNOPSIS:

exit
exit (N)

Exit is comparable to **logout**(1). It is a means of terminating the current working environment (the shell) by killing the process associated with that shell. **Exit** is a permanent termination of the current working shell, as opposed to a temporary escape (fork) which keeps the escaped shell process active until the user returns to it.

If the current working shell is the user's login shell, **exit** executes the user's .logout file (if it exists) and logs the user off the system. If the ignoreexit variable is set, the error message:

Can't exit, ignoreexit is set

is returned. (See Section 7.1 on Predefined C Shell Variables)

If the ignoreeof is set, a control-D command returns the error message:

Use "exit" to logout.

In either case, if the current shell is not the login shell, a **logout**(1) command returns the error message:

Not login shell.

If both ignoreexit and ignoreeof variables are set, and the current shell is not the login shell, the only way to exit that shell is to **unset** one of the variables. The command:

unset ignoreeof

permits the **control-D** (the "end-of-file" eof) to terminate the current shell. The command

unset ignoreexit

permits the **exit** command to terminate the current shell.

The **exit** command by itself leaves the current shell with the value of the status variable.

This value is shown with the command:

echo $status

This command displays the number status code of the current shell, "0" is the normal exit status, "1" or any non-zero number constitutes an abnormal exit status -- e.g. if a command fails.

With a number argument, **exit** leaves the shell setting the status variable to the specified number N. This is useful in tracing the progress of C shell scripts. The parenthesis surrounding the N are necessary.

See **logout**(1)

The notion of shells and the process of forking new shells is usually very confusing to new users. The illustration below is an attempt to present a graphic representation of various levels of shell interaction.

The situation depicted in the illustration could arise in the following manner:

(1) The user logs in. This is the first, or login shell.

(2) The user enters the editor **vi** to edit a file. In so doing, the user has forked a new shell -- it is this new shell which is running **vi**. The new shell dies as soon as the user leaves the **vi** program.

In this case, someone else writes a message to the user within the **write** program. The user wants to **write** back to this other person without leaving the **vi** editor.

(3) From within the **vi** program, the user forks a temporary escape with either the **:csh** command, or the **:!write** command (see The Ex Reference Manual).

(4) It is possible to fork yet another shell from within the **write** program, e.g. to use the desk calculator program **dc**.

(5) Upon leaving each of these programs, the shell which runs that program dies until the user exits the login shell which logs the user off the system.

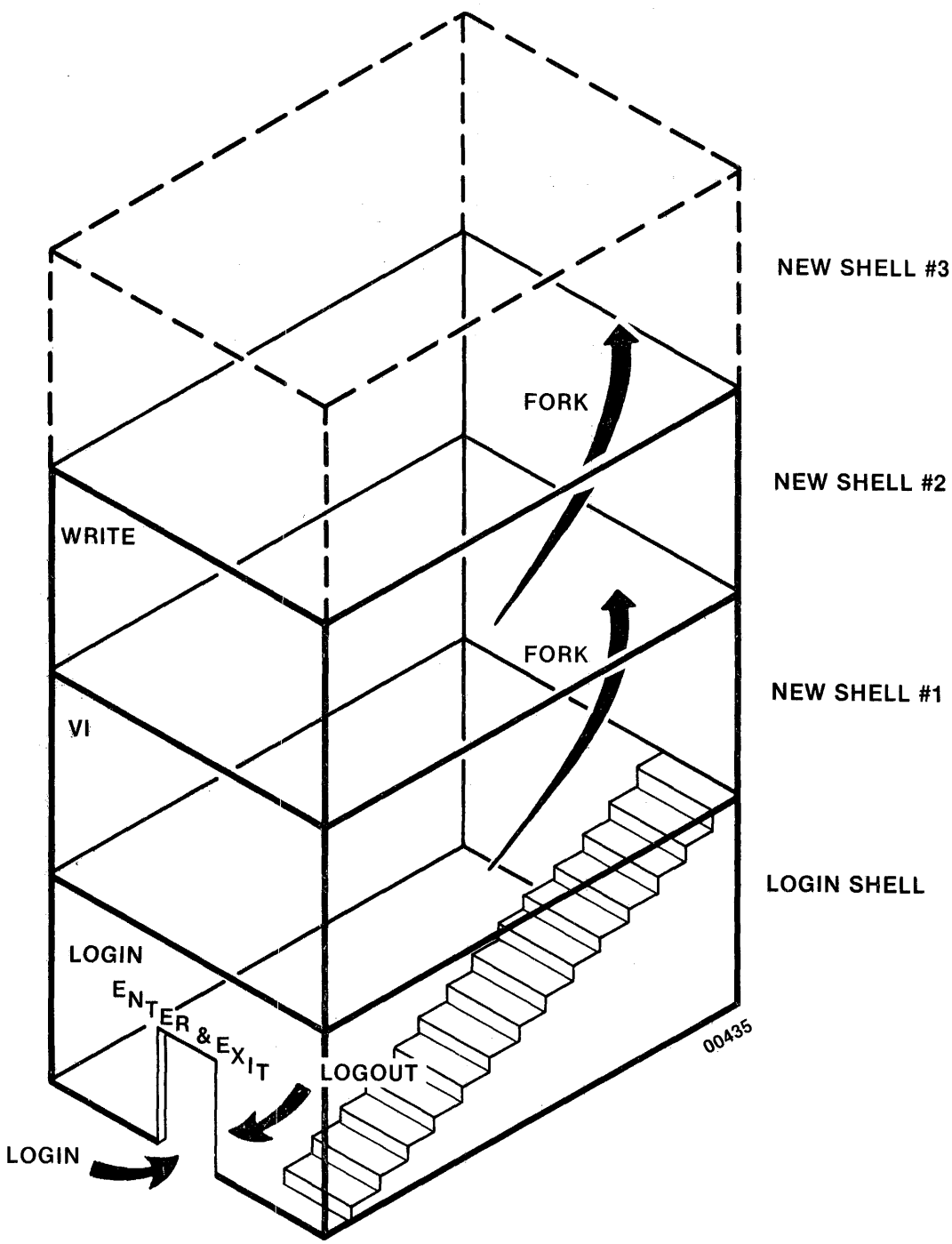


Figure 5-1 Representation of the Fork Process

5.3.3. logout:

SYNOPSIS:

logout

Logout terminates a login shell, executing the contents of the ~/.logout file (if it exists). Especially useful if **ignoreeof** is set, inhibiting a control-D command.

Logout can only be executed from the login shell, if the current shell is any other shell, the error message:

Not login shell.

returns.

ALSO SEE:

Section 9.2.1 -- The ~/.logout file

5.3.4. set / unset:

SYNOPSIS:

set
set variable=word
set variable=(wordlist)
set variable[index]=word

unset variable

The **set** command without an argument shows the value of all shell variables in the following format:

```
S               /z/deck/Util/Sh.1
U               /z/deck/Util/New.csh
argv            ()
exinit          set number wm=20 | version
history         50
home            /z/deck
ignoreeof
ignoreexit      1
mail            /usr/spool/mail/deck
path            (. /usr/bin /bin /z/deck/bin /etc)
prompt          deck # ! >
shell           /bin/csh
status          0
term            vt100
```

Variables that have other than a single word as value print as a word list enclosed in parentheses.

C Shell variables can be established with the command:

set variable=word

which sets variable to word. Variable can be a predefined C Shell variable (See Section 7.1 Predefined C Shell Variables), or a user-defined variable.

Variables can also be set to a list of words, like an array with the syntax:

set variable=(wordlist)

For example, the variable X can be set to all the filenames beginning with the letters "csh" in the current directory with the command:

set X=csh*

Verification that X contains the list comes from the command:

echo $X

which produces output in the following format.

csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.Ø6 csh.9B

Each component part of this list can be addressed with a bracketed subscript, as in the command:

echo $X[3]

which **echos** the third element of the list:

csh.Ø3

These individual elements of a list can be altered in the same way that they are addressed -- with a subscript value. The command syntax is:

set variable[N]=word

which sets the Nth component of variable to word; for example, the following command resets the value of the 3rd component of the X variable to the word test:

set X[3]=test

this component must already exist.

Verification can be seen in the following commands:

echo $X

reports all the elements in the variable X (which is an array of words):

csh.Ø1 csh.Ø2 test csh.Ø4 csh.Ø5 csh.Ø6 csh.9B

The command:

echo $X[3]

reports the value of the 3rd element of that array:

test

Variables are removed from the variable list with the **unset** command:

unset variable

These **set** arguments can be repeated to set multiple values in a single set command. Variable expansion happens for all arguments before any setting occurs.

ALSO SEE:

Section 7 -- Shell Variables, and
Section 3.2 Metacharacters ([, and])

5.3.5. setenv / env:

SYNOPSIS:

setenv NAME=value
env

The **setenv** command works like the **set** command, it sets environment variables while **set** sets shell variables. See Section 1Ø for a discussion of the environment and its variables.

The command

setenv NAME=value

sets the value of environment variable NAME to value,.

Predefined environment variables are:

LOGNAME	Login name
EXINIT	Ex editor initialization variables
HOME	Home directory
PATH	Search path for commands
SHELL	Shell being used
TERM	Type of terminal
TZ	Timezone

Env prints the values of the environment variables currently set. Refer to Section 1Ø for a description of the environment and its environment variables.

ALSO SEE:

env(1)
Section 1Ø -- The Environment

5.3.6. source:

SYNOPSIS:

source file.name

The shell reads commands from file.name and implements them in the current shell (as opposed to forking a new shell).

The **source** command implements changes made to the .login and .cshrc files. **Source** output cannot be re-directed.

ALSO SEE:

Section 1Ø for a discussion of how the C
Shell executes commands.

5.3.7. unalias / alias:

SYNOPSIS:

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed with the command:

unalias *

ALSO SEE:

Section 5.3.1 -- alias / unalias

5.3.8. unset / set:

SYNOPSIS:

unset pattern

All variables whose names match the specified pattern are removed. All variables are removed with the command:

unset *

this can have undesirable side-effects.

ALSO SEE:

Section 5.3.4 -- set / unset

5.3.9. The At Sign -- @:

SYNOPSIS:

@ variable=(number operator number)

The at sign sets variables that use math functions. The variable holds the product of the math function rather than the operative string itself.

For example the command:

set x=(6 + 6)

yields the results of an **echo $x** command as:

6 + 6

```
@ x=( 6 + 6 )
```

produces the results:

```
12
```

Note the at sign is sensitive to syntax. The spaces separating the numbers are essential.

Table 5-2 Built-in Command Summary -- Group 2

GROUP 2	
ENVIRONMENTAL COMMANDS USEFUL FROM THE PROMPT:	
alias	Substitute word for long command
env	Print the current environment variables
printenv	Print the current environment variables
exit	Terminate a shell (logout)
logout	Exit from the login shell
set	Establish a shell variable
setenv	Establish an environment variable
source	Execute a script in the current shell
unalias	Remove an alias
unset	Unset a shell variable
@	Like "set" but uses math functions

SECTION 6
THE C SHELL PROGRAMMING LANGUAGE STRUCTURE

The C Shell can be considered a programming language because of the powerful flow control mechanisms it supports.

The syntax of the C Shell language is much like that of the C Programming Language. (See The C Programming Language by Brian Kernighan and Dennis Ritchie). Table 6-1 summaries the commands in the C Shell programming language.

6.1. foreach and end group

SYNOPSIS:

foreach name **(** list **)**

command

end

When the **foreach** command is typed at the prompt, and the name and list are typed with the appropriate syntax, a new prompt (a question mark) appears to indicate that a C shell loop is in progress.

At the prompt ("?") one or more command statements can be entered. The loop is initiated when the word **end** is typed on a line by itself at the "?" prompt. At that point, the word list is expanded (if it is a magic character) and the command is executed once for each element in the list. Execution continues until the list is exhausted.

The programming structure of the foreach loop (and all control structures of the C shell) makes use of conventional C programming syntax. The example below demonstrates a simple **foreach** loop and the data it produces.

```
% foreach i ( 1 2 3 4 )
? echo $i
? end
1
2
3
4
```

In this example the arbitrary variable "i" is first set to the character "1", then the command **echo $i** is executed. When the **end** statement is encountered, control passes back to the **foreach** statement which checks to see if there are any more items in list. If there is another item, "i" is set to that item and the loop repeats until there are no more items in the list between the parenthesis.

In each successive pass of the loop, **"$i"** refers to the current value of the "i" as established in the **foreach** statement. At the first iteration of the loop, **i** is set to the character "1". At the second iteration, **i** is set to the character "2", and so on.

Any character or string can be used for a name, and any character or string can be used for list. Magic characters will expand unless quoted.

For example, the command:

```
foreach i (*)
```

will expand " $i " to each file name in the current working directory. One example of this process might be:

```
% foreach i (csh.??)
? echo $i
? ls -l $i
? end
```

This command will echo the name of the files starting with "csh." followed by 2 single characters, and return a long listing of that file for each matching file in the current directory, producing results similar to:

```
csh.Ø1
-rw-r--r-- 1 deck  system  13620 Nov  4 15:Ø5 csh.Ø1
csh.Ø2
-rw-r--r-- 1 deck  system  14537 Nov  4 13:24 csh.Ø2
csh.Ø3
-rw-rw-r-- 1 deck  system  24776 Nov  4 15:11 csh.Ø3
csh.Ø4
-rw-rw-r-- 1 deck  system   8996 Nov  4 11:Ø6 csh.Ø4
csh.Ø5
-rw-r--r-- 1 deck  system   6631 Nov  3 17:Ø4 csh.Ø5
csh.9A
-rw-rw-r-- 1 deck  system   3481 Nov  4 14:42 csh.9A
csh.9T
-rw-r--r-- 1 deck  system   2995 Nov  3 17:14 csh.9T
```

Both **foreach** and **end** must appear alone on separate lines.

The built-in command **continue** can be used to continue the loop prematurely and the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read through before any statements in the loop are executed.

See Example 1 in Section 6.7

6.2. while and end group

SYNOPSIS:

while (expression**)**

command

end

While the specified expression evaluates nonzero, the commands between the **while** and the matching end are executed. **Break** and **continue** can be used to terminate or continue the loop prematurely. The **while** and **end** must appear on separate lines. Prompting occurs here the first time through the loop as for the foreach statement if the input is a terminal.

See Example 2 in Section 6.7

6.3. The if, else, endif Group

SYNOPSIS:

```
if (expression.1) then
        command.1
else if (expression.2) then
        command.2
else
        command.3
endif
```

If is a loop control statement generally useful for making decisions within the while and foreach loop structures.

If expression.1 evaluates true, command.1 is executed. Command.1 must be a simple command, not a pipeline, a command list, or a command list within parentheses. Input/output redirection occurs when the command is executed even if expression.1 is false (this is a bug).

If expression.1 is not true, the else if condition is tested, and if expression.2 is true, the commands in command.2 to the second else are executed. This process continues down the script.

Any number of else-if pairs are possible; only one endif is needed. The else part is likewise optional. The words else and endif must appear at the beginning of lines; the if must appear at the beginning of a line or after an else.

See Example 3 and Example 4 in Section 6.7

6.4. The Switch Group

SYNOPSIS:

```
switch (string)
        case label1:
                command
        breaksw
        case label2:
                command
        breaksw
        default
                command
endsw
```

Switch is generally useful in the context of a foreach or while statement.

Each case label is successively matched against the specified string. If the case label matches the string, the associated command is executed.

The file metacharacters "*, ?, [, and]" can be used in the case labels. If none of the labels match before a default label is found, the execution begins after the default label.

Each case label and the default label must appear at the beginning of a line. The command **breaksw** causes execution to continue after the **endsw.** Otherwise, control falls through case labels and default labels, as in C. If no label matches and there is no default, execution continues after the **endsw.**

See Example 5 in Section 6.7

Table 6-1 summarizes the third group of commands.

Table 6-1 Built-in Command Summary -- Group 3

GROUP 3

LOOP CONTROL COMMANDS USEFUL WITHIN A SCRIPT:

foreach	Initiate a **foreach** loop
end	End of a **foreach** or **while** loop
while	Initiate a **while** loop
end	End of a **foreach** or **while** loop

The If group:

if	Initiate an **if** loop
else	Alternative decision in an **if** statement
endif	End of an **if** loop

The Switch group:

switch	**Switch** to the next iteration of the variable
case	Label in a **switch** statement
breaksw	Causes a break from a **switch**
default	Default case in a **switch** statement
endsw	End of an **switch** loop

Independent loop control commands:

break	Drops out of the nearest loop
continue	Continue execution of nearest loop
goto	Jump to a new location
shift	Go to the next argument in the argument variable

6.5. Independent Flow Control Statements

6.5.1. break:

SYNOPSIS:

break

Causes execution to resume after the **end** of the nearest enclosing **foreach** or while. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

See Example 6 in Section 6.7

6.5.2. continue:

SYNOPSIS:

continue

Continue execution of the nearest enclosing **while** or **foreach.** The rest of the commands on the current line are executed.

See Example 7 in Section 6.7

6.5.3. goto:

SYNOPSIS:

goto word

The specified word is file name and command expanded to yield a string of the form label. The shell rewinds its input as much as possible and searches for a line of the form label: possibly preceded by blanks or tabs. Execution continues after the specified line.

See Example 8 in Section 6.7

6.5.4. shift:

SYNOPSIS:

shift
shift variable

The members of argv are shifted to the left, discarding argv[1]. It is an error for argv not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

See Example 9 in Section 6.7

6.6. Independent Shell Script Commands

6.6.1. exec:

SYNOPSIS:

exec command

The specified command is executed in place of the current shell.

For example, the command:

exec date

executes the **date** command, and then terminates the shell. If the command is executed from the login shell, the command will log the user off the system. The .logout file (if it exists) will not execute.

6.6.2. nohup:

SYNOPSIS:

nohup
nohup command

In a dial-up situation, when a terminal is connected to the system with a modem through the telephone lines, hanging up the telephone from the terminal end results in a logout.

Nohup (no hang-up) cause telephone hangups to be ignored for the remainder of a shell script allowing the script to continue. The shell script must be running detached (in background).

The **nohup** command, with a command argument causes the specified command to be run with hangups ignored.

6.6.3. onintr:

SYNOPSIS:

onintr -
onintr label

The **onintr** (on interrupt) command controls the action of the shell script when an interrupt signal (usually the DELETE key) is encountered from the keyboard.

With a minus sign argument, all interrupts are ignored.

With a label argument, **onintr** causes the shell to execute a **goto** label when an interrupt is received.

See Example 4 in Section 6.7

The following table summarizes the fourth group of commands.

Table 6-2 Built-in Command Summary -- Group 4

GROUP 4	
GENERAL PURPOSE COMMANDS, USEFUL WITHIN A SCRIPT:	
exec	Causes execution of a command with no return
nohup	No Hangup in a dial-up phone situation
onintr	Goto a new label on receiving an interrupt signal

6.7. Example Shell Scripts

NOTE

In these shell script examples, words in bold are the actual command text. Lines beginning with a pound sign are comments only and may be omitted from the script.

```
# EXAMPLE 1 -- Foreach
#
foreach i (*)       # The variable "i" is set to all
                    # the filenames in the current
                    # working directory

     echo $i        # within the loop, each iteration
                    # of the variable "i" is printed
                    # on the screen

end                 # end of the loop
```

Figure 6-1 A Basic Foreach Loop

This C shell script produces output in the following format:

```
csh.01
csh.02
csh.03
csh.04
csh.05
csh.06
csh.07
csh.08
csh.09
temp
```

```
# EXAMPLE 2 -- While
#
while ( 1 )          # "1" is always true, therefore
                     # this is an endless loop

      echo "This is an endless loop"

                     # The string is printed forever
                     # or until it is interrupted

end                  # end of the while loop
```

Figure 6-2 A Basic While Loop

This C Shell script produces the following output:

```
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
This is an endless loop
.
.
.
```

until an interrupt (the DELETE key) is encountered.

```
# EXAMPLE 3 -- If
#
foreach i ( * )     # The variable "i" is
                    # set to all filenames
                    # in the current working directory

    if ( $i == temp ) then

                    # If this iteration of "$i" is
                    # a file named "temp", then do
                    # the following:

      echo "Here is the temp file"

                    # Print the string

    else
                    # If "$i" is not temp, then do
                    # the following:

      echo $i
                    # Print the filename

    endif           # Necessary end of the conditional
                    # "if" statement

end                 # End of the "foreach" loop
```

Figure 6-3 An If Statement in a Foreach Loop

The above C Shell script produces output in the following format:

```
csh.01
csh.02
csh.03
csh.04
csh.05
csh.06
csh.07
csh.08
csh.09
Here is the temp file
```

```
# EXAMPLE 4, An "if" conditional statement within a "while"
# loop, -- "onintr" and "set" using the math
# statement "@" are also demonstrated
#
onintr hook         # Establishes "hook" as the label
                    # to "goto" on interrupt

set a=Ø             # Initializes the variable "a"
                    # to "Ø" at the beginning of
                    # the "while" loop

while ( 1 )         # "1" is always true, this is
                    # an endless loop

    if ( $a < 5 ) then

                    # If the variable "a" is less
                    # than 5, then perform the next step

      echo "The number is less than 5"

                    # Print the string
    else
                    # If the number is not less than 5
                    # then perform the next step:

      echo "The number is 5 or greater"

                    # Print the string

    endif           # End of the "if" conditional statement

@ a++               # Set "a" to "a+1" (increment "a")

end                 # End of the "while" loop

hook:               # the label identified in the
                    # "onintr" statement. If an interrupt
                    # is encountered, perform the following:

      echo "Interrupt encounter -- Good bye"

                    # Print the string
```

Figure 6-4 An Enhanced If Statement

The above C Shell script produces output in the following format:

```
The number is less than 5
The number is less than 5
The number is less than 5
The number is less than 5
The number is less than 5
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
The number is 5 or greater
.
.
.
Interrupt encounter -- Good bye
```

```
# EXAMPLE 5, A "switch" statement nested in a "foreach"
# loop -- use of the metacharacter "?" is also demonstrated

foreach i (*)           # Sets the variable "i" to all
                        # the filenames in the current
                        # directory.

    switch ( $i )       # Check this iteration of "$i"
                        # to see if it meets the following
                        # conditions.

      case ????:        # If the filename in "$i" has four
                        # characters, perform the following:

          echo " $i is a four character name "

                        # Print the string

      breaksw           # And exit out of this case test.

      case ?????:       # If the filename in "$i" has five
                        # characters, perform the following:

          echo " $i is a five character name "
                        # Print the string

      breaksw           # And exit out of this case test.

      case ??????:          # If the filename in "$i" has six
                        # characters, perform the following:

          echo " $i is a six character name "
                        # Print the string

      breaksw           # And exit out of this case test.

      default           # if the filename in "$i" does not match
                        # any of the above criteria, perform the
                        # following:

          echo " $i is not a four, five or six character name"

                        # Print the string

      endsw             # And exit out of the whole switch loop

end                     # End of the "foreach" loop.
```

Figure 6-5 A Switch Statement in a Foreach Loop

The above C shell script produces output in the following format. (NOTE: output will vary with the filenames in the current working directory).

```
OUTØ1 is a five character name
OUTØ2 is a five character name
OUTØ3 is a five character name
a.out is a five character name
all.tables is not a four, five or six character name
csh.Ø1 is a six character name
csh.Ø2 is a six character name
csh.Ø3 is a six character name
echo.test is not a four, five or six character name
tax is not a four, five or six character name
temp is a four character name
```

```
# EXAMPLE 6: A Break statement in a While loop
#
while (1)
                    # This sets up an endless loop
    echo -n " enter x: "
                    # The echo statement prompts
                    # for input from the the terminal

    set x = `gets`
                    # the "set" expression sets the
                    # variable "x" to whatever is
                    # entered at the terminal (the
                    # input is captured with the
                    # `gets` expression)

    if ( $x == 'a' ) then
                    # If the input is the letter
                    # "a" then proceed to the break
                    # statement.

      break # goes to the "end" statement
                    # echos the last statement "it
                    # broke" and drops out of the loop

    else            # If the input is not the letter
                    # "a" then proceed back to the
                    # nearest loop statement (while).

      echo "it did not break"
                    # returns to the
                    # beginning of the while loop

    endif
                    # ends the "if" branch

end
                    # ends the "while" loop

echo "it broke"
                    # demonstrates the break
```

Figure 6-6 A Break Statement

The above C Shell script produces an exchange in the following format:

```
 enter x: b
it did not break
 enter x: c
it did not break
 enter x: a
it broke
```

NOTE: that the "b", "c", and "a" characters are input from the keyboard in response to the "enter x:" prompt.

```
# EXAMPLE 7 -- Prompting for input, getting input, evaluating
#  the input with an if statement, and demonstrating
#  the continue statement. Onintr is also demonstrated.
#
while (1)
                    # sets up an endless loop

      echo -n "enter x:"
                    # as in the example above, this
                    # prompts for input from the
                    # terminal

      set x = `gets`
                    # this set the variable "x" to
                    # whatever is entered with the
                    # `gets` command

      if ( $x == 'a' ) then
                    # If the input is "a" continue

            echo "it continued"
                    # This demonstrates the continuation
                    # if the input is "a"

            continue
                    # goes to beginning of the enclosing
                    # while loop and starts over

      endif
                    # ends the "if" statement if
                    # the input is not "a"

            echo "it did not continue"
                    # the if statement does not continue

            exit
                    # if the input is not "a" the
                    # "exit" command terminates the
                    # "while" loop

end
                    # end the "while" loop
```

Figure 6-7 An Example of the Continue Statement

The above C shell script produces an exchange in the following format:

```
enter x:a
it continued
enter x:a
it continued
enter x:a
it continued
enter x:b
it did not continue
```

NOTE: As in the previous example, the "a" and "b" characters are input from the keyboard.

```
# EXAMPLE 8 -- The Goto statement within a foreach
# loop. This shell script tests each filename for the
# name "temp". When the "temp" file is encountered, control
# goes to the label "branch".

foreach i ( * )
          # sets a "foreach" loop with the controlling
          # list variable "i" which is set to all the
          # files in the current directory.

     if ( $i == temp ) then
          # tests each filename to see if it matches
          # the word "temp"

          goto hook
               # if it matches, control jumps
               # to the label "hook"

     else if ( $i != temp)
          # if the filename does not match the
          # word "temp" control drop to the next
          # statement

          echo "$i is not the temp file "
               # echos that the filename is
               # not "temp"

     endif
          # ends the "if" branch statement

end
          # ends the "foreach" loop

hook:
          # the destination of the "goto" label

     echo "Here is the temp file -- end loop "
          # reports finding the "temp" file and
          # drops out of the loop
```

Figure 6-8 An Example of the Goto Statement

The above C shell script produces output in the following format. Note that the output is dependent upon the filenames in the current working directory.

```
csh.Ø1 is not the temp file
csh.Ø2 is not the temp file
csh.Ø3 is not the temp file
csh.Ø4 is not the temp file
csh.Ø5 is not the temp file
csh.Ø6 is not the temp file
csh.Ø7 is not the temp file
csh.Ø8 is not the temp file
Here is the temp file -- end loop
```

```
# EXAMPLE 9 -- The Shift statement

#
set a = (*)
      # sets the variable "a" to the list of all filenames
      # in the current working directory. If the files
      # are "csh.Ø1 csh.Ø2 csh.Ø3 and csh.Ø4" then
echo $a
      # the list will echo:
      # csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4
shift a
      # shifting drops the leftmost element to produce:
echo $a
      # csh.Ø2 csh.Ø3 csh.Ø4
shift a
      # shifting drops the leftmost element to produce:
echo $a
      # csh.Ø3 csh.Ø4
```

Figure 6-9 An Example of the Shift Statement

The output will look something like the following:

```
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4
csh.Ø2 csh.Ø3 csh.Ø4
csh.Ø3 csh.Ø4
```

SECTION 7
SHELL VARIABLES

7.1. Predefined C Shell Variables

The ZEUS working environment can be customized in a number of ways, this section explains the use of C shell variables.

17 variable names are predefined by the C Shell. These variables control many of the C shell's built-in functions.

Section 7.3 presents a discussion of user-defined variables.

7.1.1. argv:

SYNOPSIS:

(Not set at the terminal)

Argv is short for "argument variable". Each command typed to the prompt is broken into arguments (parsed) and each argument in the command is numbered from zero and placed in the argument variable for execution. In the command:

ls -l file.Ø1

there are 3 arguments:

argument Ø is the command itself, **ls**
argument 1 is the **-l**
argument 2 is the filename file.Ø1

The C shell keeps track of the arguments in the the variable **argv ()** At login, **argv** is set to zero by the C shell. This value is reset at each command to the names of the arguments given for each command.

To demonstrate the **argv** variable, the following file named test is a C shell script and contains six lines:

```
# test
echo $argv
echo $argv[1]
echo $argv[2]
echo $argv[3]
echo $argv[4]
```

The first line of the file contains a pound sign (to indicate that it is a C shell script) and the name of the file test.

The rest of the file contains command lines designed to demonstrate the properties of the argv variable. The second line is a command to **echo** the full contents of the argv variable, the second line is a command to **echo** argument 1 of the argv variable, the third line is a command to **echo** argument 2, etc.

Once the file has been created it must be made executable with the command:

```
chmod 777 test
```

(See **chmod**(1) in the ZEUS Reference Manual) and then executed with three arguments as follows:

```
test a b c
```

producing the following results:

```
a b c
a
b
c
Subscript out of range.
```

the script command **echo $argv** prints out the full contents of the argv variable -- the first argument to the last argument in argv (all arguments except the zero-th argument). The expression argv[*] can also be used. The following **echo** commands print out the specific components of the command arguments -- the first, second, and third arguments. Note that a call to the fourth argument (**$argv[4]**) produces the error:

```
Subscript out of range.
```

Each component of **argv** can also be accessed with the syntax:

$N

where N is a number corresponding to the position of the argument in the argument list. Thus the script could have been written:

```
# test
echo $*
echo $0
echo $1
echo $2
echo $3
echo $4
```

Executing the script again with three arguments:

```
test a b c
```

produces the following results:

```
a b c
test
a
b
c
```

The difference is two-fold. First, a call to the command itself (argument number zero) **$0** is possible, and second, a call to a subscript value that is out of the range of the number of arguments does not produce an error, only a blank line.

DEFAULT:

```
argv=()
```

ALSO SEE:

Section 5.2.2 -- echo and 5.2.9 -- umask

7.1.2. child:

SYNOPSIS:

(Not set at the terminal)

The **child** variable holds the number of the last background process.

This is useful in stopping a job running in background. The command:

```
kill -9 $child
```

will terminate the last background job.

DEFAULT:

unset by default

ALSO SEE:

Section 2.5 -- Running a Command in Background

7.1.3. echo:

SYNOPSIS:

set echo

The **echo** variable controls whether or not commands are echoed (printed) immediately after they are typed at the prompt. When set, the echo variable produces results in the following format:

```
% ls
ls
csh.01  csh.02  csh.03  csh.04 csh.05
```

The echo variable is also set when the " **-x** " command line option is given to a **csh**(1) command. As in the command:

csh -x test

For non-built-in commands, all expansions occur before **echo**-ing. Built-in commands are **echo**ed before command and file name substitution.

DEFAULT:

unset by default

ALSO SEE:

echo(1), echo2(1)
Section 5.2.2 -- echo

7.1.4. history:

SYNOPSIS:

set history=N

The **history** variable controls the number of commands stored in memory on the history list. Numbers that are too large can run the C shell out of memory. The last executed command is always saved on the history list.

DEFAULT:

unset by default

ALSO SEE:

Section 4 -- The History Function

7.1.5. home:

SYNOPSIS:

set home=/path/home.directory

The home variable refers to the home directory. It is set (initially) by an entry in the /etc/passwd file which is created when the account is created. It can be reset at the prompt, or in a script.

The **home** variable is used to establish the destination for the **cd** command (when used without an argument). It is also used to establish the value of the metacharacter tilde " ~ ".

Home can be set to any directory, but it is most useful when it points to the home directory.

DEFAULT:

home=/path/home.directory

ALSO SEE:

Section 10 -- Environment Variables (HOME)
Section 3 -- Metacharacters (~) and (/)

7.1.6. ignoreeof:

SYNOPSIS:

set ignoreeof

The ignoreeof variable determines how the C shell handles end-of-file signals (control D) from the terminal.

If it is set, the ignoreeof variable prevents the parent (login) shell from being killed by accidental control-Ds.

If ignoreeof is set, and a control-D is entered, an error message returns with:

Use "logout" to logout.

This variable is useful in programs where control-Ds must be entered at the terminal.

DEFAULT:

unset by default

ALSO SEE:

Section 5 -- Built-in Commands (logout, exit)

7.1.7. mail:

SYNOPSIS:

set mail=/path/directory
set mail=(N /path/directory)

Note that parentheses must surround a word list with embedded spaces.

The mail variable sets a procedure that checks the directory /path/directory every N seconds for new mail.

If N is omitted, the shell checks the file every 5 minutes.

Checking is done after each command past N seconds (thus no checking is done in a lengthy program like **vi**(1); instead, the check is performed after leaving **vi**).

Upon finding new mail, the C shell reports

You have new mail.

Several files can be specified, and if there are multiple mail files, the C shell specifies the mail file name with:

New mail in name.

DEFAULT:

mail=/usr/spool/mail/name

ALSO SEE:

mail(1)

7.1.8. noclobber:

SYNOPSIS:

set noclobber

If the noclobber variable is set, restrictions are placed on output redirection to insure that files are not accidentally destroyed. An attempt to redirect output to an existing file (e.g. test) as with the command:

who > test

results in the error message:

test: File exists.

In addition, restrictions are placed on appending (" >> ") redirections to insure that the named output files refer to existing files. An attempt to append information to a non-existent file (e.g. new.file), as with the command:

ls >> new.file

results in the error message:

new.file: **No such file or directory.**

DEFAULT:

unset by default

ALSO SEE:

Section 2 -- Input and Output Redirection

7.1.9. noglob:

SYNOPSIS:

set noglob

If set, the noglob variable inhibits file name expansion -- the metacharacters described in Section 3.1 will not expand to matching filenames. The command:

echo *

will return:

*

This is useful in C shell scripts not dealing with file names, or in a situation where metacharacters need to be passed unexpanded.

DEFAULT:

unset by default

ALSO SEE:

Section 3 -- Filename expansion

7.1.1Ø. nonomatch:

SYNOPSIS:

set nonomatch

With the nonomatch unset, a command using a filename expansion metacharacter that fails to match a filename as in a situation where there are no files that begin with the

letter "a". The command:

ls a*

returns the error:

No match.

but if the nonomatch variable is set, it is not an error for a file name expansion to not match any existing files.

Instead, the pattern is returned with the message:

a* not found.

It is still an error for the primitive pattern to be malformed; for example the command:

echo [

still returns the error:

Missing].

DEFAULT:

unset by default

ALSO SEE:

Section 3 -- Filename Substitution

7.1.11. path:

SYNOPSIS:

set path=/directory...

At login, the shell searched down the directories specified in the path to create a hash table of the files listed in each directory. This hash table becomes the list of commands that are known to the shell.

This is true, except for dot " . " which specifies the current working directory. Dot, (the current working directory) if listed in the path variable, is always searched and hashed for each command. Thus, it should always be included

in the path variable.

If there is no path variable, only commands which specify a full path name will execute as in the command:

/bin/ls

The default search path is ., /bin, and /usr/bin. For the super-user, the default search path is /etc, /bin, and /usr/bin.

The C shell will search first in the current working directory (indicated with a dot " . "), if the C shell finds a file name that is identical to the name of the command, the C shell will attempt to execute the file as if it were a program. If the file does not execute properly, the C shell will report the error on the standard error channel (which is usually the terminal) with the syntax:

command: **Command not found.**

If the file is not found in the current directory the C shell looks in the next directory (in this case /bin). The /bin directory holds the most common ZEUS commands. If the command/file is found in /bin it is executed, if it is not, the C shell searches in /usr/bin where the "next-most-common" ZEUS commands reside.

Other directories can be included in the search path. If the path contains spaces, it must be surrounded with parentheses.

DEFAULT:

path=(. /bin /usr/bin)

ALSO SEE:

Section 5 -- Built-in Commands (Rehash)

7.1.12. prompt:

SYNOPSIS:

set prompt=string

The prompt is the signal from the C Shell that the operating system has finished the last command and is ready to accept

another command.

The default prompt for regular users is the percent sign " **%** ", the default prompt for the super-user (ZEUS) is a pound sign " **#** "

The **prompt** variable can be modified to provide more useful information. The most common example is to place the command number within the prompt. The command:

set prompt=" % \! "

will produce the prompt:

% 1

Note that quotes must be used if the prompt string contains a space.

The number will increment by 1 for each command. The command number is useful information when used with the **history** function (See below).

DEFAULT:

prompt=%

ALSO SEE:

Section 4 -- The History Function

7.1.13. shell:

SYNOPSIS:

set shell=/path/shell.name

The shell variable sets the shell to be used at login, either /bin/csh for the C Shell, or /bin/sh for the Bourne Shell.

This variable is set (initially) in the /etc/passwd file when the account is created and can be reset either at the prompt or in the body of a C shell script.

DEFAULT:

shell=/bin/csh

ALSO SEE:

csh(1) and sh(1)

7.1.14. status:

SYNOPSIS:

(Not set at the terminal)

The status variable holds the exit status returned by the last command.

When a command executes successfully (without error), it sets the status variable to Ø.

If a command fails to execute properly (e.g. if there is a syntax error, or some other form of error), the command will leave the status variable with some value other than Ø.

This variable is useful in C shell scripts to report errors on the execution of C shell script commands.

Note: The command **set** will always show the status variable to have a value of Ø because the **set** command executes successfully. The true value of the status variable can be seen with the command:

echo $status

DEFAULT:

status=Ø

ALSO SEE:

Section 5 -- Built-in commands (exit)

7.1.15. term:

SYNOPSIS:

set term=terminal.type

The term variable sets the type of terminal expected by the

system. This information is important to programs that use specific commands to manipulate the cursor (e.g. the visual editor **vi**(1)).

The shell matches the two-character code from the term variable with the terminal description line in the /etc/termcap file. A sample of the first line of the terminal description code for the **VTZ 2/1Ø** is shown below:

```
vz|vtz|mcz-2/6Ø|vtz-2/1Ø
```

The **term** variable is initially set in /etc/ttytype file but can be reset at the prompt or in the body of a C Shell Script.

Using the two-character code found in the **term** variable, the C shell then matches that code to the entry in the /etc/termcap file to initialize the "terminal-to-ZEUS" software interface.

DEFAULT:

```
term=vz
```

ALSO SEE:

The /etc/termcap file on-line
and termcap(5).

7.1.16. time:

SYNOPSIS:

set time=N

The time variable controls the automatic timing of commands. If it is set, any command which takes more than "N " CPU seconds results in a line showing user time, system time, real time, and a utilization percentage (ratio of user time plus system time to real time).

The output is in the following format: (output of the time command is the last line of the example)

```
% ps
   PID TTY TIME CMD
 23399 2   Ø:26 -csh
 252Ø7 2   Ø:Ø3 ps
Ø.2u 2.8s Ø:Ø5 6Ø%
```

DEFAULT:

unset by default

ALSO SEE:

time(1).
Section 5 - Built-in commands (time)

7.1.17. verbose:

SYNOPSIS:

set verbose

The verbose variable functions like the echo variable. When it is set, each command is echoed, as in the following example:

```
% ls
ls
csh.Ø1 csh.Ø2 csh.Ø3 csh.Ø4 csh.Ø5 csh.Ø6 csh.Ø7
```

The verbose variable is also set by the " **-v** " command line option to the **csh** command, as in:

csh -v test

the verbose variable causes the arguments of each command to be printed after history substitution (unlike the echo variable which causes arguments to be printed before history substitution).

DEFAULT:

unset by default

ALSO SEE:

Section 3 -- Filename Substitution
Section 4 -- History Function

7.2. Predefined Variables -- Default Values

If no other values are established for these initial C shell variables, the following "default" properties are set by the C shell (or a function of the C shell) at login.

The **set** command reveals the following list of predefined C shell variables and their default value.

```
argv    ()
home    /path/home.directory
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
term    vz
```

Table 7-1 C Shell Predefined Variables

Name:	Function:
argv	Tracks command arguments
child	Records the number of last background command
echo	Echos each command
history	Sets the length of the history memory
home	Sets the home directory
ignoreeof	Sets response to control-D commands
mail	Sets mailbox and frequency of mail checks
noclobber	Sets file over-write protection
noglob	Sets filename expansion inhibitor
nonomatch	Sets "no match" error override
path	Sets search path for commands
prompt	Sets prompt
shell	Sets shell (/bin/csh or /bin/sh)
status	Tracks exit status of commands
term	Sets terminal type
time	Sets frequency of "time" report command
verbose	Sets verbose echoing of command lines

7.3. User-defined Variables

In addition to the built-in variables supplied with the C Shell, non-built-in variables can be established and

manipulated by each user.

A variable can be any character string of characters, and can be set to anything -- numbers, letters, file or directory names, strings of numbers and letters, etc.

One handy trick is to set a directory name to a single letter variable, as in the following example:

set M=/usr/doc/man/man1

With M set, the command:

cd $M

is the same as the command

cd /usr/doc/man/man1

Another trick is to set a variable to the date, as with the command:

set DATE=`date`

The variable DATE can then be manipulated in any number of creative ways.

7.4. User-defined Variable Substitutions

SYNOPSIS:

set name=value
set name[N]=value

Any word or character string can be set as a C shell variable.

Given the command to set a variable named DATE to the date with the command:

set DATE=`date`

to see the contents of the DATE variable, the command:

echo $DATE

produces the output:

Fri Dec 17 17:24:Ø8 PST 1982

NOTE: Variables must be called with a dollar sign before the name.

Each element in the string can be referenced with a subscript selector appended to the variable name as in:

echo $DATE[3]

to produce the third word in the string:

17

To check the number of words in the DATE variable, the command:

echo $#DATE

produces:

6

As part of a sequence of commands in the body of a C shell script it may be useful to determine whether a variable has been set or not, thus the command:

echo $?DATE

will yield a "1" if DATE is set, and "Ø" if it is not set.

Table 7-2 Variable Substitution Syntax

Syntax:	Meaning:
$name	The variable name
${name}	Insulate name from surrounding characters
$name[N]	Argument N of name
${name[N]}	Argument N of name
$#name	Give number of words in the variable
${#name}	Give number of words in the variable
$?name	Substitute "1" if name is set "Ø" if not
$number	Same as $argv[number]
${number}	Same as ${argv[number]}

Table 7-3 Metacharacters in Variable Substitution

Character:	Meaning:
$N	Same as $argv[N]
$Ø	Command file name (zeroth argument)
$?Ø	1 if current input filename is known, Ø if not
$*	Same as $argv[*]
$$	Process I.D. number of parent shell

7.5. Using Modifiers in Variable Substitions

The modifiers "**:h**", "**:t**" and "**:r**", apply to variables the way that they apply to commands.

For example, if the capital letter U is set to the directory /z/deck/Util/New.csh, the following commands apply. First, the command to **echo** the full variable:

```
echo $U
/z/deck/Util/New.csh
```

The **:h** modifier strips off the trailing filename (/New.csh) leaving the head:

```
echo $U:h
/z/deck/Util
```

The **:r** modifier strips off the trailing filename suffix (the csh portion after the dot) leaving the root:

```
echo $U:r
/z/deck/Util/New
```

The **:t** modifier strips off the head (/z/deck/Util) leaving the tail portion of the name:

```
echo $U:t
New.csh
```

If insulating braces " { } " appear in the command form, the modifiers must appear within the braces.

Table 7-4 Variable Substitution Modifier Table

Modifier:	Effect:
:h	Head only
:t	Tail only
:r	Root only

ALSO SEE:

See Section 4.5 -- Modifying Previous Command Words
Section 5 -- Built in Commands (modifiers)

SECTION 8
THE CSH COMMAND AND C SHELL SCRIPTS

8.1. The Csh Command

SYNOPSIS:

csh
csh [-option**]** filename
filename

The C shell can be invoked at the prompt as a command. Used as a command, and without an argument, as in the command:

csh

the parent C shell (the login shell) creates (forks) a new C shell (referred to as a child process), re-reads and re-executes the contents of the ~/.cshrc file (but not the ~/.login file) and begins a new environment with a new history list.

By itself, the **csh** command creates a new working environment with the default values (those established in the environment and in the ~/.cshrc file). This is useful in a situation where the user wishes to clear any new variables, aliases, or history references from the immediate working environment in order to test a shell script or some other shell manipulation without logging off the system and then logging back on. The **csh** command is of greater utility when invoked from within a running program (e.g. **vi**(1), **more**(1) and **write**(1)).

See Section 9.2 -- Other Related C Shell Files

8.2. Invoking Csh to Execute a Shell Script

SYNOPSIS:

csh filename

With a filename argument, where filename is the name of the file containing one or more commands (the file is known as a script or shell script), the C Shell attempts to execute the file.

For example, given a file named test having the following lines:

```
ls
who
pwd
date
```

all commands in the file will be executed in succession with the command

```
csh test
```

to produce the results:

```
csh.Ø1
csh.Ø2
csh.Ø3
csh.Ø4
csh.Ø5
csh.Ø6
csh.9T
refer.sheet

karen    ttyØ     Nov  1 Ø8:1Ø
cheryl   console  Nov  1 14:Ø2
deck     tty2     Nov  1 1Ø:37
mike     tty6     Nov  1 14:43
carol    tty8     Nov  1 Ø8:35
george   tty9     Nov  1 Ø8:35

/z/deck/Util/New.csh

Mon Nov  1 15:19:39 PST 1982
```

the **csh** command creates (forks) a new C shell (also referred to as a child process). This new C shell reads and executes the contents of the ~/.cshrc file and then reads through the contents of the shell script, line by line, and executes each command in its turn (Spaces between the sections of output have been added to clarify the example).

When all the lines have been read, the new C shell (the child) dies and control returns to the parent (login) C shell (See Section 1Ø for a discussion of this process).

Executing a shell script with the **csh** command is equivalent to making the shell script executable with the **chmod**(1) command and typing the name of the file as if it were a command. This is demonstrated in the following example:

chmod 777 filename
filename

8.3. Using C Expressions in Scripts

Shell scripts can become far more complex than a collection of single line commands. A number of built-in commands can be used in expressions using operators similar to those of the C programming language (Refer to the book The C Programming Language by Brian Kernighan and Dennis Richie).

These expressions can be used with the @(set), exit, if, and while commands. Table 8-1 shows the available operators:

Table 8-1 Relational Operators in C Shell Scripts

Character:	Meaning:
||	Logical "or"
&&	Logical "and"
|	Bitwise inclusive "or"
^	Bitwise exclusive "or"
&	Bitwise "and"
== !=	Equal to; Not Equal to
<= >= < >	Less than or equal to, Greater than or equal to, Less than, Greater than
<< >>	Shift operators
+ -	Add, Subtract
* / %	Multiply, Divide, Modulo
!	Negation
~	Complement
( )	Parenthesis -- bracket expressions

Precedence increases from top to bottom. Operators on the same line are left to right associative.

NOTE

The "equal to" and "not equal to" operators (" ==" and " !=") compare arguments as strings; all other operators operate on numbers.

The expression:

```
if $argv[1] == temp
```

will try to match the contents of $argv[1] with a file named temp, while the expression

```
if $argv[1] >= temp
```

will produce a syntax error.

Null or missing arguments are considered "Ø". Thus, if the variable a is set to "1", the expression:

```
if ( $a > ) echo hi
```

will echo "hi".

The result of all expressions are strings which represent decimal numbers.

All operators must be separated from the surrounding text with spaces except the following characters:

ampersand	"&"
pipe	"|"
less than	"<"
greater than	">"
parenthesis	"(" and ")"

which can be placed next to the operands, as in the example:

```
#
set a=4
if ($a>3) who
```

which executes the **who** command.

8.4. Examples of Shell Scripts using Operators

8.4.1. And and Or Operators:

The following operators are useful in conditional expressions where the values of expressions and commands need to be evaluated -- "true" and/or "false".

The following "truth table" illustrates results of these operators:

```
Ø = false
1 = true
a = left side of the operator
b = right side of the operator

                |   &
a b             ||  &&   ^

Ø Ø     Ø   Ø   Ø
Ø 1     1   Ø   1
1 Ø     1   Ø   1
1 1     1   1   Ø
```

Figure 8-1 Truth Table

Using the following syntax:

if (expression.a operator expression.b) command

and with the logical "or" operator "||" if expression.a is "true" -- giving it a value of "1", and expression.b is "false" -- giving it a value of "Ø", then the net result of these two expressions is "true" (giving it a value of "1" -- see the third line of the table above) and an attempt will be made to execute command.

Consider the following shell scripts:

|| Logical "or"

```
#
set a=1
set b=9
if ( $a == 1 || $b == 1 ) who
```

This script executes the **who** command.

&& Logical "and"

```
#
set a=1
set b=9
if ( $a == 1 && $b == 9 ) who
```

This script executes the **who** command.

| Bitwise inclusive "or"

```
#
set a=1
set b=9
if ( $a == 1 | $b == 1 ) who
```

This script executes the **who** command.

^ Bitwise exclusive "or"

```
#
set a=1
set b=9
if ( $a == 1 ^ $b == 1 ) who
```

This script executes the **who** command.

& Bitwise "and"

```
#
set a=1
set b=9
if ( $a == 1 & $b == 9 ) who
```

This script executes the **who** command.

8.4.2. Relational, Equality Operators:

== Equal to

```
#
set a=1
if ( $a == 1 ) who
```

This script executes the **who** command.

!= Not Equal to

```
#
set a=1
if ( $a != 9 ) who
```

This script executes the **who** command.

<= Less than or equal to

```
#
set a=1
if ( $a <= 9 ) who
```

This script executes the **who** command.

>= Greater than or equal to

```
#
set a=1
if ( $a >= 1 ) who
```

This script executes the **who** command.

< Less than

```
#
set a=1
if ( $a < 1 ) who
```

This script executes the **who** command.

> Greater than

```
#
set a=1
if ( $a > 0 ) who
```

This script executes the **who** command.

8.4.3. Shift Operators:

<< Left Shift operators Shifting is a binary operation. Essentially, shifting a number left 1 is the same as mutilpying by 2. By the same token, shifting a number left 3 (**n << 3**) is the same as multiplying that number by 8 (2 to the 3rd).

```
#
@ x=2
@ y=( $x << 1 )
if ( $y == 4 ) who
```

This script executes the **who** command.

>> Right Shift operators Shifting right is dividing by two, shifting right 3 (**n >> 3**) is the same as dividing by 8 (2 to the 3rd.)

```
#
@ x=4
@ y=( $x >> 1 )
if ( $y == 2 ) who
```

This script executes the **who** command.

8.4.4. Math Operators:

\+ Addition

```
#
@ a=1+3
if ( $a == 4 ) who
```

This script executes the **who** command.

NOTE

Note the use of the at sign "@" in the context of a math operation. This assigns a decimal number to the variable, rather than assigning a string to the variable.

\- Subtraction

```
#
@ a=9-1
if ( $a == 8 ) who
```

This script executes the **who** command.

* Multiplication

```
#
@ a=2*4
if ( $a == 8 ) who
```

This script executes the **who** command.

/ Division

```
#
@ a=8/2
if ( $a == 4 ) who
```

This script executes the **who** command.

% Modulo

```
#
@ a=9%4
if ( $a == 1 ) who
```

This script executes the **who** command.

8.4.5. Other Operators:

! Negation

```
#
set a=1
if ( $a != 9 ) who
```

This script executes the **who** command.

( ) Parenthesis -- bracket expressions

```
#
set a=1
set b=9
if ( $a == 1 & $b == 9 ) who
```

This script executes the **who** command.

8.5. File Inquiry Operators

File inquiry operators are used to test the qualities of a given file. When used with an if statement, the expression is used with the following synopsis:

```
#
if ( -operator filename ) command
```

In the above example, if the file filename meets the conditions set by the operator, the following command is executed.

The exclamation point "! " is used to test if the condition is not met, as in the synopsis:

```
#
if ( ! -operator filename ) command
```

The following file inquiry operators are available.

Table 8-2 File Inquiry Operators

Character:	Meaning:
r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

If the file does not exist or is inaccessible, then all inquiries return "**Ø**" (the value of a false expression).

If more detailed status information is required, the command should be executed outside of an expression and the variable status examined (see Section 7.1 -- "Predefined C Shell Variables" The Status Variable).

An example of a file inquiry operator used in a shell script is:

```
#
if ( -e test ) echo "The test file is here"
```

If the file named "test" exists in the current working directory, this script will echo "The test file is here".

8.6. Options to the Csh Command

SYNOPSIS:

csh -option filename

-c Commands are read from the (single) following argument that must be present. Any remaining arguments are placed in argv.

-e The C shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

-f The C shell starts faster, because it neither searches for nor executes commands from the file ~/.cshrc in the invoker's home directory.

-i The C shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals. and outputs are terminals. Commands are parsed, but not executed. This aids in syntactic checking of C shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A \ can be used to escape the new line at the end of this line and continue onto another line.

-v Causes the verbose variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the echo variable to be set, so that commands are echoed immediately before execution.

-V Causes the verbose variable to be set even before .cshrc is executed.

-X Causes the echo variable to be set before .cshrc is executed. After processing of flag arguments, if arguments remain but neither the **-c, -i, -s,** or **-t** options are given, the first argument (argument Ø -- the name of the C shell script) is taken as the name of a file of commands to be executed. The C shell opens this file, and saves its name for possible re-substitution by $Ø. The C shell executes a standard (Bourne) shell if the first character of a script is not a pound sign "**#**". All the remaining arguments are placed in the variable argv.

Table 8-3 Options to the Csh Command

Option:	Meaning:
-c	single argument only
-e	exit on error
-f	faster (doesn't read ~/.cshrc)
-i	interactive -- prompts for input
-s	take commands from standard input
-t	read and execute single line of input
-v	verbose -- echo commands after history
-x	verbose -- echo commands before history
-V	verbose variable set before ~/.cshrc is read
-X	verbose set before ~/.cshrc and before history subs

8.7. Comment Lines in the Shell

In general, for the C shell, a word beginning with a pound sign "**#**" causes that word and all the following characters up to a new-line to be ignored.

As the first character, in the first column on the first line of a shell script, the pound sign means "use the C shell (/bin/csh) to execute this script." Other characters in the first position mean other things. Please refer to the following chart for the specific meanings:

Table 8-4 Shell Script Shell Indicators

First Line in Script:	Meaning:
#!sh	Use the "Bourne" shell -- /bin/sh
#!csh	Use the "C" shell -- /bin/csh
#!	Error, can't determine which shell
#!/xxx	Use the shell in the file /xxx
X	Any character other than "#" uses /bin/sh

SECTION 9
C SHELL FILES

9.1. Start-up Files

The letters "rc" append certain filenames. These initials stand for "read command". Thus the file ~/.cshrc is the "csh read command" file. Each time the C shell (csh) is invoked it reads and executes the contents of this file.

"Rc" files are read at different times by the shell (and other programs) to perform different tasks. The file named "/etc/rc_csh" is read each time the ZEUS operating system is booted (Refer to the System Administrator Manual for more details). Another "rc" files is "~/.exrc" which is read each time the **ex**(1) editor or **vi**(1) editor is called.

In addition to the "rc" files, there are a number of other files that hold importance for the C shell.

Table 9.1 shows the built-in files that are important to the system. Other files can be defined and used by each user as needed.

Table 9-1 Special Files

~/.cshrc
~/.login
~/.logout
~/.exrc
~/.profile
/bin/sh
/bin/csh
/dev/null
/etc/cshprofile
/etc/passwd
/etc/group
/tmp/sh*

There are no clear rules for the implementation of the initialization files (~/.cshrc, ~/.login, ~/.logout and ~/.exrc). The examples provided are merely suggestions, they have been presented to suggest a limited range of possibilities, some experienced users prefer much more elaborate files. It is a matter of taste.

NOTE

The convention " ~/.file " refers to the fact that the file must be in the user's home directory. Either the C Shell, or the operating system (ZEUS) looks in that specific directory for the particular file.

9.1.1. ~/.cshrc:

The "~/.cshrc" file is read at the execution of each C shell. At login, the "~/.cshrc" file is read before the "~/.login" file.

The ~/.cshrc file should contain commands that are needed for every shell.

```
# ~/.cshrc file
set ignoreeof
set history=15
set mail=(5 /usr/spool/mail/deck)
set prompt="`whoami` # > "
stty erase ^H kill ^X
umask 2
set U=~/Util/New.csh
alias h     history
alias bye   logout
```

Figure 9-1 A Sample ~/.cshrc File

 # .cshrc file
The ~/.cshrc file is read by the C shell, thus it must begin with a pound sign. The ".cshrc" label is ignored as a comment.

set ignoreeof
The ignoreeof variable makes the C shell ignore accidental control-Ds and thus, accidental "logout's"

ALSO SEE:

Section 7.1 -- Predefined C Shell Variables

set history=15
The history variable controls the size of the history list.

ALSO SEE:

Section 7.1 -- Predefined C Shell Variables.
Section 4 -- The History Function

set mail=(5 /usr/spool/mail/deck)

The mail variable sets the frequency of new mail checks (in this case, 5 seconds between checks). The second argument sets the location of the mailbox. The time variable can be left out, but if it is included, the expression must be enclosed in parentheses.

ALSO SEE:

Section 7.1 -- Predefined C Shell Variables.

set prompt="`whoami` # \! >"

The prompt variable sets the ZEUS "ready" signal to the user. In this case, the prompt consists of: the "`whoami`" command which, since it is enclosed in backquotes is replaced by login name of the user; a pound sign, the backslash- exclamation point is replaced by a number that increments by 1 with each command; and a "greater than sign" which is used here to show the end of the prompt -- in this context it has no special (metacharacter) meaning. Other features of the prompt could include special terminal screen attributes (reverse video, flashing, etc.)

ALSO SEE:

Section 7.1 -- Predefined C Shell Variables.

stty erase ^H kill ^X

stty(1) is a command external to the C Shell (it is not a built in command) which sets various terminal characteristics Here, the erase character is set to a control-H (the traditional backspace key), and the kill character is set to control-X.

ALSO SEE:

stty(1)

umask 2

The built-in command **umask** sets the file protection mode for newly created files. A umask value of 2 sets a file protection mode which permits read and write permission for the user, the group, and read permission for everyone else (rw-rw-r--).

ALSO SEE:

Section 5.2 -- General Purpose Commands from the Prompt.

set U=~/Util/New.csh
This is a user-defined variable. The command **cd $U** is the same as **cd ~/Util/New.csh.** $U is a shorthand notation.

ALSO SEE:

Section 7.3 -- User-defined Variables

alias h history

alias bye logout
If an alias is needed from within a sub-shell, the alias command should be placed in the ~/.cshrc file since each new shell begins with its own alias list (aliases that are set in the ~/.login file will not be exported to new shells).

ALSO SEE:

Section 5.3 -- Environmental Commands from Prompt.

9.1.2. ~/.login:

The ~/.login file is read by the shell at login just after the ~/.cshrc file. It is read once at login thus it should contain commands that either export across shells (see Section 10 for a discussion of the Environment Variables and exporting), or commands that are needed only once at the beginning of the session at the terminal (in this category might fall a reminder program or a **calendar**(1) program.)

```
# .login file
setenv EXINIT "set number wm=20 | version"
setenv HOME /z/deck
setenv PATH .:/usr/bin:/bin:~/bin:/usr/games
setenv SHELL /bin/csh
set prompt="% "
echo " "
cat ~/.reminder
echo " "
calendar
```

Figure 9-2 A Sample ~/.login File

 # .login file

The .login file is also a shell script, read by C shell, thus it must begin with a pound sign. The ".login" label is ignored as a comment.

setenv EXINIT "set number wm=20 | version"

The EXINIT variable takes the place of ~/.exrc file -- it is read by the **ex** command to pre-set **ex** options. In this case, the number and word margin options are set, and the **version** command is executed. The EXINIT variable is faster than the ~/.exrc file since it is exported with each new shell and need not be re-read each time the editor is called.

ALSO SEE:

Section 10.2 -- Environment Variables Explained.

setenv HOME /z/deck

The HOME variable is set to this user's home directory.

ALSO SEE:

Section 10.2 -- Environment Variables Explained.

setenv PATH .:/usr/bin:/bin:~/bin:/usr/games

The PATH variable is set to a number of directories with useful commands in them.

ALSO SEE:

Section 10.2 -- Environment Variables Explained.

setenv SHELL /bin/csh

The SHELL variable sets the login shell to be the C Shell.

ALSO SEE:

Section 10.2 -- Environment Variables Explained.

set prompt="% "

The first prompt is set to the default: the percent sign.

ALSO SEE:

Section 7 -- Shell Variables.

This command simply provides a space between any existing material on the screen (like the message of the day) and the material that follows.

cat ~/.reminder
This user has created a reminder file for upcoming events. This command reads that file at each login.

echo " "

This command provides another blank line.

calendar
This command executes the user's calendar.

ALSO SEE:

calendar(1)

9.2. Other Related C Shell Files

9.2.1. ~/.logout:

The ~/.logout file is read by login shell, at logout. It should contain any information the user needs just before leaving the terminal session.

```
# .logout file
who
echo " "
date
echo " "
cd; calendar
```

Figure 9-3 A Sample ~/.logout File

 # .logout file
The .logout file is a shell script, read by C shell, thus it must begin with a pound sign. The ".logout file" label is ignored as a comment.

who At logout, the **who** command informs the user of the other users left on the system.

echo " "

Blank line.

date The **date** command.

echo " "

Blank line.

cd; calendar
The **cd; calendar** command executes the **calendar** in this user's home directory for information on the next day's schedule.

9.2.2. ~/.exrc:

The ~/.exrc file is read when the **ex** or **vi** editors are called. The shell variable **EXINIT** performs the same function.

The ~/.exrc file is read by the **ex** or **vi** editors, not the C shell, thus no comment lines are available.

SEE ALSO:

Ex Reference Manual in the **Zeus Utilities Manual** for the available **ex** and **vi** options.

```
set number
set wm=20
set noredraw
set slowopen
set showmatch
version
```

Figure 9-4 A Sample ~/.exrc File

9.2.3. /bin/sh:

This file contains the Bourne shell, for shell scripts not starting with a pound sign "#"

ALSO SEE:

The Bourne Shell in the Zeus Utilities Manual

9.2.4. /bin/csh: This file contains the C shell, for shell scripts starting with a pound sign "#"

ALSO SEE:

csh(1).

ALSO SEE:

csh(1).

9.2.5. /dev/null:

This system file is the source of empty files. Any output directed to this file is lost.

9.2.6. /etc/cshprofile:

The /etc/cshprofile file is like the ~/.cshrc file, except it is read at the system level, before the ~/.cshrc file is read. It contains parameters for each C Shell operating environment. It is read by the login shell, before ~/.cshrc file.

ALSO SEE:

cshrc(5).

9.2.7. /etc/passwd:

This system file is the source of home directories and other basic login information.

ALSO SEE:

passwd(5)
Zeus Administrator Manual

9.2.8. /tmp/sh*:

Temporary file for " << " input.

In programs that take input from the body of a shell script with the double 'less than' signs, the shell makes a copy of the input and places it in a new file named /tmp/shNNNN, where NNNN is some number assigned by the shell and used to distinguish one /tmp/sh file from any other.

The input for the shell script is then read from this temporary file in /tmp.

ALSO SEE:

Section 2.9.2 -- Input Within a Script

SECTION 10
THE ENVIRONMENT

The Environment is a list of variables that are available to all the programs executed by the shell which created the environment variables.

Every time a shell is created (forked), it reads in the variables set in the environment. Each shell, thus inherits these environment variables and their values.

Environment variables can be considered "global" variables, while C shell variables can be considered "local" variables. Like C shell variables, there are pre-defined environment variables and user-defined environment variables.

Because environment variables are inherited they need be set only once at login in the ~/.login file which is read at the beginning of each login session. These variables are exported to all subsequent shells, they are available to all subsequent programs without the need to re-set them for each program.

Environment variables are established with the following syntax:

setenv NAME value

This command can be given at the prompt, or written into one of the "start-up" files (the ~/.login file is recommended since the command needs to be read only once). Naming environment variables with all capital letters is merely a useful device to tell the two kinds of variables apart. Environment variables can be named with any string of characters.

10.1. Environment Variables

Environment variables are useful where a variable must be used across a number of different shells.

The table below shows the predefined environment variables and their meaning:

Table 10-1 Environment Variables

EXINIT	Ex editor initialization variables
HOME	Home directory
LOGNAME	Login name
PATH	Search path for commands
SHELL	Shell being used
TERM	Type of terminal
TERMCAP	File from which the TERM is read
TZ	Timezone

10.2. Environment Variables Explained

In the ZEUS operating system, environment variables are read by each new C shell and given corresponding values with a corresponding named transliterated into lower case. These new "name/value" pairs become new C shell variables available locally to the new shell.

10.2.1. EXINIT:

SYNOPSIS:

setenv EXINIT options

EXINIT stands for "ex initialization"; the EXINIT variable initializes the **ex** editor options.

One example of a command to set the EXINIT variable is:

setenv EXINIT "set number wm=20 showmatch | version"

which sets the editors line number function, sets the "wrap-margin" function to 20 spaces from the right margin, sets the "showmatch" options (which highlights matching brackets, parenthesis, and braces), and prints the "version" of the editor each time **ex** or its visual counterpart **vi** are called.

Note that multiple commands are set in quotes, and that the set routine is piped through the **version** command.

The EXINIT variable performs the same function as its predecessor, the ~/.exrc file, however, it is faster since the EXINIT variable is automatically a part of the **ex** environment, while the ~/.exrc file must be read each time the editor is called.

DEFAULT:

unset

SEE ALSO:

The EX Reference Manual in the ZEUS Utilities Manual

10.2.2. HOME:

SYNOPSIS:

setenv HOME /path/home.directory

The HOME variable serves the same function as the home variable in the C shell. It established the location for the **cd** command, and the file name for the tilde "~" when it is used as a metacharacter.

If the home variable is not set (either in the ~/.cshrc file, the ~/.login file, or at the prompt) the home variable takes its value from the HOME variable. That is, the value of HOME is exported to each new C shell as it is created (forked).

Regardless of the shell invoked, each new process inherits the values of all set environment variables. Both shells read (and inherit) the values set in the environment.

DEFAULT:

HOME /path/users.home.directory

Unless otherwise set, the HOME variables takes its value from the home directory field of the /etc/passwd file.

10.2.3. LOGNAME:

SYNOPSIS:

setenv LOGNAME name

The LOGNAME variable holds the user's login name.

10.2.4. PATH:

SYNOPSIS:

setenv PATH /path/directory:/path/directory

The PATH variable serves the same function as the path variable for the C shell.

EXAMPLE:

setenv PATH .:/usr/bin:/bin:~/bin:/etc:/usr/games

DEFAULT:

PATH .:/usr/bin:/bin

10.2.5. SHELL:

SYNOPSIS:

setenv SHELL /path/shell.program

The SHELL variable serves the same function as the shell variable for the C shell.

EXAMPLE:

setenv SHELL /bin/csh

DEFAULT:

SHELL /bin/csh

Unless otherwise set, the SHELL variables takes its value from the shell field of the /etc/passwd file.

10.2.6. TERM:

SYNOPSIS:

setenv TERM terminal.type

The TERM variable serves the same function as the term variable for the C shell.

EXAMPLE:

setenv TERM vz

DEFAULT:

TERM vz

Unless otherwise set, the TERM variables takes its value from the /etc/ttytype file.

10.2.7. TERMCAP:

SYNOPSIS:

setenv TERMCAP /path/directory

The TERMCAP variable holds the name of the file used to establish the TERM commands.

EXAMPLE:

setenv TERMCAP ~/bin/new.termcap

DEFAULT:

unset

Although the TERMCAP variable is unset by default, the TERM value is taken from the file /etc/termcap.

10.2.8. TZ:

SYNOPSIS:

setenv TZ timezone

The TZ variable holds the timezone of the machine, in hours, measured from Greenwich mean time.

EXAMPLE:

setenv TZ PST8PDT

DEFAULT:

set at each site in the /etc/rc csh file

APPENDIX A
GLOSSARY

Important terms presented in this document are listed in this Appendix. References of the form (2.5) or (Section 2) indicate that more information is available in Section 2.5 or Section 2 of this document. References of the form **pr**(1) indicate that the command **pr** is documented in Section 1 of the ZEUS Reference Manual.

Dot .

The current directory is a file that has the name " . " (referred to as Dot) as well as the name printed by the command **pwd**. The current directory is usually the first component of the search path contained in the variable path. Thus, commands that are in . are found first (7.1.11). The period character is also used to separate components of file names (3.2).

The dot character . at the beginning of a component of a path name is treated specially and is not matched by the file name expansion metacharacters question mark " **?** ", asterisk " ***** ", and the left and right brackets " **[** " and " **]** " pairs (3.1).

Dot-Dot ..

Each directory has a file " .. " referred to as dot-dot, which is a reference to the directory immediately above in the file system hierarchy. This directory is known as the parent directory. After changing directories with **cd**, for example,

cd paper

it is possible to return to the parent directory by entering

cd ..

The current directory is printed by **pwd** (3.2).

A

alias
An **alias** specifies a shorter or different name for a ZEUS command, or a transformation on a command to be performed in the shell. The shell command **alias** establishes aliases and can print their current values. The command **unalias** is used to remove aliases (5.3.1).

argument
Commands in ZEUS receive a list of argument words. Thus, the command

echo a b c

consists of a command name **echo** and three argument words a, b, and c. (5.2.2).

argv The list of arguments to a command written in a shell script or shell procedure is stored in a variable called argv within the shell. This name is taken from the conventional name in the C programming language (The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie Zilog part # Ø3-3161).

B

background
A background command is a command that runs while the shell executes other commands. It is also known as a "detached" command, because it is "detached" from its dependence upon the terminal. (2.5).

bin A directory containing binaries of programs and shell scripts to be executed is typically called a bin directory. The standard system bin directories are /bin, which contains the most heavily used commands, and /usr/bin, which contains most of the other user programs. Binaries can be placed in any directory. The

name of the directories should be a component of the variable

break
exit from loops within the control structure of the shell (6.5.1).

built-in
A command executed directly by the shell is called a built-in command. Most commands in ZEUS are not built into the shell, but exist as files in /bin directories. These commands are accessible because the directories in which they reside are named in the path variable.

C

case A **case** command is used as a label in a **switch** statement in the shell's control structure, similar to that of the language C (6.4).

cat The **cat** program prints out specified files on the standard output (the terminal). It is usually used to look at the contents of a single file on the terminal (**cat**(1)).

cd The **cd** command changes the working directory. With no arguments, **cd** changes the user's working directory to be the user's home directory (7.1.5).

cmp It is usually used on binary files, or to see if two files are identical (**cmp**(1)). For comparing text files, use the program **diff**, described in **diff**(1).

command
A function performed by the system, either by the shell or by a program residing in a file in the ZEUS system, is called a command.

command substitution
The replacement of a command enclosed in back quote (`) characters by the text output by that command is referred to as **command substitution.**

component
A part of a path name between slash (/) characters is

called a component of that path name. A variable that has multiple strings as its value is said to have several components; each string is a component of the variable.

continue
A built-in command that causes execution of the enclosing **foreach** or **while** loop to cycle prematurely. Similar to the **continue** command in the C programming language (6.5.2).

core dump
When a program terminates abnormally, the system places an image of its current state in a file named core. The core dump can be examined with the system debuggers **adb(1)** to determine what went wrong with the program. If, for a system program, the shell produces a message of the form:

command: Segmentation violation -- Core dumped

(where "Segmentation violation" is only one of several possible messages).

cp The copy (**cp**(1)) program copies the contents of one file into another file.

.cshrc
The file .cshrc in the home directory is read by each shell as it begins execution. It is usually used to change the setting of the variable path and to set effect globally (9.1.1).

D

date The **date**(1) command prints the current date and time.

debugging
Debugging is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables that can be used to aid in shell debugging.

default
The label default: is used within shell **switch**

statements to label the code to be executed if none of the **case** labels matches the value switched (6.4).

DELETE
The DELETE or RUBOUT key on the terminal is used to generate a ZEUS interrupt signal that stops the execution of most programs (6.6.3).

detached
A command that runs while the shell is executing other commands is referred to as detached or "running in background" (2.5).

diagnostic
An error message produced by a program is often referred to as a **diagnostic.** Most error messages are not written to the standard output (the terminal), since the output of that is often directed away from the terminal (2.9.3). Instead, error messages are written to the diagnostic output, which usually appears on the terminal (2.9.5).

directory
A structure that contains files is called a directory. The home directory is the directory in which the user is placed upon loggin in (7.1.5).

E

echo The **echo** command prints strings (arguments). (5.2.2).

else The **else** command is part of the "if-then-else-endif" control command construct (6.3).

EOF An end-of-file is generated whenever a command reads to the end of a file that it has been given as input. It can also be generated at the terminal with a control-d. Commands receiving input from a pipe receive an EOF when the command sending them input completes. Most commands terminate when they receive an EOF. The shell has an option to ignore EOF from a terminal input, which makes it possible to avoid logging out accidentally by typing too many control-d's (7.1.6).

escape
A backward slash (\) character used to prevent the special meaning of a metacharacter is said to escape the character from its special meaning. Thus,

echo *

echoes the character *, while

echo *

echoes the names of the file in the current directory. In this example, \ escapes * (3.3).

/etc/passwd
This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by colons ":" (9.1.9). This file can be examined by entering

cat /etc/passwd

The command **grep** is often used to search for information in the file. See **passwd**(5) and **grep**(1) for more details.

exit The **exit** command, which is built into the shell, is used to force termination of a shell script (5.3.2).

exit status
A command that uncovers a problem can reflect this problem back to the command that invoked it by returning a nonzero number as its considered normal termination. The **exit** command can be used to force a shell command script to give a nonzero exit status (7.1.14).

expansion
Replacing shell input strings that contain metacharacters with other strings is referred to as the process of expansion. For example, replacing the word * with a sorted list of files in the current directory is a file name expansion. Replacing the characters !! with the text of the last command is a history expansion. Expansions are also referred to as substitutions (Section 3).

expressions
Expressions are used in csh to control the conditional structures used in writing shell scripts and in calculating values for these scripts. The operators available in csh expressions are those of the C language

(Section 6).

extension
File names often consist of a root name and an extension, separated by the period character (.). By convention, groups of related files often share the same root name. Extensions are added to differentiate among files within the group. Thus, if prog.c is a C program, the object file for this program would be stored in prog.o. Similarly, a paper written with the -ms nroff macro package might be stored in paper.ms, while a formatted version of this paper might be kept in paper.out and a list of spelling errors in paper.errs.

F

file.name
Each file in ZEUS has a name consisting of up to 14 characters, not including the slash character (/), which is used in path name building. Most file names do not begin with the period character. They contain only letters and digits, with perhaps a period separating the root portion of the file name from an extension.

file name expansion
File name expansion uses the metacharacters *, ?, [,], {, and } to provide a convenient mechanism for naming files. Using file name expansion makes it easy to name all the files in the current directory, or all files that have a common root name. Other file name expansion mechanisms use the metacharacter ~ and allow files in other users' directories to be named easily (3.1).

flag Many ZEUS commands accept arguments that are not the names of files or other users, but are used to modify the action of the commands. These are referred to as flag options and, by convention, consist of one or more letters preceded by the hyphen (-) character (2.2). For example, the **ls** list file command has an option to list the sizes of files. This is specified

```
ls -l
```

foreach
The **foreach** command is used in csh scripts and at the terminal to specify repetition of a sequence of commands while the value of a given csh variable falls within a specified range (6.1).

G

getty
The **getty** program determines the speed at which the terminal is to run when the user first logs in. It displays the initial system banner and **login.**

goto The csh command **goto** is used in csh scripts to transfer control to a given label (6.5.3).

grep The **grep** command searches through a list of argument files for a specified string. For example,

```
grep roberta /etc/passwd
```

prints each line in the file /etc/passwd that contains the string **roberta.** Actually, **grep** scans for regular expressions in the sense of the editors **ed**(1) and **ex**(1). Grep stands for "globally find regular expression and print".

H

hangup
When a user hangs up a phone line, a hangup signal is sent to all running processes on the user's terminal, causing them to terminate execution prematurely. To allow commands to continue running after logging off a dialup, use the command **nohup** (6.6.2).

head The **head** command prints the first few lines of one or more files. Run the **head** program with a group of file names as arguments to get a general idea of the

contents of the files (**head**(1)).

history
The **history** function of csh allows previous commands to be repeated. Csh has a history list where these commands are kept, and a history variable that controls how long the list is. (Section 4).

home directory
Each user has a home directory, that is given in the password file. The user is placed in the home directory when first logging in. The **cd** command with no arguments returns the user to this directory. The name of this directory is recorded in the shell variable home.

I

if The **if** command is a conditional command used in csh command scripts to determine what course of action to take next (6.3).

ignoreeof
Normally, the user's shell exits, printing logout if the user types a control-d at a % prompt. This is the usual way to log off the system. The user can **set** the ignoreeof variable in the **ignoreeof** in the .login file, and then use **logout** to log out. This is useful to avoid accidentally logging off by typing too many control-d characters. (7.1.6).

input
Information taken from the terminal or from files is called input. Commands normally read input from their standard input which is, by default, the terminal. The metacharacter "less than" "<" followed by a file name can be used to cause input to be read from a file. Many commands also read from a file specified as an argument. Commands placed in pipelines are read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if its input is not redirected and if a file name is not given to use as standard input. Special mechanisms exist for supplying input to commands in csh scripts (2.9.1).

interrupt
An interrupt is a signal that causes most programs to stop execution. It is generated by pressing the RUB or DEL key. Certain programs such as csh and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While csh is executing another command and waiting for it to finish, csh does not respond to interrupts. (2.5).

K

kill The **kill** program terminates processes (**kill**(1)).

L

.login
The file .login in the user's home directory is read by csh each time the user logs in to ZEUS; the commands there are executed (9.1.2).

logout
The **logout** command causes a login shell to exit. Normally, a login shell exits when control-d is pressed, generating an EOF. If ignoreeof has been set in the .login file, control-d does not work, and it is necessary to use the command **logout** to log off the ZEUS system (5.3.3).

.logout
When a user logs off of ZEUS, the shell prints **logout** and executes commands from the file .logout in the user's home directory.

lpr The command **lpr** is the line printer command. The standard input of **lpr** is spooled and printed on the ZEUS line printer. It is possible to give **lpr** a list of file names as arguments to be printed. It is common to use **lpr** as the last component of a pipeline (**lpr**(1)).

ls The list file **(ls)** command is one of the most commonly used ZEUS commands. With no argument file names, it displays the names of the files in the current directory. It has a number of useful flag arguments. It can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (**ls**(1)).

M

mail The **mail** program is used to send and receive messages from other ZEUS users (**mail**(1)).

make The **make** command is used to maintain one or more related files and to organize functions to be performed on these files. Its primary use is maintaining a single program consisting of several source files. In many ways, **make** is easier to use, and more helpful, than shell command scripts (**make**(1)).

makefile
The file containing the commands for **make** is called **makefile.**

metacharacter
Many characters that are neither letters nor digits have special meaning, either to the shell or to ZEUS. These characters are called metacharacters. It is necessary to enclose these characters in quotes if they are used in arguments to commands and no special meaning is required. An example of a metacharacter is the character >, which is used to indicate placement of output into a file. For the purposes of the history mechanism, most unquoted metacharacters form separate words (Section 3). Appendix A of this document lists the metacharacters.

mkdir
The **mkdir** command is used to create a new directory (**mkdir**(1)).

modifier
A modifier is a part of a command line that changes the way the original command is interpreted. Substitutions, with the history mechanism (keyed by the

character !), or of variables using the metacharacter $, are often subjected to modifications, which are indicated by placing the character : after the substitution and following this with the modifier itself (Section 4).

N

noclobber

The csh variable noclobber can be set in the file > output redirection metasyntax of the shell (2.9.6 and 7.1.8).

nohup

The shell **nohup** command is used to run background commands to completion even if the user logs off before these commands complete (6.6.2).

nroff

The standard text formatter on ZEUS is the program **nroff.** Using **nroff** and one of the available macro packages for it, it is possible to have documents automatically formatted and prepared for phototypesetting using the typesetter program **troff** (**nroff**(1)).

O

onintr

The **onintr** command is built into the C Shell and is used to control the action of a shell command script when an interrupt signal is received (6.6.3).

output

Many commands in ZEUS produce data that is called output. This output is usually placed on what is known as the standard output, which is normally connected to the user's terminal. The shell has a syntax using the metacharacter > for redirecting the standard output of a command to a file (2.9). Using the pipe mechanism

and the metacharacter |, it is also possible for the standard output of one command to become the standard input of another command (2.10). Some commands do not direct their output to the standard output. The line printer command (**lpr**), for example, diverts its output to the line printer. The **write** command places its output on another user's terminal (**write**(1)). Commands also have a diagnostic output where they write their error messages. Normally, these go to the terminal even if the standard output has been sent to a file or another command. However, it is possible to direct error diagnostics along with standard output using a special metanotation (2.9.5).

P

path The csh variable path gives the names of the directories in which it searches for the commands it is given. It always checks first to see if the named command is built into the shell. If it is, it does not need to search for the command, as it can perform it internally. If the command is not built in, csh searches for a file with the name given in each of the directories in the path variable, left to right. Since the normal definition of the path variable is

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

Csh normally looks in the current directory, and then in the standard system directories, /bin and /usr/bin, for the named command (7.1.11 and 10.3.3). If the command cannot be found, csh prints an error diagnostic. Scripts of C shell commands are executed using another shell to interpret them if they have execute bits set. This is normally true because a command of the form

chmod 755 script

is executed to turn on these execute bits (**chmod**(1)).

path name
A list of names, separated by slash (/) characters forms a path name. Each component between successive "slant" (/) characters names a directory in which the next component file resides. Path names that begin

with the character / are interpreted relative to the root directory in the file system. Other path names are interpreted relative to the current directory as reported by **pwd**. The last component of a path name can name a directory; however, it usually names a file.

pipeline
A group of commands that are connected together with the standard output of each connected to the standard input of the next is called a pipeline. The pipe mechanism used to connect these commands is indicated by the vertical bar (|) metacharacter (2.10).

pr The **pr** command prepares listings of the contents of files with headers that give the name of the file and the date and time at which the file was last modified (**pr**(1)).

printenv
The **printenv** command is used on ZEUS systems to print the current setting of variables in the environment.

process
An instance of a running program is called a process (**ps**(1)). The numbers used by **kill** and printed by **wait** are unique numbers generated for these processes by ZEUS. They are useful in **kill** commands, which can be used to stop background processes (**kill**(1)).

program
A program (usually synonymous with command) is a binary file or csh command script that performs a useful function.

prompt
Many programs print a prompt on the terminal when they expect input. For example, the editor **ex(1)** prints a colon (:) when it expects input. The shell prompts for input with a percent sign (%), and occasionally with a question mark (?), when reading commands from the terminal (7.1.12). The csh variable prompt can be set to a different value to change the shell's main prompt. This is primarily used when debugging the shell (7.1.12).

ps The **ps** command shows the processes a user is currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (**ps**(1)). Login shells

(such as the csh obtained when logging in) are shown as -.

pwd The **pwd** command prints the full path name of the current working directory.

Q

quit The quit signal, generated by a control-\, terminates programs that are behaving abnormally. It normally produces a core image file (**core**(5)).

quotation
The process that prevents metacharacters from being interpreted with special meaning, usually by using the single quote (') character in pairs or by using the backslash (\) character, is referred to as quotation.

R

redirection
The routing of input or output from or to a file is known as redirection of input or output (2.9).

repeat
The **repeat** command iterates another command a specified number of times (5.2.6).

RUB The RUB or DEL key generates an interrupt signal that is used to stop programs or to cause them to return and prompt for more input (6.6.3).

S

script
Sequences of csh commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program by using the shell to selectively run other programs (Section 6).

set The built-in **set** command assigns new values to shell variables and displays the values of the current variables. Many csh variables have special meaning to csh itself (5.3.4).

setenv
On ZEUS systems, variables in the environment environ(5) can be changed by using the **setenv** built-in command (Section 10). The **printenv** command can be used to print the value of the variables in the environment.

shell
A shell is a command language interpreter. It is possible for users to write and run their own shells, as shells are no different from any other programs in terms of system response. This document deals with the details of one particular shell, called **csh.**

shell script
See **script** (Sections 5 and 6).

sort The **sort** program sorts a sequence of lines in ways that can be controlled by argument flags (**sort**(1)).

source
The **source** command causes csh to read commands from a specified file. It is useful for reading files such as .cshrc after changing them (5.3.6).

special character
See metacharacters and Appendix A of this document.

standard
The standard input and standard output of commands are often referred to. See input and output (2.9).

status
A command normally returns a status when it finishes. By convention, a status of zero indicates that the command succeeded. Commands can return nonzero status to indicate that some abnormal event has occurred. The csh variable status is set to the status returned by the last command. It is most useful in shell scripts (7.1.14).

substitution
Csh implements several substitutions where sequences indicated by metacharacters are replaced by other sequences. Examples of this are history substitution keyed by the metacharacter !, and variable substitution indicated by $. Substitutions are also referred to as expansions (3.1).

switch
The **switch** command of csh allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the **switch** statement in the C language (6.4).

T

termination
When a command being executed finishes, it is said to terminate. Commands normally terminate when they read an EOF from their standard input. It is also possible to terminate commands by sending them an interrupt or quit signal. The **kill** program terminates commands specified by their process numbers.

then The **then** command is part of csh's if-then-else-endif control construct used in command scripts (6.3)

time The **time** command measures the amount of CPU and real time consumed by a specified command (5.2.8).

troff
The **troff** program is used to typeset documents. See also **nroff** (**troff**(1)).

U

unalias
The **unalias** command removes aliases (5.3.7).

unset
The **unset** command removes the definitions of csh variables (5.3.8).

V

variable expansion
See variables and expansion (Section 7).

variables
Variables in csh hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See path, noclobber, and ignoreeof for examples. Variables such as argv are also used in writing csh command scripts (7.1.1).

verbose
The verbose csh variable causes commands to be echoed after they are history expanded. This is often useful in debugging csh scripts. The verbose variable is set by the shell's command line option (7.1.17).

W

wait The built-in command **wait** causes csh to pause, and not prompt, until all commands run in the background have terminated (5.2.10).

while
The **while** built-in control construct is used in csh command scripts (6.2).

word A group of characters that forms an argument to a command is called a word. Many characters that are neither letters, digits, -, ., or / form words by themselves, even if they are not surrounded by blanks. Any sequence of characters can be made into a word by surrounding it with single quote (') characters, except for the single quote character itself and !, which require special treatment.

working directory
Any directory a user is currently working in is called a working directory. This directory name is printed by the **pwd** command, and the files listed by **ls** are the ones in this directory. The user can change working directories using the **cd** command. (3.2 - Dot)

write
The **write** command is used to communicate with other users who are logged in to ZEUS (**write**(1)).

Z

ZEUS ZEUS provides facilities that allow csh to invoke other programs, such as editors and text formatters.

APPENDIX B
C SHELL ERROR MESSAGES

B.1. Error Messages Explained

The following is an annotated partial list of error messages are produced by the C Shell as a response to various input errors.

The full list of error messages follows in the next section.

<< terminator not found

In the context of a C shell script, this error indicates that the label used to indicate the end of input is not a part of the script. The following example would cause such an error:

```
# test
ex test << EOF
g/^$/d
w
q
```

The solution is to put the terminator EOF at the end of the script. See section 2.9 Input/Output Control.

Alias loop

If an alias is established that calls itself, an **alias loop** is created. The following two commands create an alias loop:

```
alias ls list
alias list ls
```

Both aliases can be established, but an attempt to execute either will result in the error message. This error is resolved by **unaliasing** the alias that has created the error message with the command:

unalias alias.name

See Section 5.3 Environmental Commands from the Prompt for details on the **alias** command.

Ambiguous

This error is created when a filename metacharacter -- **"*", "~", "?"** is used in such a manner that refers to a number of files or directories in a situation that

requires a single file or directory, as in the command:

```
cd *
```

The error message:

```
*: Ambiguous.
```

results. The solution is to replace the metacharacter with a more specific file or directory name.

Arguments too long

This error is usually associated with metacharacter expansion. It can result from the following command:

```
echo /*/*/*
```

The solution is to provide a more specific argument.

Cannot determine type of shell to use

This error results from a symbol in the first column of the first line of a shell script that does not indicate which shell is to be used in executing the script. The following script will cause the error:

```
#!
who
```

Specifying a shell with a legitimate character will resolve the error. See Section 8.6 "Comment lines in the Shell".

Can't from terminal

Some commands cannot be executed from a terminal. For example, the built-in command **onintr** will produce the error message:

```
onintr: Can't from terminal
```

if it is attempted from a terminal. Commands that produce this error are intended for use within the body of a shell script. See Section 6 The C Shell Programming Language Structure.

Can't make pipe

The space provided for temporary files used in pipes in the root "/". If it fills up, there is no space left for the files needed by the pipe mechanism.

The solution is to clear out space in the root.

Command not found
If the shell can't locate the command, or if the command name has been mistyped, this error results. The error will also result if the command is not installed in the shell's hash table of commands. See Section 5.2.6 **rehash**.

Divide by Ø
This error results from a math operation within a shell script involving division by Ø.

end not found
Both the **foreach** and **while** shell script loops require a closing **end** statement.

endif not found
The **if, else** structure requires an **endif** statement. See Section 6.3

endsw not found
The **switch** structure requires an **endsw** statment. See Section 6.4

Expression syntax
Various syntactic errors can produces this error message. The following **if** statement:

```
if ( a > b ) echo HI
```

produces the error:

```
if: Expression syntax
```

because the alpha characters "a" and "b" cannot be compared with the math operator greater-than " > "

Improper mask
The "mask" refers to the umask file protection mode code.

Improper then
Refers to the then statment in an if-then context.

Interrupted
Indicates a program interupt. If a DELETE is hit in the beginning of an **ex** command, this error will result.

Invalid variable
An error occurs in calling or assigning variables. The solution is to call or assign the variable correctly.

label not found
In the context of a shell script with a goto label construct, the label must appear in the script. It is an error for the label to be missing.

Missing)
In the context of a

foreach (list)
or
while (list)

statements, the list must be enclosed within two parenthesis. It is an error for one parenthesis to be missing.

Missing]
In the context of a

command [range]

command, the range must be enclosed within two brackets. It is an error for one brackets to be missing.

Missing }
In the context of a

command { list }

command, the list must be enclosed within two braces. It is an error for one braces to be missing.

Missing file name

Mod by Ø
In a math operation involving the modulo function " fB% ", the right hand side of the equation cannot be zero.

No file for $Ø
Argument zero is the name of the file being executed. In the file test with the following lines:

```
# test
echo $Ø
```

execution with the command:

```
csh test
```

(or changing the execution bits with the **chmod** command

and executing it by name) results in the response:

test

the same command from the prompt:

echo $Ø

results in the message.

No home
Any command that depends upon the $HOME variable (e.g. the **cd** command) will produce an error message if the $HOME variable is not set.

No match
When using filename expansion metacharacters characters: (" * ", " [", "] ", " { ", " } ", " ? ") it is an error for no filename to match (unless the nonomatch C Shell variable is set).

No more processes
Only a limited number of background jobs can be run by any single "parent" (login) process. An attempt to initiate additional jobs in background (detached) result in this error.

No more words
An attempt to address words past the end of the list in a word list, perhaps in a foreach, while, or case statement.

non-ascii shell script
An attempt to execute a file as a shell script if it is comprised of non-ascii characters.

Not in while/foreach
In shell scripts, this error may result from an attempt to adress an argument out of the while or foreach loop.

Not login shell
An attempt to **logout** from a subshell (any shell other than the login shell) produces this error.

The solution is to **exit** out of any subshell and then issue the **logout** command.

Out of memory
The C shell can run out of memory.

Output redirection not allowed
Commands that do not provide for output redirection (such as the **source** command) produce an error in the following format:

source: Output redirection not allowed.

Pathname too long
If a pathname is too long, this message results. The solution is to change directory to a closer directory and access the file(s) from there.

Subscript error
An attempt to subscript a variable with an illegal subscript value.

Subscript out of range
In the script:

```
# test
echo $argv[1]
echo $argv[2]
echo $argv[3]
echo $argv[4]
```

given with the command:

test a b c

the statement

echo $argv[4]

will produce the error:

subscript out of range

because there are only 3 arguments.

Syntax error

then/endif not found

Too dangerous to alias that
An attempt to alias the word **alias** with the command:

alias alias a

results in this error. If the word **alias** needs to be aliased, the problem can be avoided with the command:

alias a alias

accomplishing the same results.

Too few arguments
Some commands require a specific number of arguments.

Too many arguments
Some commands require a specific number of arguments

Too many)'s
The statement:

foreach i (a b c))

produces the error.

Undefined variable
An attempt to use an undefined variable produces this error.

Unmatched `
Commands with an unclosed backquote, as in:

echo `date

produce this error.

Unmatched %c
This is a catch-all error, it refers to any command which requires two parts of the statment. The error results if the second part is missing.

Variable syntax
A syntax error.

Word too long
Some times the C shell cannot cope with a word that contains too many characters.

Words not ()'ed
Words in a list, not enclosed with the necessary parenthesis, as in the command:

foreach i a b c d

produces the error:

foreach: Words not ()'ed

B.2. The List

```
 -- Core dumped
%s: File exists
%s: non-ascii shell script
: Event not found
<< terminator not found
Alarm clock
Alias loop
Ambiguous
Ambiguous input redirect
Ambiguous output redirect
Arg list too long
Argument too large
Arguments too long
Bad ! arg selector
Bad ! form
Bad ! modifier:
Bad : mod in $
Bad address
Bad file number
Bad substitute
Bad system call
Badly formed number
Badly placed (
Badly placed ()'s
Block device required
Broken pipe
Bus error
Can't << within ()'s
Can't exit, ignoreexit is set
Can't from terminal
Can't make pipe
Cannot determine type of shell to use
Command not found
Cross-device link
Data transfer error
Device busy
Device write protected
Divide by Ø
EMT trap
End of data
End of media
Error Ø
Exec format error
Exit status %s
Expansion buf ovflo
Expression syntax
File exists
File table overflow
File too large
```

```
Floating exception
I/O error
IOT trap
Illegal instruction
Illegal seek
Improper mask
Improper then
Interrupted
Interrupted system call
Invalid argument
Invalid null command
Invalid variable
Is a directory
Killed
Line overflow
Missing )
Missing ]
Missing file name
Missing name for redirect
Missing }
Mod by Ø
Modifier failed
Mount device busy
New mail
No args on labels
No children
No file for $Ø
No home
No match
No media
No more processes
No more processes, waiting for current ones to complete.
No more words
No output
No prev lhs
No prev search
No prev sub
No space left on device
No such device
No such device or address
No such file or directory
No such process
Not a directory
Not a typewriter
Not enough core
Not in while/foreach
Not login shell
Not owner
Out of memory
Output redirection not allowed
Pathname too long
```

```
Permission denied
Quit
Read-only file system
Result too large
Rhs too long
Segmentation violation
Sig %d
Subscript error
Subscript out of range
Subst buf ovflo
Syntax error
Terminated
Text file busy
Too dangerous to alias that
Too few arguments
Too many ('s
Too many )'s
Too many arguments
Too many links
Too many open files
Too many words from ``
Trace/BPT trap
Undefined variable
Unknown error
Unknown user: %s
Unmatched
Unmatched %c
Unmatched `
Use "exit" to leave csh.
Use "logout" to logout.
Variable syntax
Word too long
Words not ()'ed
You have %smail.
end not found
endif not found
endsw not found
label not found
non-ascii shell script
source: Output redirection not allowed
then/endif not found
```

THE ZEUS LINE-ORIENTED TEXT EDITOR, ed*

* This information is based on articles originally written by Brian W. Kernighan, Bell Laboratories.

Preface

Although most text manipulation on the ZEUS Operating System is done with the screen-oriented editor, vi, some special circumstances warrent the use of the line editor, ed. This document is a tutorial guide to help beginners get started with ed and to introduce experienced users to its more complex options.

Sections 1-12 are oriented mostly for beginners. These sections cover basic commands or basic uses of more complex commands. When a subsection of a command is for experienced users, it is labeled as such. Beginners should be aware that more information is presented in these subsections than they need for basic tasks and that concepts are used in these explanations that have not yet been introduced in the regular text. Sections 13-23 offer experienced users more complex commands and describe ways that commands act on each other. Basic commands are summarized in the Appendix.

The recommended way for both beginners and experienced users to learn ed is to read this document, simultaneously using ed to follow the examples, then to read the description in Section 1 of the ZEUS Reference Manual. Experiment with ed. The only sure way of seeing how a command works is to try it. The exercises cover material not completely discussed in the text. A learn(1) script, %learn editor, is also available for ed.

The end-of-line character varies between terminals. This character is the RETURN key on most terminals, and is referred to in this text as RETURN.

This document is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that ed offers. Also, there is not enough space to explain basic ZEUS procedures; read ZEUS for Beginners to learn how to log in to ZEUS and what a file is.

Table of Contents

SECTION 1
GETTING STARTED

Ed is a line-oriented text editor--an interactive program for creating and modifying text on a line-by-line basis, using directions typed at a terminal. The text is often a document like this one, a program, or data for a program.

In ed terminology, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, or as the information to be edited.

Tell ed what to do to the text by typing instructions called "commands." Most commands consist of a single letter that must be typed in lowercase. Type each command on a separate line. Ed makes no response to most commands, it simply carries them out. Enter a RETURN after every ed command line.

The prompt character, either a $ or a %, appears after logging into the system. Invoke ed by typing

```
ed      (followed by a RETURN)
```

after the prompt. Ed is now waiting for commands.

When ed starts, it is like a blank piece of paper--there is no text or information present. Text must be supplied by typing it into ed, or by reading it into ed from a file.

The first command is append, written as the letter

```
a
```

by itself. It means "append (add) text lines to the buffer, as they are typed in." Appending is like writing fresh material on a piece of paper.

To enter lines of text into the buffer, type an a (followed by a RETURN), followed by the lines of text, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. If ed is not responding, it is probably because the . was omitted.

After the append command, the buffer contains the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The a and . are not there because they are not text.

To add more text, issue another a command and continue typing.

An error in the commands typed to ed results in the response

```
?
```

This is a cue to look for an error.

To save text for later use, write the contents of the buffer into a file. Use the write command

```
w
```

followed by the file name to be written on. This copies the buffer's contents into the specified file and destroys any previous information in the file. To save the text in a file named junk, for example, type

```
w junk
```

Leave a space between w and the file name. Ed responds by printing the number of characters it wrote out. In this case, ed responds with

```
68
```

Blanks and the return character at the end of each line are included in the character count.

Writing a file makes a copy of the text. The contents of the buffer are not disturbed, so lines can be added to it. This is an important point. Ed always works on the buffer copy of a file, not the file itself. No change in the contents of a file takes place until ed receives a w command. Writing out the text to a file from time to time as it is being created is a good idea. If the system crashes, only the text in the buffer is lost, but any text written in a file is safe.

To terminate a session with ed, save the text by writing it into a file, using the w command. Then type the command

```
q
```

which stands for quit. The shell responds with the prompt character $ or %. At this point, the buffer with all its text is no longer present. To protect the buffer from an accidental erasure, ed displays ? if it receives a quit command that was not preceded by a w command. At that point, either write the file or type another q to get out of ed.

Exercise 1

Enter ed and create some text using

```
a
... text ...
.
```

Write it out using w. Then leave ed with the q command, and print the file to see that everything worked. To print a file, type

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.

SECTION 2
READING TEXT FROM A FILE WITH 'e'

2.1. Basic Uses

The most common way to get text into the buffer is to read it from a file in the file system. This is done to edit text saved with the w command in a previous session. The edit command e fetches the entire contents of a file into the buffer.

If the three lines "Now is the time ..." have been saved with a w command, the ed command

```
e junk
```

fetches the entire contents of the file junk into the buffer, and responds

```
68
```

which is the number of characters in junk. Remember that if anything was already in the buffer, it is deleted first.

Using the e command to read a file into the buffer eliminates the need to use a file name after a subsequent w command; ed retains the last file name used in an e command, and w writes on this file. Thus, a good way to operate is with the following set of commands:

```
ed
e file
[editing session]
w
q
```

Simply enter w from time to time; the file name used at the beginning is updated with w.

To find out what file name ed is working on, type the file command f. In this example, an

```
f
```

prompts ed to reply

```
junk
```

2.2. Advanced Uses

The command

```
e  newfile
```

says "edit a new file called newfile without leaving the editor." The e command clears the buffer and reads in newfile. It is the same as the q command followed by a reentry of ed with a new file name, except that if ed retained a pattern, then a command like // still works.

Entering ed with the command

```
ed  file
```

has ed read file into the buffer and hold the name of the file. Any subsequent e, r, or w commands that do not contain a file name refer to this file. Thus, the commands

```
ed  file1
 ... (editing) ...
w       (writes back in file1)
e  file2        (edit new file, without leaving editor)
 ... (editing on file2) ...
w       (writes back on file2)
```

do a series of edits on various files without leaving ed; it is not necessary to type the name of any file more than once.

To change the name of the hold file, use f as follows:

```
ed  precious
f  junk
 ... (editing) ...
w
```

This reads the file precious into the buffer, then changes the name of the hold file junk. The w command applies the editing changes to the junk file, leaving the precious file untouched.

SECTION 3
READING TEXT FROM A FILE WITH 'r'

To read a file into the buffer without destroying anything that is already there, use the read command r. The command

```
r junk
```

reads the file junk into the buffer by adding it to the end of whatever is already in the buffer. Doing a read after an edit, that is, entering

```
e junk
r junk
```

puts a duplicate copy of the text after the current copy. The buffer now contains the following six lines:

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

The r command displays the number of characters read in after the reading operation is complete.

Exercise 2

Experiment with the e command. Try reading and printing various files. Ed may respond with ?name, where name is the name of a file. This means that the file does not exist, typically because the file name is spelled wrong, or reading the file is not allowed. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is equivalent to

```
ed
e filename
```

SECTION 4
PRINTING THE CONTENTS OF THE BUFFER

4.1. Print Command

Use the print command p to display the entire or partial contents of the buffer at the terminal.

4.2. Specific Lines

Specify the lines where printing is to begin and end, separated by a comma, and followed by the letter p. Thus, to print the first two lines of the buffer (that is, lines 1 through 2), enter

```
1,2p    (starting line=1, ending line=2 p)
```

Ed responds with

```
Now is the time
for all good men
```

To print all the lines in the buffer, ed provides a shorthand symbol for "line number of the last line in the buffer"--the dollar sign ($). Use the command:

```
1,$p
```

to print all the lines in the buffer, line 1 to last line. To stop the printing before it is finished, push the DEL (delete) key. Ed responds with

```
?
```

and waits for the next command.

To print the last line of the buffer, it would be possible to use

```
$,$p
```

However, ed lets this be abbreviated to $p. Any single line can be printed by typing the line number followed by a p. Thus,

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

It is possible to abbreviate even further by entering the line number without the letter p. So $ causes ed to print the last line of the buffer.

The $ can be used in combinations such as

```
$-1,$p
```

which prints the last two lines of the buffer.

Exercise 3

Create some text using the a command and experiment with the p command. Verify that line Ø or a line beyond the end of the buffer cannot be printed and that attempts to print a buffer in reverse order by typing

```
3,1p
```

also fail.

4.3. Current Line

Suppose the buffer contains the six lines as above, that the command 1,3p was issued, and that ed has printed the three lines. Typing

```
p       (no line numbers)
```

causes ed to print

```
to come to the aid of their party.
```

which is the third line of the buffer. It is also the last or most recent line that had actions performed on it. This p command can be repeated without line numbers, and ed continues to print line 3.

Ed maintains a record of the last line that had actions performed on it so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol dot (.).

Dot is a line number in the same way that $ is. It means "the current line" or "the line that most recently had action on it," and can be used in several ways. One possibility is to type

This prints all the lines from and including the current line through the end line of the buffer. In this example, these are lines 3 through 6.

Some commands change the value of dot, and others do not. The p command sets dot to the number of the last line printed; the last command sets dot to six.

Dot is most useful in combinations such as:

This means "print the next line" and is a handy way to step slowly through a buffer.

The command

means "print the line before the current line." This allows the line number to go backwards. Another useful command is

which prints the previous three lines.

Remember that all these commands change the value of dot. To find out what dot is at any time, type

Ed responds by printing the value of dot.

To summarize, p can be preceded by zero, one, or two line numbers. If there is no line number given, ed prints the current line; that is, the line that dot refers to. If there is one line number given with or without the letter p, it prints that line and sets dot there. If there are two line numbers, it prints all the lines in that range and sets dot to the last line printed. If two line numbers are specified, the first cannot be bigger than the second (Exercise 2).

Typing a single return prints the next line and is equivalent to Typing a - is equivalent to .-1p.

4.4. Advanced Commands

For the experienced user, the list command (l) gives slightly more information than p. In particular, l makes characters visible that are normally invisible, such as tabs and backspaces. With l, each tab appears as > and each backspace appears as <. This command makes it much easier to correct typing mistakes that insert extra spaces adjacent to tabs, or insert a backspace followed by a space.

The l command also provides for displaying long lines on short terminals. Any line that exceeds 72 characters is displayed on multiple lines, and each folded line, except the last, is terminated by a backslash.

Occasionally, the l command prints a string of numbers preceded by a backslash, such as \Ø7 or \16. These combinations make visible characters that normally do not print, such as form feed. Each such combination is a single character value of the nonprinting character in octal. Delete these characters unless they produce the desired result on the specific device used for ed output.

SECTION 5
DELETING LINES

Suppose the buffer contains two copies of junk as in Section 6. To get rid of the three extra lines in the buffer, use the delete command d The lines to be deleted are specified for d exactly as they are for p:

starting line, ending line d

Thus the command

4,$d

deletes line 4 through the end. There are now three lines left, which can be checked by entering 1,$p The $ now is line 3. Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise 4

Experiment with a, e, r, w, p, and d. Be sure to understand how dot, $, and line numbers are used.

Next, try using line numbers with a, r, and w as well. Verify that:

- a appends lines after the line number specified rather than after dot
- r reads a file in after the line number specified and not the end of the buffer
- w writes out exactly the lines specified, not the whole buffer

These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering

Ør filename

Lines can be inserted at the beginning of the buffer by entering

Øa

SECTION 6
MODIFYING TEXT

6.1. Substitute Command

One of the most important commands is the substitute command

```
s
```

which changes individual words or letters within a line or group of lines. It is used, for example, for correcting spelling mistakes and typing errors. This command has the most complexity of any ed command and can provide the greatest use.

6.2. Basic Modification

Suppose that line 1 reads

```
Now is th time
```

The e has been left off the. Use s to fix this as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters th the characters the." To verify that it works, type

```
p
```

and get

```
Now is the time
```

Dot must have been set to the line where the substitution took place, since the p command printed that line. Dot is always set this way with the s command.

The general way to use the substitute command is

```
starting-line, ending-line s/change this/to this/
```

A string of characters between the first pair of slashes is replaced by a string between the second pair in all the lines between starting-line and ending-line. Only the first occurrence on each line is changed. To change every occurrence, see Exercise 5. The rules for line numbers are

the same as those for p, except that dot is set to the last line changed. If no substitution took place, dot is not changed. This causes ? to appear as a warning.

Thus, enter

```
1,$s/speling/spelling/
```

to correct the first spelling mistake on each line in the text.

If no line numbers are given, the s command assumes "make the substitution on line dot," so it makes changes only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it.

It is also possible to type

```
s/something//
```

to change the first string of characters to nothing, that is, remove them. This is useful for deleting extra words in a line or for removing extra letters from words. For instance, in the line

```
Nowxx is the time
```

type

```
s/xx//p
```

to get

```
Now is the time
```

In ed, two adjacent slashes (//) mean no characters, not a blank.

Exercise 5

Experiment with the substitute command. Verify that the substitute command changes only the first occurrence of the first string. For example, enter:

```
a
the other side of the coin
.
s/the/on the/p
```

to get

    on the other side of the coin

To change all occurrences, add a g (for "global") to the s command, like this:

    s/ ... / ... /gp

Try other characters instead of slashes to delimit the two sets of characters in the s command. Any character except blanks or tabs will work. &.1p The following characters have special meanings:

    ^ . $ [ ] * \ &

Read Section 13 for an explanation of their use.

6.3. Advanced Modification

Either form of the s command can be followed by p or l to print or list the contents of the line. The commands

    s/this/that/p
    s/this/that/l
    s/this/that/gp
    s/this/that/gl

are all legal, and mean slightly different things. Also, ed does not recognize pg as being equivalent to gp.

Any s command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus, the command

    1,$s/mispell/misspell/

changes the first occurrence of mispell to misspell in every line of the file, but the command

    1,$s/mispell/misspell/g

changes every occurrence in every line.

Adding a p or l to the end of any of these substitute commands prints only the last line that was changed.

The undo command (u) "undoes" the last substitution: the last line that was substituted can be restored to its previous state by typing the command

SECTION 7
CONTEXT SEARCHING

Suppose the original three lines of text are in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

To find the line that contains their, use context searching. This specifies a line, regardless of what its number is, by specifying some of its contents.

Say "search for a line that contains this particular string of characters" by typing

```
/string of characters/
```

For example, the ed command

```
/their/
```

is a context search to find the next occurrence of the characters between slashes (their). It also sets dot to that line and prints the line for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that ed starts looking for the string at line .+1, searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search "wraps around" from $ to 1. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters cannot be found in any line, ed types the error message

```
?
```

To search for the desired line and substitute with one command, enter

```
/their/s/their/the/p
```

which yields

```
to come to the aid of the party.
```

There are three parts to that command: context search for

the desired line, make the substitution, and print the line.

Context searches are interchangeable with line numbers and can be used by themselves to find and print a desired line, or as line numbers for some other command, like s. They were used both ways in the previous examples.

With the buffer lines

```
Now is the time
for all good men
to come to the aid of their party.
```

the ed line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and all refer to the same line (line 2). To make a change in line 2, enter

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated by convenience. To print all three lines, enter

```
/Now/,/party/p
or
/Now/,/Now/+2p
```

or by any number of similar combinations. The first of these is better if the number of lines involved is unknown.

Ed also provides a shorthand for repeating a context search for the same string. For example, the ed line number

```
/string/
```

finds the next occurrence of string. If this is not the desired line, the search must be repeated. This can be done by typing

//

This shorthand stands for the most recently used context search expression. It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which finds the next occurrence of string1 and replaces it with string2.

Exercise 6

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Use context searches as line numbers for the substitute, print, and delete commands. Context searches are used less frequently with r, w, and a, but try them.

Try context searching using ?text? instead of /text/. This scans lines in the buffer in reverse order (end to beginning). This is useful when a desired string of characters is passed while going forward.

Again, the following characters have special meaning:

^ . $ [ ] * \ &

Read Section 13 for an explanation of their use.

SECTION 8
CHANGING AND INSERTING TEXT

This section discusses the change command and the insert command. Both of these commands operate on a group of one or more lines.

The change command is written as

```
c
```

and replaces a number of lines with different lines that are typed in at the terminal. For example, to change lines .+1 through $ to something else, type

```
.+1,$c
... type the lines of text here ...
.
```

The lines typed between the c command and the . take the place of the original lines between start line and end line. This is useful for replacing a line or several lines that have errors in them. It is possible to replace a single line with several lines.

If only one line is specified in the c command, just that line is replaced. The dot ends the input and works like the dot in the append command; it must appear by itself on a new line. If no line number is given, line dot is replaced and the value of dot is set to the last line typed in.

Insert (i) is similar to append. For instance

```
/string/i
... type the lines to be inserted here ...
.
```

inserts the given text before the next line that contains the string. The text between i and dot is inserted before the specified line. If no line number is specified, dot is used and dot is set to the last line inserted.

Exercise 7

The change command is rather like the combination delete followed by insert. Experiment to verify that

```
start, end d
i
... text ...
.
```

is like

```
start, end c
... text ...
.
```

These are not precisely the same if line $ gets deleted. Check this. What is dot?

Experiment with a and i to see that they are similar, but not the same. For instance,

```
line-number a
... text ...
.
```

appends after the given line, while

```
line-number i
... text ...
.
```

inserts before it. If no line number is given, i inserts before line dot, but a appends after line dot.

SECTION 9
MOVING TEXT

The move command (m) moves a group of lines from one place to another in the buffer. To put the first three lines of the buffer at the end, enter:

```
1,3m$
```

The general format is

start line, *end line* m *after this line*

where *after this line* specifies where to put the text.

The lines to be moved can also be specified by context searches. To reverse the two paragraphs

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

type:

```
/Second/,/end of second/m/First/-1
```

The -1: moves the text before the line specified. Dot is set to the last line moved.

SECTION 10
USING SPECIAL CHARACTERS

10.1. General

The following characters have special meaning to ed when used in context searches and in the substitute command:

```
^   $   .   []   *   \   &
```

10.2. Period

On the left side of a substitute command or in a search with /.../, the period (.) stands for any single character. Thus, the search

```
/x.y/
```

finds any line where x and y occur and are separated by a single character, as in

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with the repetition character (*). Thus, a* is a shorthand for any number of a's, and .* matches any number of any characters. The expression

```
s/.*/stuff/
```

changes an entire line and

```
s/.*,//
```

deletes all characters in the line up to and including the last comma (.* finds the longest possible match).

Since the period matches a single character, there is a way to deal with previously invisible characters printed by l.

Suppose there is a line that, when printed with the l command, appears as

```
....   th\07is   ....
```

The character string \Ø7 really represents a single character (Section 7.4), so typing

```
s/th.is/this/
```

matches the character set between the h and the i, whatever it is. Since the period matches any single character, the command

```
s/./,/
```

converts the first character on a line into a comma.

The period has several meanings, depending on its context. The command

```
.s/././
```

shows all three.

The first period is the number of the line being edited, also called line dot. The second period is a special character 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, a period is not special. Applying this command to the line

```
Now is the time.
```

results with

```
.ow is the time.
```

10.3. Backslash

The backslash (\) turns off any special meaning that the next character might have. In particular, \. converts . from a "match anything" into a period, so it can be used to replace the period in

```
Now is the time.
```

with a question mark like this:

```
s/\./?/
```

The pair of characters \. is interpreted by ed as a single period.

The backslash can also search for lines that contain a special character. To look for a line that contains

.PP

the search

/.PP/

is not adequate, because it finds a line

THE APPLICATION OF ...

since the . matches the letter A. However, the command /\.PP/ finds only lines that contain .PP.

The backslash can also turn off special meanings for characters other than period. For example, to find a line that contains a backslash, precede one backslash with another as in /\\/ Similarly, search for a forward slash (/) with /\//

The backslash turns off the meaning of the immediately following /, so that it does not terminate the /.../ construction prematurely.

Any character can be used instead of slash to delimit the elements of an s command, but slashes must be used for context searching. For instance, in a line that contains many slashes, such as //exec //sys.fort.go // etc... a colon can be used as the delimiter. To delete all the slashes, type s:/::g

Exercise 8

Find two substitute commands to convert the line \x\.\y into the line \x\y

Here are several solutions to verify.

```
s/\\\.//
s/x../x/
s/..y/y/
```

10.4. Dollar Sign

Dollar sign ($) stands for the end of the line. To add the word time to the end of the line

Now is the

use the dollar sign s/$/ time/ to get

    Now is the time

A space must appear before time in the substitute command, or the result is

    Now is thetime

To convert the line

    Now is the time, for all good men,
    into
    Now is the time, for all good men.

the command needed is s/,$/./ The $ sign here provides context to make specific which comma is meant. Without it, the s command operates on the first comma, to produce

    Now is the time. for all good men,

As another example, to convert

    Now is the time.
    into
    Now is the time?

use s/.$/?/

The dollar sign has multiple meanings depending on context. In the line $s/$/$/ the first dollar sign refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

10.5. Circumflex

The circumflex (^) stands for the beginning of the line. To look for a line that begins with the, use /^the/ to narrow the context and arrive at the desired word more easily.

The other use of ^ inserts text at the beginning of a line. The command s/^/ / places a space at the beginning of the current line.

Special characters can be combined. To search for a line that contains only the characters .PP use the command /^\.PP$/

10.6. Asterisk

A character followed by an asterisk (*) stands for a variable number of consecutive occurrences of that character. A line can look like this:

text x y text

where text stands for a lot of text and there is an undetermined number of spaces between the x and the y.

To replace all the spaces at once, use

```
s/x *y/x y/
```

Thus x *y means "an x, as many spaces as there are then a y."

The asterisk can be used with any character, not just space. If the original example were

text x--------y text

then all - signs can be replaced by a single space with the command

```
s/x-*y/x y/
```

To change a line entered as text x.......y text

turn off the special meaning of dot (a match of any single character) with a backslash, as in

```
s/x\.*y/x y/
```

The because \.* means "as many periods as possible."

There are times when the pattern .* is exactly what is needed. For example, to change

```
Now is the time for all good men ....
into
Now is the time.
```

use .* to remove everything after the for with the command s/ for.*/./

Zero is a legitimate number of possible occurrences. For example, for a line

text xy text x y text

the command s/x *y/x y/ was entered. The first xy matches this pattern, since it consists of an x, zero spaces, and a y. The result is that the substitute acts on the first xy, and does not touch the later one, which actually contains some intervening spaces.

The way around this is to specify a pattern like /x *y/ which describes an x, a space, then as many more spaces as possible, that is, one or more spaces, then a y.

The command to convert an x into y s/x*/y/g when applied to the line abcdef produces yaybycydyeyfy This is because zero is a legal number of matches. There are no x's at the beginning of the line, and no-x gets converted to a y. There are no x's between a and b, so the non-x (zero characters) is converted into y. This process continues down the string. To solve the problem, write s/xx*/y/g where xx* is one or more x's.

10.7. Brackets

The brackets ([]) match any element of the character class within them.

To delete any numbers that appear at the beginning of all lines of a file, use the construction

```
[0123456789]*
```

This matches zero or more digits. Thus, the command

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class. The only special characters inside the brackets are ^ in the initial position and - between characters; even the backslash does not have a special meaning.

To search for special characters, for example, use

```
/[.\$^[]/
```

Within [...], the [is not special. To get a] into a character class, make it the first character.

To abbreviate the digits, use [0-9]. Similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase letters.

Specify a class that means "none of the following characters" by beginning the class with a circumflex. For example,

```
[^0-9]
```

stands for any character except a digit. To find the first line that does not begin with a tab or space, search with command

```
/^[^(space)(tab)]/
```

Within a character class, the ^ has a special meaning only if it occurs at the beginning. As an exercise, verify that

```
/^[^^]/
```

finds a line that does not begin with a circumflex.

10.8. Ampersand

The ampersand (&) is used to save typing. Suppose the line

```
Now is the time
```

must be changed to

```
Now is the best time
```

The command s/the/the best/ can be used, but it is redundant to repeat the the. The ampersand eliminates that repetition. On the right side of a substitute, the ampersand means "whatever was just matched," so the command s/the/& best/ && stands for the. For example, to parenthesize a line, regardless of its length, use s/.*/(&)/

The ampersand can occur more than once on the right side: s/the/& best and & worst/ makes the original line into

```
Now is the best and the worst time
```

and s/.*/&? &!!/ converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, use the backslash to turn off the special meaning. The command

```
s/ampersand/\&/
```

converts the word into the symbol. Ampersand has its special meaning only on the right side of a substitute command, not on the left side.

SECTION 11
USING GLOBAL COMMANDS

11.1. Global g

Global commands operate on the entire buffer instead of an individual line.

The global command (g) executes one or more ed commands on all lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain peling. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which prints only the last line substituted. Another difference is that the g command does not give a ? if it does not find peling, but the s command does.

Use these examples to see the difference between the global command g and the g following a substitute command. These g's occur at different places in the command line and have different meanings.

11.2. Global v

The v command is the same as g, except that the commands are executed on every line that does not match the string following v. For example:

```
v/ /d
```

deletes every line that does not contain a blank.

11.3. Advanced Global Commands

The global commands g and v perform one or more editing commands on all lines that either contain (with g) or do not

contain (with v) a specified pattern.

The pattern that goes between the slashes can be anything used in a line search or in a substitute command; the same rules and limitations apply.

The command g/^\./p prints all the formatting commands in a file because these lines begin with a dot. (Section 13 describes use of backslash to escape dot.)

The command that follows g or v can be anything. So g/^\./d deletes all lines that begin with . and g/^$/d deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command to change and print each affected line for verification. For example, to change the word zeus to ZEUS everywhere and verify that it worked, enter g/zeus/s//ZEUS/gp The // in the substitute command means "the previous pattern," in this case, zeus. The p command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making 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, dot is set to that line, and the command executed. This means that it is possible for the command that follows a g or v to use addresses or set dot. The command g/^\.PP/+ prints the line that follows each .PP command. Remember that + means "one line past dot." The command g/topic/?^\.SH?1 searches for each line that contains topic, scans backwards until it finds a line that begins .SH (a section heading) and prints the line that follows that, thus showing the section headings under which topic is mentioned.

Finally, g/^\.EQ/+,/^\.EN/-p prints all the lines between lines beginning with .EQ and .EN formatting commands.

The g and v commands can also be preceded by line numbers to search only those in the range specified.

11.4. Advanced Multiline Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is often cumbersome. As an example, suppose the task is to change x to y and a to b on all lines that contain thing. Then the commands

```
g/thing/s/x/y/\
s/a/b/
```

are sufficient. The backslash (\) signals the g command that the set of commands continues on the next line and terminates on the first line that does not end with \. A substitute command cannot be used to insert a new line within a g command.

To match the last pattern that was actually executed, use:

```
g/x/s/x/y/\
s/a/b/
```

To execute a, c, and i commands under a global command, add a backslash at the end of each line except the last. Thus, to add a .nf and .sp command before each .EQ line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a . to terminate the i command unless there are further commands under the global.

SECTION 12
SUBSTITUTING NEW LINES

Ed provides a facility for splitting a single line into two or more shorter lines by substituting a new line. As the simplest example, suppose a line is unmanageably long. If it looks like

    text xy text

it can be broken between the x and the y like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. The \ at the end of a line makes the following new line there no longer special.

Make a single line into several lines with this same mechanism. The word very in a long line can be underlined by splitting very onto a separate line and preceding it with the roff formatting command .ul.

    text a very big text

The command

```
s/ very /\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word very by the line .ul, and eliminating the spaces around the very.

When a new line is substituted, dot points at the last line created.

SECTION 13
MANIPULATING LINES

13.1. Join Lines

Lines can be joined together with the j command. If dot is set to the first of the lines

```
Now is
 the time
```

The j command joins them. A blank has been added at the beginning of the second line because the command itself does not cause blanks to be added.

By itself, a j command joins line dot to line dot+1. Any contiguous set of lines can also be joined by specifying the starting and ending line numbers. For example, 1,$jp joins all the lines into one line and prints it.

13.2. Rearrange Lines

Lines can be rearranged by tagging the pieces of the pattern by enclosing them between \(and \) and then rearranging the pieces. On the left side of a substitution, whatever matched that part is remembered and available for use on the right side. On the right side, the symbol \1 refers to whatever matched the first pair, \2 to the second pair, and so on.

For example, to convert a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

to a file in the form

```
A. B. Smith
C. Jones
```

use the command 1,$s/^\([^,]*\), *\(.*\)/\2 \1/ The first \(...\) matches the last name (any string up to the comma) and is referred to on the right side with \1. The second \(...\) is whatever follows the comma and any spaces, and is referred to as \2.

When this type of editing is performed, use the global commands g or v followed by p to print each substitution as it is made.

SECTION 14
MANIPULATING ADDRESSES

14.1. Line Addressing

Line addressing is the method used to specify what lines are to be affected by editing commands. Constructions like 1,$s/x/y/ start on line 1 and specify a change on all lines.

14.2. Address Arithmetic

Line numbers such as . and $ can be combined with + and - in a process called address arithmetic. For example, $-1 is a command to print the next-to-last line of the current file (that is, one line before line $). To see how much was entered in a previous editing session, use $-5,$p to print the last six lines.

The command .-3,.+3p prints from three lines before the current line to three lines after. The + can be omitted, so the command .-3,.3p is identical in meaning.

The - and + can be used as line numbers by themselves. The - by itself is a command to move up one line in the file. Several minus signs can be strung together to move back that many lines. For example, --- moves up three lines, as does -3. Thus -3,+3p is also identical to the previous examples.

Since - is shorter than -1, constructions such as -,.s/bad/good/ are useful. This changes bad to good on the previous line and on the current line.

The + and - can be used in combination with searches using /.../ and ?...?, and with $. The search /thing/-- finds the line containing thing, and positions dot two lines before it.

SECTION 15
DOING REPEATED SEARCHES

The construction // is a shorthand for "the previous thing that was searched for," whatever it was. This can be repeated as many times as necessary. The search can also go backwards. The command ?? searches for the same thing, but in the reverse direction.

The // can also be used as the left side of a substitute command to mean the most recent pattern. The command

```
/horrible thing/
s//good/p
```

finds the line containing horrible thing, prints the line, changes horrible thing to good, and prints the changed line.

To go backwards and change a line, enter

```
??s//good/
```

The & can be used on the right side of a substitute to stand for the character that was matched. The command

```
//s//& &/p
```

finds the next occurrence of whatever was searched for last, replaces it with two copies of itself, then prints the line.

SECTION 16
USING DEFAULT LINE REFERENCES

One of the most effective ways to speed up editing is always knowing what lines will be affected by a command and the value of dot when a command finishes.

If a search command

```
/thing/
```

is issued, dot points at the next line that contains thing. No address is required with commands

s to make a substitution on that line

p to print it

l to list it

d to delete it

a to append text after it

c to change it

i to insert text before it

If no match occurs, the position of dot is unchanged. This is also true if dot is at the only thing when the command is issued. The same rules hold for searches that use ?...?; the only difference is the direction of the search.

The delete command d leaves dot pointing at the line that followed the last deleted line. If line $ gets deleted, however, dot points at the new last line.

The line-changing commands a,c, and i all affect the current line. If no line number is given with them, a appends text after the current line, c changes the current line, and i inserts text before the current line.

Commands a,c, and i move dot to the last line entered. For example, the commands

```
a
 ... text ...
 ... botch ...          (minor error)
.
s/botch/correct/       (fix line)
a
 ... more text ...
```

can be given without specifying any line number for the substitute command or for the second append command. Alternatively, use

```
a
 ... text ...
 ... horrible botch ...      (major error)
.
c                            (replace entire line)
 ... fixed line ...
```

The r command reads a file into the text being edited, either at the end if no address is given, or after the specified line if there is an address. In either case, dot points at the last line read. Remember that 0r reads a file in at the beginning of the text.

The w command writes the entire file. If the command is preceded by one line number, that line is written. If it is preceded by two line numbers, that range of lines is written. The w command does not change dot; the current line remains the same, regardless of what lines are written. This is true even if there is a command such as

```
/^\.AB/,/^\.AE/w abstract
```

involving a context search.

The s command positions dot on the last line that changed. If there were no changes, then dot is unchanged.

With the text

```
x1
x2
x3
```

the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. With the three lines

```
x1
y2
y3
```

the same command changes and prints only the first line and positions dot there.

SECTION 17
USING THE SEMICOLON

In ed, the semicolon (;) can be used like comma, except that a semicolon forces dot to be set where the line numbers are being evaluated. In effect, the semicolon moves dot. &.lp Searches with /.../ and ?...? start at the current line and move forward or backward until they either find the pattern or return to the current line. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, the command

```
/a/,/b/p
```

would be expected to print all the lines from the ab to the bc. Instead, both searchs start from the same point and they both find the line that contains ab. The result is to print a single line. Worse, if there had been a line with a b in it before the ab line, the print command would be in error, since the second line number would be less than the first; it is illegal to try to print lines in reverse order.

The comma separator for line numbers does not set dot as each address is processed. Instead, each search starts from the same place. Thus, in this example, the command

```
/a/;/b/p
```

prints the range of lines from ab to bc. After the a is found, dot is set to that line, then b is searched for, starting beyond that line.

To find the second occurrence of thing, enter

```
/thing/;//
```

This finds the first occurrence of thing, sets dot to that line, then finds the second and prints only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

As an exercise, try printing the third or fourth occurrence in either direction.

To find the first occurrence of something in a file, starting at an arbitrary place within the file, use Ø;/thing/ This starts the search at line 1.

SECTION 18
INTERRUPTING THE EDITOR

Pressing the INTERRUPT, DELETE, RUBOUT, or BREAK key while ed is doing a command restores the state in effect before the command began. An interrupt during reading or writing a file, making substitutions, or deleting lines stops the command in an unpredictable state and does not always change dot.

Printing does not change dot until the printing is done. Thus, if the DELETE key is pressed while a file is being printed, dot is still where it was when the p command was started.

SECTION 19
MANIPULATING FILES

19.1. General

In addition to editor commands, other commands exist to manipulate files. Manipulating files includes changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

19.2. Change the Name of a File

To change a file name, use mv.

    mv oldname newname

This program moves the file from the old name to the new name. For example, to change a file named memo into one called paper, enter

    mv memo paper

NOTE

If there is already a file with the new name, its present contents are overwritten by the information from the old file. Also, a file cannot be moved to itself. So

    mv x x

is illegal.

19.3. Copy a File

Copy a file with the cp command. The format of cp is

    cp original copy

to copy original into copy. To save a file called good choose a name (here savegood) then type

```
cp good savegood
```

This copies good onto savegood, so that there are two identical copies of the file good. If savegood previously contained something, it is overwritten.

To restore the original state of good, enter

```
mv savegood good
```

which erases savegood, or

```
cp savegood good
```

to retain a safe copy.

19.4. Remove a File

To remove a file forever, use the rm command. The entry

```
rm savegood
```

permanently erases the file called savegood.

19.5. Put Two or More Files Together

Collecting two or more files into one is performed with cat (short for concatenate).

To combine the files file1 and file2 into a single file called bigfile, enter

```
cat file1 file2 >bigfile
```

The > before bigfile means to take the output of the cat command and put it into bigfile. As with cp and mv, anything that was already in bigfile is destroyed.

More than two files can be combined. The command

```
cat file1 file2 file3 ... >bigfile
```

collects many files.

19.6. Adding Text to the End of a File

To add one file to the end of another, use the >> construction. This is identical to >, except that instead of

overwriting the old file, it simply adds text at the end. Thus, enter

```
cat goodl >>good
```

to add goodl to the end of good. If good did not previously exist, this makes a copy of goodl called good.

19.7. Insert One File into Another

Suppose that a file called memo needs the file called table to be inserted just after the reference to Table 1. That is, in memo somewhere is a line that says

> Table 1 shows that ... and the data contained in table goes there.

Edit memo, find Table 1, and add the file table by entering

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table
```

The critical line is the last one; the r command reads the file table and inserts it immediately after the referenced line.

19.8. Write Part of a File

It is possible to split into a separate file the table from the previous example. In the file being edited, there are the lines

```
.TS
...[lots of stuff]
.TE
```

To isolate the table in a separate file called table, first find the start of the table (the .TS line), then write out the table

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
&.)1
```

All these steps can be consolidated with

```
/^\.TS/;/^\.TE/w table
```

The w command can write out a group of lines instead of the whole file.
In fact, a single line can be written by giving one line number instead of two. For example, if there is complicated line that is going to be needed later, save to avoid retyping it.
Enter

```
a
...lots of stuff...
...complicated line...
.
.w  temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

19.9. Move Lines

To move a paragraph from its present position in a paper to the end, use the editor move command (m).

The m command takes up to two line numbers in front that tell what lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between line1 and line2 after line3.

If dot is at the first line of the paragraph beginning with .PP, type

```
.,/^\.PP/-m$
```

The order of two adjacent lines can be reversed by positioning the first one after the second. If dot is at the first line, the command

```
m+
```

moves line dot to a position one line after the first line. If dot is at the second line, the command m-- interchanges the two lines.

The m command is more succinct and direct than writing, deleting, and rereading. The main difficulty with the m command is that if patterns are used to specify both the lines being moved and the target, they must be specified properly. Doing the job a step at a time makes it easier to verify at each step that the desired result is accomplished. Issue a w command before doing any complicated commands. If there is an error, it is easy to back up.

19.10. Mark a Line

Ed provides a facility for marking a line with a particular name to later reference it by name, regardless of its line number. This can be handy for moving lines and for keeping track of them as they move. The mark command is k. The command kx assigns the name x to the current line, where x is any single lowercase letter. (To mark a line for which the line number is known, precede the k with the line number.) Refer to the marked line with the address

```
'x
```

For example, to move a block of text, find the first line of the block to be moved, and mark it with ka.
Then find the last line and mark it with kb.
Now position dot where the text is to go and enter
'a,'bm.

Only one line can have a particular mark name associated with it at any given time.

19.11. Copy Lines

Ed provides another command, called t (for transfer) for making a copy of a group of one or more lines. This is often easier than writing and reading.

The t command is identical to the m command, except that instead of moving lines it duplicates them at the place named. Thus 1,t duplicates the entire file that is being edited.

A more common use for t is for creating a series of lines that differ only slightly. For example, type

```
a
.......... x ......... (long line)
.
t.          (make a copy)
s/x/y/      (change it a bit)
t.          (make third copy)
s/y/z/      (change it a bit)
```

and so on.

19.12. Temporary Escape

The escape command (!) provides a way to temporarily leave the editor for a ZEUS command and immediately return to the editor.

Entering

```
!any ZEUS command
```

suspends the current editing state and executes the command asked for. When the command finishes, ed prints another prompt and editing can be resumed. Any ZEUS command, including another ed, can be entered following the escape.

SECTION 20
SUPPORTING TOOLS

20.1. General

There are several tools and techniques based on the editor. In this section are some introductory examples of these tools.

20.2. Grep

To find all occurrences of some word or pattern in a set of files, use the program grep. The search patterns described in the document are often called "regular expressions," and "grep" stands for

```
g/re/p  (get / regular expression / print)
```

That describes exactly what grep does--it prints every line in a set of files that contains a particular pattern. Thus grep 'thing' file1 file2 file3 ... finds thing wherever it occurs in any of the files listed. Grep also indicates the file in which the line was found for any further file manipulation.

The pattern represented by thing can be any pattern that can be used in ed. Always enclose the pattern in single quotes if it contains any nonalphabetic characters. These characters carry special meaning in the ZEUS command interpreter (Section 15).

There is also a way to find lines that do not contain a pattern. The command

```
grep -v 'thing' file1 file2 ...
```

finds all lines that do not contain thing. The -v must occur in the position shown. Given grep and grep -v, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain x but not y, use grep x file... | grep -v y The notation | is a pipe command, which causes the output of the first command to be used as input to the second command.

20.3. Editing Scripts

To execute a complicated set of editing operations on a set of files, make up a script, that is, a file that contains the operations to perform. Then apply this script to each file.

For example, to change every Zeus to ZEUS and every bad to good in a large number of files, put into the file script the lines

```
g/Zeus/s//ZEUS/g
g/bad/s//good/g
w
q
```

Now enter

```
ed file1 <script
ed file2 <script
...
```

This causes ed to take its commands from the prepared script.

20.4. Sed

Sed (stream editor) processes unlimited amounts of input. Sed copies its input to its output, applying one or more editing commands to each line of input.

As an example, to change Zeus to ZEUS as in the previous example without rewriting the files, use the command

```
sed 's/Zeus/ZEUS/g' file1 file2 ...
```

This applies the command s/Zeus/ZEUS/g to all lines from the files specified and copies all lines to the output. The advantage of using sed is that it handles input too large for ed. All the output can be collected in one place, and either saved in a file or piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file with a slightly more complex syntax. To take commands from a file, for example, use sed -f cmdfile input-files...

APPENDIX A
SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of ed commands is the command name, perhaps preceded by one or two line numbers, and, in the case of e, r, and w, followed by a file name. Only one command is allowed per line, but a p command can follow commands other than e, r, w, and q.

a: Append (add) lines to the buffer at line dot unless a different line is specified. Appending continues until dot is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text that follows. The new lines are terminated by a dot, as with a. If no lines are specified, line dot is changed. Dot is set to the last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

e: Edit new file. Previous contents of the buffer are deleted.

f: Print current filename. If a name follows f, the current name is set to it.

g: The command

```
g/---/commands
```

executes the commands on those lines that contain ---, which can be any context search expression.

i: Insert lines before specified line or dot until a dot is typed on a new line. Dot is set to the last line inserted.

m: Move lines specified to a position after the line specified after m. Dot is set to the last line moved.

p: Print specified lines. If none are specified, print line dot. A single line number is equivalent to line number p. A single return prints .+1 (the next line).

q: Quit ed. Deletes all text in buffer if it is given twice in a row without first giving a w command.

r: Read a file into the end of the buffer unless a different location is specified. Dot is set to last line read.

s: The command

    s/string1/string2/

substitutes the characters string2 for string1 in the specified lines. If no lines are specified, it makes the substitution in line dot. Dot is set to the last line in which a substitution took place (if no substitution took place, dot is not changed). s changes only the first occurrence of string1 on a line. To change all occurences, type a g after the final slash.

v: The command

    v/---/commands

executes commands on those lines that do not contain ---, which can be any context search expression.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. The = by itself prints the value of $.

!: The line

    !command-line

causes command-line to be executed as a ZEUS command.

/-----/: Context search. Search for next line that contains this string of characters and print it. Dot is set to the line where string was found. Search starts at .+1, wraps around from $ to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at .-1, scan to 1, and wrap around to $.

"EX REFERENCE MANUAL"

This information is based on an article written by William Joy and revised for versions 3.5/2.13 by Mark Horton.

Table of Contents

SECTION 1
INTRODUCTION

1.1. Starting ex

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

When invoked, ex determines the terminal type from the TERM variable in the environment. If there is a TERMCAP variable in the environment, and the type of terminal described matches the TERM variable, that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) the editor will seek the description of the terminal in that file (rather than the default /etc/termcap).

If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable; otherwise, if there is a file .exrc in your HOME directory ex reads commands from that file, simulating a C shell ("Csh") source command. Option setting commands placed in EXINIT or .exrc will be executed before each editor session.

A command to enter fIex has the following format

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

where parameters within "[]" are optional. The most common case edits a single file with no parameters; that is,

ex name

The - command line option suppresses all interactive-user feedback (eg. prompts) and is useful in processing editor scripts in command files.

The -v option is is equivalent to using vi rather than ex.

The -t option is equivalent to an initial tag command, editing the file containing the tag and positioning the editor at its definition.

The -r option is used for recovery after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. (See Section 1.3.2.)

The -l option sets up for editing LISP, setting the showmatch and lisp options.

The -R option sets the readonly option at the start. (See Section 1.2.5.)

Name arguments indicate files to be edited.

A +command argument indicates that the editor should begin by executing the specified command. If command is omitted, it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

1.2. File Manipulation

1.2.1. Current File:

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

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

1.2.2. Alternate File:

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

1.2.3. Filename Expansion:

Filenames within the editor can be specified using the normal shell expansion conventions. In addition, the character `%' in filenames is replaced by the current file name and the character `#' by the alternate file name. This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an edit command after a No write since last change diagnostic is received.

1.2.4. Multiple Files and Named Buffers:

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

Ex has a group of named buffers for saving blocks of text while editing, and especially when editing more than one file. These are similar to the normal buffer, except that a limited number of operations are available on them; the buffers have names a through z. It is also possible to refer to A through Z; the upper case buffers are the same as the lower, but commands append to named buffers rather than replacing, if upper case names are used.

1.2.5. Read Only:

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

1.3. Exceptional Conditions

1.3.1. Errors and Interrupts:

When errors occur, ex (optionally) rings the terminal bell and, in any case, prints an error diagnostic. However, if the primary input is from a file, (as in editor script), editor processing terminates.

If an interrupt signal is received, ex prints "Interrupt" and returns to its command level. If the primary input is a file, then ex exits when this occurs.

1.3.2. Recovering from Hangups and Crashes:

If a hangup signal is received and the buffer has been modified since it was last written out, the editor attempts to preserve the buffer.

Also, if the system crashes, the system attempts to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file use the -r option. If editing the file resume, change to the directory where you were when the crash occurred, giving the command

ex -r resume

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

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

ex -r

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

1.4. Editing Modes

Ex has five distinct modes. The primary mode is command mode. Commands are entered in command mode when a `:' prompt is present, and are executed each time a complete line is sent.

In text input mode, ex gathers input lines and places them in the file. The append, insert, and change commands use text input mode. No prompt is printed in text input mode. This mode is left by typing a `.' alone at the beginning of a line, and command mode resumes.

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

1.5. Command Structure

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

1.5.1. Command Parameters:

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses are discussed below. A number of commands also can take a trailing count specifying the number of lines to be involved in the command. Thus, the command "1Øp" prints the tenth line in the buffer while "delete 5" will delete five lines from the buffer, starting with the current line. Some commands take other information or parameters; this information is given after the command name.

1.5.2. Command Variants:

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

1.5.3. Flags After Commands:

The flag commands `#' (abbreviation for the `nu' command), `p' (print command), and `l' (list command) can be placed after many commands. In this case, the flag command is executed after the primary command completes. Since ex normally prints the new current line after each change, `p' is rarely necessary.

Any number of `+' or `-' characters can also be given with these flags. If they appear, the specified offset is applied to the current line value before the flag command is executed.

1.5.4. Comments:

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

1.5.5. Multiple Commands per Line:

More than one command can be placed on a line by separating each pair of commands by a `|' character. However, the global (`g') commands, comments, and the shell escape `!' must be the last command on a line, as they are not terminated by a `|'.

1.5.6. Reporting Large Changes:

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the report option (Section 3.1.19). This feedback helps to detect undesirably large changes so that they can be quickly and easily reversed with an undo. command. After commands with more global effect such as global or visual, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

1.6. Command Addressing

1.6.1. Addressing Primitives:

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

n The nth line in the editor's buffer, lines being numbered sequentially from 1.

$ The last line in the buffer.

% An abbreviation for ``1,$'', the entire buffer.

+n -n
An offset relative to the current buffer line. The forms `.+3' `+3' and `+++' are all equivalent; if the current line is line 100, they all address line 103.

/pat/ ?pat?
Scan forward and backward respectively for a line containing pat, a regular expression (Section 2.2.1). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing pat, then the trailing / or ? can be omitted. If pat is omitted or explicitly empty, the last regular expression specified is located. The forms `/' and `?' scan using the last regular expression used in a scan; after a substitute, `//' and `??' would scan using the substitute's regular expression.

Marking with quotes
Before each non-relative motion of the current line is marked with a tag, subsequently referred to as `"' (two forward quotes). This makes it easy to refer or return to this previous context. Marks can also be established by the mark command, using single lower case letters x and the marked lines referred to as `'x.'

1.6.2. Combining Addressing Primitives:

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

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

SECTION 2
EDIT COMMANDS

2.1. Command Descriptions

The following form is a prototype for all ex commands:

[address] **command** [!] [parameters]
[count] [flags]fR [!] [parameters]
[count] [flags]

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

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

(.) **append** abbr: **a**
text
.

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

a!
text

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

args

The file names from the "ex" command line are printed. The current file name is delimited by `[' and `]'.

cd directory

The cd command is a synonym for chdir.

(. , .) **change** count abbr: **c**
text
.

Replaces the specified lines with the input text. The current line becomes the last line input, if no lines were input; the command is the same as delete.

c!
text
.

The variant toggles autoindent during the change.

chdir directory

The specified directory becomes the current directory. If no directory is specified, the current value of the home option is used as the target directory. After a chdir the current file is not considered to have been edited so that write restrictions on pre-existing files apply.

(. , .) **copy** addr flags abbr: **co**

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

Cshell abbr: **cs**

A new C shell is created; when it terminates, editing resumes

(. ,.) **delete** buffer count flags
abbr: **d**

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

edit file abbr: **e**
ex file

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last write command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is not a binary file such as a directory, a block or character special file other than /dev/tty, a terminal, a binary, or executable file the editor reads the file into its buffer.

If reading the file produces no errors, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded.

If none of these errors occurred, the file is considered edited. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line `.' at the last line read. If this command is executed from within open or visual, the current line is initially the first line of the file.

e! file

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

e +n file

causes the editor to begin at line n rather than at the last line; n can also be an editor command containing no spaces; for example, "+/pat" (to search for the pattern, pat).

file abbr: f

prints the current file name, whether it has been `[Modified]' since the last write command, or whether it is read only In addition, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line is printed.

In the rare case that the current file is `[Not edited]' this is also noted; in this case use the form **w!** to write to

the file, since the editor is not sure that a **write** will not destroy a file unrelated to the current contents of the buffer.

file *file*

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

(1 , $) **global** /*pat*/ *cmds* abbr: **g**

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

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

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

g! /*pat*/ *cmds* abbr: **v**

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

(.)**insert** abbr: **i**
text
.

places the given text before the specified line. The current line is left at the last line input; if there were

no lines input, it is left at the line before the addressed line. This command differs from append only in the placement of text.

i!
text
.

The variant toggles autoindent during the insert.

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

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

j!

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

(.) **k** x

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

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

prints the specified lines so that tab and end-of-line characters are represented; tabs are printed as `^I' and the end of each line is marked with a trailing `$'. The current line is left at the last line printed.

(.) **mark** x

gives the specified line mark x, a single lower case letter. The x must be preceded by a blank or a tab. The addressing form 'x (forward quote followed by the tag) addresses this line; The current line is not affected by this command.

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

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

next abbr: **n**

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

n!

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

n filelist
n +command filelist

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

(. , .) **number** count
flags abbr: **#** or **nu**

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

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

enters editing in open mode at each addressed line. If pat is given, the cursor placed initially at the beginning of the string matched by the pattern. To exit this mode use Q (see Introduction to Display Editing with Vi for more details).

preserve abbr: **pre**

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a write command has resulted in an error. Following a

preserve you will receive mail which explains how to recover your file.

(. , .)**print** count abbr: **p** or **P**

prints the specified lines with non-printing characters printed as control characters `^x'; delete (octal 177) is represented as `^?'. The current line is left at the last line printed.

(.)**put** buffer abbr: **pu**

puts back previously deleted or yanked lines. Normally used with delete to effect movement of lines, or with yank to effect duplication of lines. If no buffer is specified, the last deleted or yanked text is restored. By using a named buffer, text can be restored that was saved there at any previous time.

No modifying commands can intervene between the delete or yank and the put. Lines are moved between files only by using a named buffer.

quit abbr: **q**

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

Ex also issues a diagnostic if there are more files in the argument list.

q!

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

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

places a copy of the text of the given file in the editing buffer after the specified line. If no file is given the current file name is used. The current file name is not changed unless there is none, in which case file becomes the

current name. The sensibility restrictions for the edit command apply here also. If the file buffer is empty and there is no current name, ex treats this as an edit command.

Address `Ø' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the edit command when the read successfully terminates. After a read the current line is the last line read in ex. Within open and visual the current line is set to the first line read rather than the last.

(.) **read** !command

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

recover file

recovers file from the system save area. Used after a accidental hangup of the phone, or a system crash or preserve command. However, the system saves a copy of the file being edited, only changes have been made to the file. You will be notified by mail when a file is saved.

rewind abbr: **rew**

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

rew!

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

set parameter

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

Giving an option name followed by a `?' causes the current value of that option to be printed. The `?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form `set option' to turn them on

or `set nooption' to turn them off; string and numeric options are assigned via the form `set option=value'.

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

shell abbr: **sh**

A new Bourne shell is created; when it terminates, editing resumes.

source file abbr: **so**

reads and executes commands from the specified file. Source commands can be nested.

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

On each specified line, the first instance of pattern pat is replaced by replacement pattern repl. If the global indicator option character `g' appears, all instances are substituted.

If the confirm indicatior `c' appears, the line is typed with the **pat** string marked with with `↑' characters; typing a `y' causes the substitution to be performed and any other input causes no change to take place. After a substitute, the current line is the last line substituted.

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

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

If pat and repl are omitted, the last substitution is repeated. This is a synonym for the `&' command. (See Section 2.2.5)

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

The t command is a synonym for copy.

ta tag

The focus of editing switches to the location of tag, switching to a different line in the current file where it is defined or, if necessary, to another file. If you have modified the current file before giving a tag command, write it out; giving another tag command, specifying no tag reuses the previous tag.

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

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

undo abbr: **u**

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

Undo always marks the previous value of the current line `.' as `'' (two forward quotes). After an undo the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as global and visual the current line regains it's pre-command value after an undo.

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

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

version abbr: **ve**

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

(.) **visual** type count flags abbr: **vi**

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

visual file
visual +n file

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

(1 , $) **write** file abbr: **w**

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

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

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

(1 , $) **write>>** file abbr: **w>>**

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

w! name

overrides the checking of the normal write command, and writes to any file which the system permits.

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

writes the specified lines into command. Note the

difference between **w!** which overrides checks and **w !** which writes to a command.

wq name

Similiar to a write and then a quit command.

wq! name

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

xit name

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

(. , .)**yank** buffer count abbr: **y**

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

(**.+1**) **z**count

print the next count lines, default window.

(.) **z** type count

prints a window of text with the specified line at the top. If type is `-' the line is placed at the bottom; a `.' causes the line to be placed in the center. A count gives the number of lines to be displayed rather than double the number specified by the scroll option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed. Forms `z=' and `z↑' also exist; `z=' places the current line in the center, surrounds it with lines of `-' characters and leaves the current line at this line. The form `z↑' prints the window before `z-' would. The characters `+', `↑' and `-' can be repeated for cumulative effect.

! command

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

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

(addr , addr) **!** command

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

($) =

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

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

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

^D

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

(.+1 , .+1)
(.+1 , .+1) |

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

(. , .) & options count flags

repeats the previous substitute command.

(. , .) ~ options count flags

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

2.2. Regular Expressions and Substitute Replacement Patterns

2.2.1. Regular Expressions:

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

2.2.2. Magic and Nomagic:

The regular expressions allowed by ex are constructed in one of two ways depending on the setting of the magic option. The ex and vi default setting of magic gives quick access to a powerful set of regular expression metacharacters. The disadvantage of magic is that the user must remember that these metacharacters are magic and escape them (precede them with the character `') to use them as ``ordinary'' characters.

With nomagic, the default for edit, regular expressions are much simpler; there are only two metacharacters. The power of the other metacharacters is still available by preceding the ordinary character with a `'. Note that `' is always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is magic. To discern what is true with nomagic, remember that the only special characters in this case will be `↑' at the beginning

of a regular expression, `$' at the end of a regular expression, and `'. With nomagic the characters `~' and `&' also lose their special meanings related to the replacement pattern of a substitute.

2.2.3. Basic Regular Expression Summary:

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

↑ At the beginning of a pattern forces the match to succeed only at the beginning of a line.

$ At the end of a regular expression forces the match to succeed only at the end of the line.

. Matches any single character except the new-line character.

\< Forces the match to occur only at the beginning of a ``variable'' or ``word''; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> Similar to `<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

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

2.2.4. Combining Regular Expression Primitives:

The concatenation of two regular expressions matches the leftmost and the longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single

character matching) regular expressions mentioned above can be followed by the character `*' to form a regular expression which matches any number of adjacent occurrences (including Ø) of characters matched by the regular expression it follows.

The character `~' can be used in a regular expression, and matches the text which defined the replacement part of the last substitute command. A regular expression can be enclosed between the sequences ` and `)' with side effects in the substitute replacement patterns.

2.2.5. Substitute Replacement Patterns:

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

Other metasequences possible in the replacement pattern are always introduced by the escaping character `\'. The sequence `\n' is replaced by the text matched by the n-th regular subexpression enclosed between `\(' and `\)'. When nested, parenthesized subexpressions are present, n is determined by counting occurrences of `\(' starting from the left.

The sequences `\u' and `\l' cause the immediately following character in the replacement to be converted to upper- or lower-case, respectively, if this character is a letter. The sequences `\U' and `\L' turn such conversion on, either until `E' or `\' is encountered or until the end of the replacement pattern.

SECTION 3
OPTIONS

3.1. Option Descriptions

3.1.1. Autoindent:

autoindent, ai default: noai

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

If the user enters lines of text, they continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line starts aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop enter **^D**. The tab stops, going backwards, are defined at multiples of the shiftwidth option. You cannot backspace over the indent, except by typing a **^D**.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the autoindent is discarded). Also specially processed in this mode are lines beginning with an `↑' and immediately followed by a **^D**. This repositions the input at the beginning of the line, but retaining the previous indent for the next line. Similarly, a `0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

Autoindent is not meaningful in global commands or when the input is not a terminal.

3.1.2. Autoprint:

autoprint, ap default: ap

Causes the current line to be printed after each delete, copy, join, move, substitute, t, undo or shift command.

This has the same effect as supplying a trailing `p' to each such command. Autoprint is suppressed in globals, and only applies to the last of many commands on a line.

3.1.3. Autowrite:

autowrite, aw default: noaw

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

3.1.4. Beautify:

beautify, bf default: nobeautify

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

3.1.5. Directory:

directory, dir default: dir=/tmp

specifies the directory in which ex places its buffer file. If this directory is not writable, the editor exits abruptly when it cannot create its buffer there.

3.1.6. Error Bells:

errorbells, eb default: noeb

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

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

3.1.7. Hard Tabs:

hardtabs, ht default: ht=8

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

3.1.8. Ignore Case:

ignorecase, ic default: noic

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

3.1.9. Lisp:

lisp default: nolisp

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

3.1.10. List:

list default: nolist

All printed lines are displayed showing tabs and end-of-lines as in the list command.

3.1.11. Magic:

magic default: magic for ex and vi+
Nomagic for edit.

If nomagic is set, the number of regular expression meta-characters is greatly reduced, with only `↑' and `$' having special effects. In addition the metacharacters `~' and `&' of the replacement pattern are treated as normal characters. All the normal metacharacters can be made magic when nomagic is set by preceding them with a `\'.

3.1.12. Number:

number, nu default: nonumber

causes all output lines to be printed with their line numbers. In addition, each input line prompts with the line number it will have.

3.1.13. Open:

open default: open

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

3.1.14. Optimize:

optimize, opt default: no optimize

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

3.1.15. Paragraphs:

paragraphs, para default: para=IPLPPPQPP LIbp

specifies the paragraphs for the `{' and `}' operations in open and visual. The pairs of characters in the option's value are the names of the macros which start paragraphs.

3.1.16. Prompt:

prompt default: prompt

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

3.1.17. Read Only:

read only default: noreadonly

puts the editor in "read only" mode (as discussed in Section 1.2.5) to protect files from being overwritten. This option

is set when involving the editor with the `-R' command line option.

3.1.18. Redraw:

redraw default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal. (For example, during insertions in visual the characters to the right of the cursor position are refreshed as each input character is typed.) This option is useful only at very high speeds.

3.1.19. Report:

report default: report=5 for ex and visual
report=2 for edit.

specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines provides feedback as to the scope of its changes. For commands such as global, open, undo, and visual, which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a global command on the individual commands performed.

3.1.20. Scroll:

scroll default: scroll=1/2 window

determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode z command (double the value of scroll).

3.1.21. Sections:

sections default: sections=SHNHH HU

specifies the section macros for the `[[' and `]]' operations in open and visual. The pairs of characters in the option's value are the names of the macros which start paragraphs.

3.1.22. Shell:

shell, sh default: sh=/bin/csh

gives the path name of the shell forked for the shell escape command `!', and by the shell command. The default is taken from SHELL in the environment, if present.

3.1.23. Shiftwidth:

shiftwidth, sw default: sw=8

gives the width for a software tab stop; this is used in reverse tabbing with **^D** when using autoindent to append text and by the shift commands.

3.1.24. Showmatch:

showmatch, sm default: nosm

In open and visual mode, when a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen; this is extremely useful with lisp.

3.1.25. Slowopen:

slowopen, slow terminal dependent

affects the display algorithm used in visual mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent (see Introduction to Display Editing with Vi for more details).

3.1.26. Tabstop:

tabstop, ts default: ts=8

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

3.1.27. Taglength:

taglength, tl default: tl=0

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

3.1.28. Ttytype:

ttytype from environment TERM

The terminal type of the output device.

3.1.29. Term:

term from environment TERM

The terminal type of the output device.

3.1.30. Terse:

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

3.1.31. Warn:

warn default: warn

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

3.1.32. Window:

window default: speed dependent

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

w300, w1200, w9600

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

3.1.33. Wrapscan:

wrapscan, ws default: ws

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

3.1.34. Wrapmargin:

wrapmargin, wm default: wm=Ø

defines a margin for automatic wrapover of text during input in open and visual modes. (See " Introduction to Display Editing with Vi for details.)

3.1.35. Writeany:

writeany, wa default: nowa

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

3.1.36. Limitations:

Editor limits that the user is likely to encounter are as follows:

(1) 1Ø24 characters per line

(2) 256 characters per global command list

(3) 128 characters per file name

(4) 128 characters in the previous inserted and deleted text in open or visual

(5) 1ØØ characters in a shell escape command

(6) 63 characters in a string valued option

(7) 3Ø characters in a tag name

(8) 250000 lines in the file is silently enforced.

SECTION 4
EX/EDIT COMMAND SUMMARY

Ex and edit are text editors, used for creating and modifying files of text on the UNIX computer system. Edit is a variant of ex with features designed to make it less complicated to learn and use. In terms of command syntax and effect, the editors are essentially identical; this command summary applies to both.

The summary is a quick reference for users already acquainted with edit or ex. Fuller explanations of the editors are available in the documents Edit: A Tutorial (a self-teaching introduction), and the Ex Reference Manual (the comprehensive reference source for both edit and ex).

In the examples included with the summary, commands and text entered by the user are printed in **boldface** to distinguish them from responses printed by the computer.

4.1. The Editor Buffer

In order to perform its tasks the editor sets aside a temporary work space, called a buffer, separate from the user's permanent file. Before starting to work on an existing file the editor makes a copy of it in the buffer, leaving the original untouched. All editing changes are made to the buffer copy, which must then be written back to the permanent file in order to update the old version. The buffer is erased at the end of the editing session.

4.2. Editing: Command and Text Input Modes

During an editing session there are two usual modes of operation: command mode and text input mode. In command mode, the editor issues a colon prompt (:) to show that it is ready to accept and execute a command. In text input mode, on the other hand, there is no prompt and the editor merely accepts text to be added to the buffer. Text input mode is initiated by the commands append, insert, and change, and is terminated by typing a period as the first and only character on a line.

4.3. Line Numbers and Command Syntax

The editor tracks lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. At any given time the editor is positioned at one of these lines; this position is called the current line. Generally, commands that change the contents of the buffer print the new current line at the end of their execution.

Most commands can be preceded by one or two line-number addresses which indicate the lines to be affected. If one number is given, the command operates on that line only; if two, on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though exceptions to this rule should be noted. The print command by itself, for instance, causes one line, the current line, to be printed at the terminal.

The summary shows the number of line addresses that can be prefixed to each command as well as the defaults assumed if they are omitted. For example, (.,.) means that up to 2 line-numbers can be given, and that if none is given the command operates on the current line.

In the address prefix notation, "." stands for the current line and "$" stands for the last line of the buffer. If no such notation appears, no line-number prefix can be used.

Some commands take trailing information; only the more important instances of this are mentioned in the summary.

4.4. Open and Visual Modes

Besides command and text input modes, ex and edit provide on some CRT terminals, other modes of editing, open and visual. In these modes the cursor can be moved to individual words or characters in a line. The commands then given are very different from the standard editor commands; most do not appear on the screen when typed. The document, Introduction to Display Editing with Vi provides a full discussion.

4.5. Special Characters

Some characters take on special meanings when used in context searches and in patterns given to the substitute command. For edit, these are "^" and "$", meaning the beginning and end of a line, respectively.

Ex has the following additional special characters:

```
*    &    .    [    ]    ~
```

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning.

SECTION 5
COMMAND SUMMARY

(.)append abbr:a

begins text input mode, adding lines to the buffer after the line specified. Appending continues until ``.'' is typed alone at the beginning of a new line, followed by a carriage return. Øa places lines at the beginning of the buffer.

```
:a
Three lines of text
are added to the buffer
after the current line.
.
:
```

(.,.)change abbr:c

places a copy of the specified lines after the line indicated by addr. The example places a copy of lines 8 through 12, inclusive, after line 25.

```
:8,12co 25
Last line copied is printed
:
```

(.,.)delete abbr:d

removes lines from the buffer and prints the current line after the deletion.

```
:13,15d
New current line is printed
:
```

ex or **edit** file **abbr:e**
ex! or **edit!** file **abbr:e!**

clears the editor buffer and copies into it the named file, which becomes the current file. This is a way of shifting to a different file without leaving the editor. The editor issues a warning message if this command is used before saving changes made to the file already in the buffer; using the form **"e!"** overrides this protective mechanism.

```
:e ch1Ø
No write since last change (:edit! overrides)
:e! ch1Ø
"ch1Ø" 3 lines, 62 characters
:
```

file name **abbr:f**

If followed by a name, renames the current file to name. If used without name, prints the name of the current file.

```
:f ch9
"ch9" [Modified] 3 lines ...
:f
"ch9" [Modified] 3 lines ...
:
```

(1,$)global abbr:g

global/pattern**/**commands

searches the entire buffer (unless a smaller range is specified by line-number prefixes) and executes commands on every line with an expression matching pattern.

```
:g/nonsense/d
:
```

(1,$)global! abbr:g! or **v**

executes commands on lines that do not contain the expression pattern.

(.)insert abbr:i

inserts new lines of text immediately before the specified line. Differs from append only in that text is placed before, rather than after, the indicated line. In other words, **1i** has the same effect as **Øa**.

```
:1i
These lines of text will
be added before line 1.
.
:
```

(.,.+1)join abbr:j

Join lines together, adjusting white space (spaces and tabs) as necessary.

```
:2,5j
Resulting line is printed
:
```

(.,.)list abbr:l

prints lines marking the end of a line with a "$" and tabs as "^I".

```
:91
This is line 9$
:
```

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

moves the specified lines to a position after the line indicated by addr.

```
:12,15m 25
New current line is printed
:
```

(.,.)number abbr:nu or **#**

prints each line preceded by its buffer line number.

```
:nu
  1Ø  This is line 1Ø
:
```

(.)open abbr:o

enters open mode; see Introduction to Display Editing with Vi for more details. Type 'Q' to get back into normal editor command mode. Edit is designed to prevent accidental use of the open command.

preserve abbr:pre

saves a copy of the current buffer contents as though the system had just crashed. This is for use in an emergency when a write command has failed.

```
:preserve
File preserved.
:
```

(.,.)print abbr:p

Prints the text of line(s).

```
:+2,+3p
The second and third lines
after the current line.
:
```

quit abbr:q
quit! abbr:q!

ends the editing session. You will receive a warning if you have changed the buffer since last writing its contents to the file. In this event you must either type **"w"** to write, or type **"q!"** to exit from the editor without saving your changes.

```
:q
No write since last change
:q!
%
```

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

places a copy of file in the buffer after the specified line. Address Ø is permissible and causes the copy of file to be placed at the beginning of the buffer. The read command does not erase any text already in the buffer. If no line number is specified, file is placed after the current line.

```
:Ør newfile
"newfile" 5 lines, 86 characters
:
```

recover file **addr:rec**

Retrieves a copy of the editor buffer after a system crash, editor crash, phone line disconnection, or preserve command.

(.,.)substitute abbr:s

substitute/pattern**/**replacement**/**
substitute/pattern**/**replacement**/gc**
replaces the first occurrence of pattern on a line with replacement. Including a **g** after the command changes all occurrences of pattern on the line. The **"c"** option allows the user to confirm each substitution before it is made; see the Ex Reference Manual for details.

```
:3p
Line 3 contains a misstake
:s/misstake/mistake/
Line 3 contains a mistake
:
```

undo abbr:u

reverses the changes made in the buffer by the last buffer-editing command. Note that this example contains a notification about the number of lines affected.

```
:1,15d
15 lines deleted
new line number 1 is printed
:u
15 more lines in file ...
old line number 1 is printed
:
```

(1,$)**write** file **abbr:w**
(1,$)**write!**file **abbr:w!**

copies data from the buffer onto a permanent file. If no file is named, the current filename is used. The file is automatically created if it does not yet exist. A response containing the number of lines and characters in the file indicates that the write has been completed successfully. The editor's built-in protections against overwriting existing files will in some circumstances inhibit a write. The form **"w!"** forces the write, confirming that an existing file is to be overwritten.

```
:w
"file7" 64 lines, 1122 characters
:w file8
"file8" File exists ...
:w! file8
"file8" 64 lines, 1122 characters
:
```

(.)z count **abbr:z**

prints a screen full of text starting with the line indicated; or, if count is specified, prints that number of lines. Variants of the z command are described in the Ex Reference Manual.

!command

executes the remainder of the line after "!" as a UNIX command. The buffer is unchanged by this, and control is returned to the editor when the execution of command is complete.

```
:!date
Fri Jun 9 12:15:11 PDT 1978
!
:
```

control-d

prints the next scroll of text, normally half of a screen. See the Ex Reference Manual for details of the scroll option.

(.+1)<cr>

An address alone followed by a carriage return causes the line to be printed. A carriage return by itself prints the line following the current line.

```
:<cr>
the line after the current line
:
```

/pattern/

Searches for the next line in which pattern occurs and prints it.

```
:/This pattern/
This pattern next occurs here.
:
```

//

repeats the most recent search.

```
:/
This pattern also occurs here.
:
```

?pattern?

searches in the reverse direction for pattern.

??

repeats the most recent search, moving in the reverse direction through the buffer.

FILE SYSTEM CHECK PROGRAM (FSCK) REFERENCE MANUAL

Preface

This document describes how the file system check program (fsck) maintains file system integrity. More broadly, this document presents the normal updating of the file system, discusses the possible causes of file system corruption, and describes the corrective actions used by fsck. Both internal functions of the fsck program and the interaction between the program and the operator appear in this document.

Section 1 gives a brief introduction to fsck, and Section 2 discusses normal updating of the file system. File system corruption is described in Section 3. Section 4 presents the set of corrective actions used by fsck. Error conditions and operator actions are explained in the Appendix.

In this document, the sentence structure "fsck can ..." often appears; for example, "Fsck can clear the inodes." This is a shorthand notation for the process of fsck prompting the operator, the operator responding to continue fsck, and fsck actually performing the action, in this case, clearing the inodes (setting its contents to zero).

Additional information on fsck appears in the System 8000 ZEUS Reference Manual (part number 03-3255) under fsck(1).

Table of Contents

SECTION 1
INTRODUCTION

When the ZEUS Operating System is brought up, the file system check program (fsck) must be run. Fsck is an interactive file system program that uses the redundant structural information in the ZEUS file system to perform consistency checks. Fsck detects file inconsistencies and reports them to an operator who elects to fix or ignore them. This precautionary measure helps to ensure a reliable environment for file storage on disk.

Every file activity (creation, modification, or deletion) updates at least one of the five data blocks that ZEUS uses to monitor files. Fsck checks for matches in the contents of redundant fields among these blocks, and for matches between information in the blocks and the files themselves. When any error is found, fsck reports it to an operator. Most errors allow for operator intervention to continue running fsck or to terminate it. Serious errors, such as illegal options, cause fsck to terminate.

SECTION 2
UPDATE OF THE FILE SYSTEM

2.1. General

Every time a file is created, modified, or removed, the ZEUS Operating System performs a series of file system updates on the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks. Update requests are honored in a specific order to yield a consistent file system. Knowing this order makes it easier to understand what happens when a problem occurs, and to repair a corrupted file system.

2.2. Super-Block

The super-block contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The root file system is always mounted, and the super-block of a mounted file system is written to the file system whenever the file system is unmounted or a sync command is issued.

2.3. Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system on closure of the file associated with the inode, and when a sync command is issued.

2.4. Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect, and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers, and a triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

2.5. Data Blocks

A data block contains file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

2.6. Free-List Blocks

The free-list blocks list all blocks that are not allocated to the super-block (only the first free-list block), inodes, indirect blocks, or data blocks. Each free-list block contains a count of the entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

SECTION 3
CORRUPTION OF THE FILE SYSTEM

3.1. General

The most common reasons for corruption of a file system are improper shutdown and hardware failure.

3.2. Improper System Shutdown and Startup

Improper shutdown procedures include forgetting to sync the system prior to halting the CPU, physically write-protecting a mounted file system, and taking a mounted file system off-line.

Improper startup procedures include not checking a file system for inconsistencies and not repairing inconsistencies.

3.3. Hardware Failure

Any piece of hardware can fail at any time. Failures range from a bad block of a disk pack to a nonfunctional disk controller.

SECTION 4
DETECTION AND CORRECTION OF CORRUPTION

4.1. General

A quiescent file system (one that is unmounted and not being written on) can be checked for structural integrity by performing consistency checks of the redundant data that is part of the file system. A quiescent state is important during the file system check because of the multipass nature of the fsck program. Fsck discovers each file inconsistency, reports it to the operator, and allows for interactive corrective action.

This section discusses how to discover inconsistencies and take corrective actions for super-blocks, inodes, indirect blocks, data blocks containing directory entries, and free-list blocks.

4.2. Super-Block

The super-block is most prone to corruption because every change to the file system's block or inodes modifies the super-block. Corruption most frequently occurs when the computer is halted and the last command involving the output of the file system was not a sync command.

Check the super-block for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

4.2.1. File-System Size and Inode-List Size: These sizes are critical because all other checks of the file system depend on them, and because fsck can only check for them being within reasonable bounds. The file system size must be larger than both the number of blocks used by the super-block and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535.

4.2.2. Free-Block List: The free-block list starts in the super-block and continues through the free-list blocks of the file system. Each free-list block is checked for a list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were

found. If anything is wrong with the free-block list, fsck can rebuild it, excluding all blocks in the list of allocated blocks.

Fsck checks the list count for the first free-block for a value of less than zero or greater than 50. It also checks each block number for a value of less than the first data block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is nonzero, the next free-list block is read in, and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

4.2.3. Free-Block Count: The super-block contains a count of the total number of free blocks within the file system. Fsck compares this count to the number of blocks it found free within the file system and, if they do not agree, replaces the count in the super-block with the actual free-block count.

4.2.4. Free-Inode Count: The super-block contains a count of the total number of free inodes within the file system. Fsck compares this count to the number of inodes it found free within the file system and, if they do not agree, replaces the count in the super-block with the actual free-inode count.

4.3. Inodes

A large quantity of active inodes increases the likelihood of corruption. Fsck sequentially checks the list of inodes for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.3.1. Format and Type: Each inode contains a mode word that describes the type and state of the inode. Valid inode types are regular, directory, special block, and special character. Valid inode states are unallocated, allocated, and neither allocated nor unallocated (incorrectly formatted as a result of bad data being written into the inode list through hardware failure). Fsck can clear the inode.

4.3.2. Link Count: A count of the total number of directory entries linked to the inode is contained in each inode. Fsck verifies this count by traversing down the total directory structure starting from the root directory and calculating an actual link count for each inode.

If the stored link count is nonzero and the actual link count is zero, no directory entry appears for the inode. Fsck can link the disconnected file to the lost+found directory.

If the stored and actual link counts are nonzero and unequal, a directory entry may have been added or removed without the inode being updated. Fsck can replace the stored link count with the actual link count.

4.3.3. Duplicate Blocks: Each inode contains a list, or pointers to lists, (indirect blocks) of all the blocks claimed by the inode. Fsck compares each block number claimed by an inode to a list of already allocated blocks. If there are any inconsistencies, fsck can clear both inodes.

If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, fsck makes a partial pass of the inode list to find the duplicate block. Fsck needs to examine the files associated with these inodes to determine which inode is corrupted and should be cleared (most frequently, this is the inode with the earlier modify time). This error condition occurs when using a file system with blocks claimed by both the free-block list and by other parts of the file system.

A large number of duplicate blocks in an inode is often due to an indirect block not being written to the file system.

4.3.4. Bad Blocks: Each inode contains a list, or pointer to lists, of all the blocks claimed by the inode. Fsck checks each block number claimed by an inode for a value within the range bounded by the first data block (minimum) and the last block (maximum) in the file system. A block number outside the range is called a bad block number.

A large number of bad blocks in an inode can be due to an indirect block not being written to the file system.

4.3.5. Size Checks: Each inode contains a 32-bit (four-byte) size field that contains the number of characters in the file associated with the inode. Fsck checks this field for inconsistencies such as directory sizes that are not a multiple of 16 characters, and for the number of blocks actually used not matching the number indicated by the inode size.

A directory inode in the ZEUS file system has the directory bit set on in the inode mode word. The directory size must be a multiple of 16 because a directory entry contains 16 bytes of information, two bytes for the inode number and 14 bytes for the file or directory name. Fsck warns of directory misalignment, but cannot gather sufficient information to correct the problem.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in an inode by the number of characters per block (512), rounding up, and adding one block for each indirect block associated with the inode. If this computed number does not match the actual number of blocks, fsck warns of a possible file-size error, but does not correct it because ZEUS does not insert blocks into files that are created in random order.

4.4. Indirect Blocks

Since indirect blocks are owned by an inode, inconsistencies in the indirect blocks affect the inode. Fsck checks that blocks are not already claimed by another inode, and that block numbers are not outside the range of the file system. The procedures discussed in Sections 4.3.3 and 4.3.4 are iteratively applied to each level of indirect blocks.

4.5. Data Blocks

The two types of data blocks are plain data blocks and directory blocks. Plain data blocks contain the information stored in a file and are not checked by fsck.

Directory data blocks contain directory entries and are checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for . (current directory) and .. (parent directory), and directories that are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, fsck can remove that directory entry. This condition usually occurs when the data block containing the directory entries are modified and written to the file system, and the inode is not yet written.

If a directory entry inode number is pointing beyond the end of the inode list, fsck can remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for . must be the first entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, fsck can replace them with the correct values.

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, fsck links the directory back into the file system in the lost+found directory. This condition can be caused by inodes being written to the file system without the corresponding directory data blocks being written.

4.6. Free-List Blocks

Free-list blocks are owned by the super-block, and inconsistencies in free-list blocks directly affect the super-block.

Fsck can check for a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

Section 4.2.2 contains a discussion of detection and correction of the inconsistencies associated with free-list blocks.

APPENDIX A
FSCK ERROR CONDITIONS

A.1. Conventions

Fsck is a multipass file system check program with each file system pass invoking a different phase of the fsck program. After the initial setup, fsck performs successive phases on each file system, checks blocks and sizes, path names, connectivity, reference counts, and the free-block list (which might be rebuilt), and performs some cleanup.

When an inconsistency is detected, fsck reports it to the operator. If a response is required, fsck prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the phase of the fsck program in which they occur. The error conditions that occur in more than one phase are discussed in the Initialization Section.

A.2. Initialization

Before a file system check can be performed, certain tables must be set up and certain files must be opened. This section lists error conditions resulting from initializing tables and opening files; specifically, it lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file.

C OPTION ?

C is not a legal option of fsck; legal options are -y, -n, -s, -S, and -t. Fsck terminates on this error condition (fsck(1)).

BAD -t OPTION

The -t option is not followed by a file name. Fsck terminates on this error condition (fsck(1)).

INVALID -s ARGUMENT, DEFAULTS ASSUMED

The -s option is not suffixed by 3, 4, blocks-per-cylinder, or blocks-to-skip. Fsck assumes a default value of 400 blocks-per-cylinder and nine blocks-to-skip (fsck(1)).

INCOMPATIBLE OPTIONS: -n and -s

It is not possible to salvage the free-block list without modifying the file system. Fsck terminates on this error condition (fsck(1)).

CAN'T GET MEMORY

Fsck's request for memory for its virtual memory tables failed. This should never happen. Fsck terminates on this error condition. See an experienced fsck user.

CAN'T OPEN CHECKLIST FILE: F

The default file system checklist file F (usually /etc/checklist) cannot be opened for reading. Fsck terminates on this error condition. Check access modes of F.

CAN'T STAT ROOT

Fsck's request for statistics about the root directory / failed. This should never happen. Fsck terminates on this error condition. See an experienced fsck user.

CAN'T STAT F

Fsck's request for statistics about the file system F failed. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

F IS NOT A BLOCK OR CHARACTER DEVICE

Fsck has a wrong regular file name that it ignores, and it continues checking the next file system given. Check file type of F.

CAN'T OPEN F

The file system F cannot be opened for reading. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

SIZE CHECK: fsize X isize Y

More blocks are used for the inode list Y than there are blocks in the file system X, or there are more than 65,535 inodes in the file system. Fsck ignores this file system and continues checking the next file system given (Section 4.2.1).

CAN'T CREATE F

Fsck's request to create a scratch file F failed. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

CANNOT SEEK: BLK B (CONTINUE?)

Fsck's request for moving to a specified block number B in the file system failed. This should never happen. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

CANNOT READ: BLK B (CONTINUE?)

Fsck's request for reading a specified block number B in the file system failed. This should never happen. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

CANNOT WRITE: BLK B (CONTINUE?)

Fsck's request for writing a specified block number B in the file system failed. The disk is write-protected. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

A.3. Phase 1: Check Blocks and Sizes

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

UNKNOWN FILE TYPE I=I (CLEAR?)

The mode word of the inode I indicates that the inode is not a special character inode, regular inode, or directory inode (Section 4.3.1).

Possible responses to the CLEAR? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another allocated inode with a zero link

count is found, this error condition is repeated.

NO Terminate the program.

LINK COUNT TABLE OVERFLOW (CONTINUE?)

An internal table for fsck containing allocated inodes with a link count of zero has no more room. Recompile fsck with a larger value of MAXLNCNT.

Possible responses to the CONTINUE? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO Terminate the program.

B BAD I=I

Inode I contains block number B with a number lower than the number of the first data block in the file system, or greater than the number of the last block in the file system. This error condition invokes the EXCESSIVE BAD BLKS error condition in Phase 1 if inode I has too many block numbers outside the file system range. This error condition always invokes the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.3.4.

EXCESSIVE BAD BLKS I=I (CONTINUE?)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system, or greater than the number of last block in the file system associated with inode I (Section 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system.

NO Terminate the program.

B DUP I=I

Inode I contains block number B which is already claimed by another inode. This error condition invokes the EXCESSIVE DUP BLKS error condition in Phase 1 if inode I has too many block numbers claimed by other inodes. This error condition always invokes Phase 1B and the BAD/DUP error condition in Phase 2 and Phase 4 (Section 4.3.3).

EXCESSIVE DUP BLKS I=I (CONTINUE?)

There is more than a tolerable number (usually 1Ø) of blocks claimed by other inodes (Section 4.4.3).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with the next inode in this file system. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system.

NO Terminate the program.

DUP TABLE OVERFLOW (CONTINUE?)

An internal table in fsck containing duplicate block numbers has no more room. Recompile fsck with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another duplicate block is found, this error condition repeats.

NO Terminate the program.

POSSIBLE FILE SIZE ERROR I=I

The inode I size does not match the actual number of blocks used by the inode. This is only a warning (Section 4.3.5).

DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning (Section 4.3.5).

PARTIALLY ALLOCATED INODE I=I (CLEAR?)

Inode I is neither allocated nor unallocated (Section 4.3.1).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

A.4. Phase 1B: Rescan for More Duplicates

When a duplicate block is found in the file system, the system is rescanned to find the inode that previously claimed that block. This section lists the error condition when the duplicate block is found.

B DUP I=I

Inode I contains block number B, which is already claimed by another inode. This error condition always invokes the BAD/DUP error condition in Phase 2. Inodes that have overlapping blocks can be determined by examining this error condition and the DUP error condition in Phase 1 (Section 4.3.3).

A.5. Phase 2: Check Path Names

This phase removes directory entries pointing to inodes with error conditions from Phase 1 and Phase 1B. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

ROOT INODE UNALLOCATED. TERMINATING.

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program terminates (Section 4.3.1).

ROOT INODE NOT DIRECTORY (FIX?)

The root inode (usually inode number 2) is not a directory inode (Section 4.3.1).

Possible responses to the FIX? prompt are:

YES Make the root inode's type a directory. If the root inode's data blocks are not directory blocks, a very large number of error conditions are produced.

NO Terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE?)

Phase 1 or Phase 1B found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system (Sections 4.3.3 and 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, this results in a large number of other error conditions.

NO Terminate the program.

I OUT OF RANGE I=I NAME=F (REMOVE?)

A directory entry F has an inode number I which is greater than the end of the inode list (Section 4.5).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE?)

A directory entry F has an inode I without allocate mode bits. The owner O, mode M, size S, modify time T, and file name F are printed (Section 4.5).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE?)

Phase 1 or Phase 1B found duplicate blocks or bad blocks associated with directory entry F, directory inode I. The owner O, mode M, size S, modify time T, and directory name F are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE?)

Phase 1 or Phase 1B have found duplicate blocks or bad blocks associated with directory entry F, inode I. The owner O, mode M, size S, modify time T, and file name F are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

A.6. Phase 3: Check Connectivity

This phase checks the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full lost+found directories.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT?)

The directory inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of directory inode I are printed (Sections 4.5 and 4.3.2).

Possible responses to the RECONNECT? prompt are:

YES Reconnect directory inode I to the file system in the directory for lost files (usually lost+found). This invokes the lost+found error condition in Phase 3 if there are problems connecting directory inode I to lost+found. This also invokes thee CONNECTED error condition in Phase 3 if the link was successful.

NO Ignore this error condition. This always invokes the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY

There is no lost+found directory in the root directory of the file system; fsck ignores the request to link a directory in lost+found. This always invokes the UNREF error condition in Phase 4. Check access modes of lost+found (fsck(1)).

SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the lost+found directory in the root directory of the file system; fsck ignores the request to link a directory in lost+found. This always invokes the UNREF error condition in Phase 4. Remove unnecessary entries in lost+found or make lost+found larger (fsck(1)).

DIR I=I1 CONNECTED. PARENT WAS I=I2

This is an advisory message indicating that a directory inode I1 is successfully connected to the lost+found directory. The parent inode I2 of the directory inode I1 is replaced by the inode number of the lost+found directory (Sections 4.5 and 4.3.2).

A.7. Phase 4: Check Reference Counts

This phase checks the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full lost+found directory, incorrect link counts for files, directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT?)

Inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Section 4.3.2).

Possible responses to the RECONNECT? prompt are:

YES Reconnect inode I to the file system in the directory for lost files (usually lost+found). This invokes the lost+found error condition in Phase 4 if there are problems connecting inode I to lost+found.

NO Ignore this error condition. This always invokes the CLEAR error condition in Phase 4.

SORRY. NO lost+found DIRECTORY

There is no lost+found directory in the root directory of the file system; fsck ignores the request to link a file in lost+found. This always invokes the CLEAR error condition in Phase 4. Check access modes of lost+found.

SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the lost+found directory in the root directory of the file system; fsck ignores the request to link a file in lost+found. This always invokes the clear error condition in Phase 4. Check size and contents of lost+found.

(CLEAR?)

The inode mentioned in the immediately previous error condition cannot be reconnected (Section 4.3.2).

Possible responses to the CLEAR? prompt are:

YES Deallocate the inode mentioned in the immediately previous error condition by setting its contents to zero.

NO Ignore this error condition.

LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X

The link count for the file inode I is X but should be Y. The owner O, mode M, size S, and modify time T are printed

(Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of file inode I with Y.

NO Ignore this error condition.

LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X

The link count for directory inode I is X but should be Y. The owner O, mode M, size S, and modify time T of directory inode I are printed (Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of inode I with Y.

NO Ignore this error condition.

LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X

The link count for F inode I is X but should be Y. The name F, owner O, mode M, size S, and modify time T are printed (Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of inode I with Y.

NO Ignore this error condition.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

File inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.2 and 4.5).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Directory inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Section 4.3.2 and 4.5).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by setting its contents to zero.

NO Ignore this error condition.

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Phase 1 or Phase 1B have found duplicate blocks or bad blocks associated with directory inode I. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by setting its contents to zero.

NO Ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX?)

The actual count of the free inodes does not match the count in the super-block of the file system (Section 4.2.4).

Possible responses to the FIX? prompt are:

YES Replace the count in the super-block by the actual count.

NO Ignore this error condition.

A.8. Phase 5: Check Free List

This section lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and an incorrect total free-block count.

EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE?)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value of less than the first data block in the file system or greater than the last block in the file system (Sections 4.2.2 and 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the free-block list and continue the execution of fsck. This error condition always invokes the BAD BLKS IN FREE LIST error condition in Phase 5.

NO Terminate the program.

EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE?)

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list (Section 4.2.2 and 4.3.3).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the free-block list and continue the execution of fsck. This error condition always invokes the DUP BLKS IN FREE LIST error condition in Phase 5.

NO Terminate the program.

BAD FREEBLK COUNT

The count of free blocks in a free-list block is greater than 50 or less than zero. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Section 4.2.2).

X BAD BLKS IN FREE LIST

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Sections 4.2.2 and 4.3.4).

X DUP BLKS IN FREE LIST

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Sections 4.2.2 and 4.3.3).

X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Section 4.2.2).

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX?)

The actual count of free blocks does not match the count in the super-block of the file system (Section 4.2.3).

Possible responses to the FIX? prompt are:

YES Replace the count in the super-block with the actual count.

NO Ignore this error condition.

BAD FREE LIST (SALVAGE?)

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system (Sections 4.2.2, 4.3.3, and 4.3.4).

Possible responses to the SALVAGE? prompt are:

YES Replace the actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.

NO Ignore this error condition.

A.9. Phase 6: Salvage Free List

This phase checks the free block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

DEFAULT FREE-BLOCK LIST SPACING ASSUMED

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of nine blocks-to-skip and 400 blocks-per-cylinder are used (fsck(1)).

A.10. Cleanup

Once a file system has been checked, cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

X FILES Y BLOCKS Z FREE

This is an advisory message indicating that the file system checked contained X files using Y blocks, leaving Z blocks free in the file system.

*******BOOT ZEUS (NO SYNC!)*******

This is an advisory message indicating that a mounted file system or the root file system has been modified by fsck. If ZEUS is not rebooted immediately, the work done by fsck may be undone by the in-core copies of tables ZEUS keeps.

*******FILE SYSTEM WAS MODIFIED*******

This is an advisory message indicating that the current file system was modified by fsck. If this file system is mounted or is the current root file system, fsck must be halted and ZEUS rebooted. If ZEUS is not rebooted immediately, the work done by fsck may be undone by the in-core copies of tables ZEUS keeps.

INDEX OF MESSAGES

(Alphabetically within each section)

INTIALIZATION

PHASE 1: CHECK BLOCKS AND SIZES

PHASE 1B: RESCAN FOR MORE DUPS

PHASE 2: CHECK PATH-NAMES

PHASE 3: CHECK CONNECTIVITY

PHASE 4: CHECK REFERENCE COUNTS

PHASE 5: CHECK FREE LIST

PHASE 6: SALVAGE FREE LIST

DEFAULT FREE-BLOCK LIST SPACING ASSUMED..... A-16

CLEANUP

*****BOOT ZEUS (NO SYNC!)*****..... A-16
*****FILE SYSTEM WAS MODIFIED*****..... A-16
X FILES Y BLOCKS Z FREE..... A-16

LEARN
COMPUTER-AIDED INSTRUCTION ON ZEUS

* This information is based on an article originally written by Brian W. Kernighan and Michael E. Lesk, Bell Laboratories.

Preface

This document describes the LEARN program and its seven Computer-Aided Instruction (CAI) scripts that provide lessons on the ZEUS Operating System. Since LEARN is a self-explanatory program, this document gives the theoretical background instead of detailed instructions on how to use it. The purpose of this document is to guide people preparing programs similar to LEARN, not to assist them in learning basic computer skills.

Section 1 contains a general introduction to LEARN and Section 2 states educational assumptions and design. The topic of each script appears in Section 3, and Section 4 describes how the LEARN program interprets the scripts. Conclusions about the LEARN experience are in Section 5.

Table of Contents

SECTION 1
INTRODUCTION

The system that teaches computer skills has two main parts: a driver called LEARN that interprets the scripts, and the scripts themselves. At present, there are seven Computer Aided Instruction (CAI) scripts:

1. first-time user introduction
2. basic file handling commands
3. ZEUS text editor (ed)
4. advanced file handling commands
5. eqn language for mathematical typing
6. the -ms macro package for document formatting
7. C programming language

The advantages of CAI scripts include the following:

- students are forced to perform the exercises
- students receive immediate feedback and confirmation of progress
- students progress at their own rate
- no schedule requirements are imposed
- lessons can be individually improved
- the computer is accessible to the student at the student's convenience.
- usage of high technology motivates students and maintains management interest

Since there is no one the student can question, CAI is comparable to a textbook, lecture series, or taped course rather than to a seminar. CAI has been used for many years in a variety of educational areas. Using the computer as a self-teaching device offers unique advantages; the skills developed to go through the script are exactly those needed to operate the computer; therefore, no effort is wasted.

SECTION 2
EDUCATIONAL ASSUMPTIONS AND DESIGN

2.1. Theoretical Assumptions

The best way to teach people how to do something is to have them do it. Scripts should not contain long explanations, but instead should frequently ask the student to do a task. Teaching is always by example; the typical lesson shows a small example of some technique and then asks the student to either repeat that example or produce a variation of it. All lessons are intended to be easy enough so that most students get most questions right, reinforcing the desired behavior.

After each correct response, the computer congratulates the student and indicates the lesson number that has just been completed, permitting the student to restart the script after that lesson. If the answer is wrong, the student is offered a chance to repeat the lesson.

It is assumed that there is no foolproof way to determine if the student truly "understands" what he or she is doing; the LEARN scripts measure performance, not comprehension.

The computer provides an immediate check of the correctness of what the student does. Unlike many CAI scripts, these scripts provide few facilities for dealing with wrong answers. In practice, if most of the answers are not right, the script is a failure. The solution to the problem of excessive student error is to provide a new, easier script. Anticipating possible wrong answers is an endless job; it is easier and better to provide a simpler script.

LEARN also provides a mechanical check on performance. If a student is unable to complete one lesson, that should not prevent access to the rest. The current version of LEARN allows the student to skip a lesson that he or she cannot pass. For example, a "no" answer to the "Do you want to try again?" question in Section 2-3 causes the program to go on to the next lesson.

There are valid objections to these assumptions, since some students object to not understanding what they are doing. Since writing a CAI script is more tedious than writing ordinary manuals, there are always alternatives to the scripts as a way of learning.

2.2. Types of Lessons

Most lessons are one of three types. The simplest lesson asks for a yes or no answer to a question. The student is given a chance to experiment before replying, and the lesson checks for the correct reply. Problems of this form are used sparingly.

The second type asks for a word or number as an answer. For example, a lesson on files might say

> How many files are there in the current directory?
> Type "answer N," where N is the number of files.

The student is expected to enter (perhaps after experimenting) a response similar to

```
answer 17
```

The idea of a substitutable argument (replacing N by 17) is difficult for nonprogrammer students, so the first few such lessons require special attention.

The third type of lesson is open-ended. A task is set for the student, appropriate parts of the input or output are monitored, and the student types:

```
ready
```

when the task is done.

2.3. Sample Lesson Display

The following sample is from the script that teaches file handling. It incorporates the open-ended and the word or number answer types of lessons. Most LEARN lessons are of this form. Student responses are shown in italics. The "$" is the system prompt.

A file can be printed on your terminal by using the "cat" command. Just say "cat file" where "file" is the file name. For example, there is a file named "food" in this directory. List it by saying "cat food"; then type "ready".

```
$ cat food
this is the file
named food.
$ ready
```

Good. Lesson 3.3a (1)

Of course, you can print any file with "cat". In particular, it is common to first use "ls" to find the name of a file and then "cat" to print it. Note the difference between "ls", which tells you the name of the file, and "cat", which tells you the contents. One file in the current directory is named for a President. Print the file, then type "ready".

```
$ cat President
cat: can't open President
$ ready
```

Sorry, that's not right. Do you want to try again? yes Try the problem again.

```
$ ls
.ocopy
X1
roosevelt
$ cat roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
$ ready
```

Good. Lesson 3.3b (Ø)

The "cat" command can also print several files at once. In fact, it is named "cat" as an abbreviation for "concatenate"....

2.4. Track Levels

In the files and editor scripts there are three tracks, differing in degrees of difficulty. The fastest script (sequence of lessons), roughly the bulk and speed of a typical tutorial manual, should be adequate for review and for well-prepared students. The next track, intended for most users, is about twice as long. The third and slowest track, which is often three or four times the length of the fast track, is intended for the most basic instruction. For example, the fast track presents an idea and asks for a variation on the example shown. The normal track first asks the student to repeat the example that was shown before attempting a variation. The lesson in Section 2.3 is from the third track.

The LEARN driver combines lessons in different ways to produce scripts in each track. For example, the fast track is

produced by skipping lessons from the slower track. The driver can also switch tracks, depending on the number of correct answers the student has given for the last few lessons.

SECTION 3
SCRIPTS

3.1. General Information

The present scripts follow a three-track theory. Care must be taken in lesson construction to see that every necessary fact is presented in every possible path throughout the scripts. In addition, it is desirable that every lesson have alternate successors to deal with student errors.

There are some preliminary skills that the student must know before any scripts can be tried. In particular, the student must know how to connect a ZEUS system, set the terminal properly, log in, and execute simple commands (for example LEARN itself). In addition, the character erase and line kill conventions (control-h and control-x) should be known. The student will need assistance for a few minutes to gain familiarity with these skills.

In existing scripts, the first few lessons are devoted to checking prerequisites. For example, before the student is allowed to proceed through the editor script, the script verifies that the student understands files and is able to type. Anyone proceeding through the scripts should get correct answers; otherwise, the system will be unsatisfactory both because the wrong habits are being learned and because the scripts make little effort to deal with wrong answers. Therefore, unprepared students should not be encouraged to continue with scripts.

3.2. First-Time User Script

The first-time user script covers a few important features of the system in very brief lessons. Here, I/O redirection, pipes, make files, the C compiler, and the ZEUS text editor (ed) are introduced.

3.3. Basic File Handling Script

It is assumed that the user of this script has basic knowledge of Script 1; it teaches the student about the ls, cat, mv, rm, cp and diff commands. It also deals with the abbreviation characters *, ?, and [ ] in file names. It does not cover pipes or I/O redirection, nor does it present the many options of the ls command.

3.4. Context Editor Script

This script trains students in the use of the ZEUS context editor, ed, a sophisticated editor using regular expressions for searching. All editor features except encryption, mark names, and ; in addressing are covered.

3.5. Advanced File Handling Script

The advanced file handling script, assuming the basic file handling script as a prerequisite, deals with ls options, I/O diversion, pipes, and supporting programs like pr, wc, tail, spell, and grep.

3.6. Eqn Language Script

This script covers the eqn language for typing mathematics and must be run on a terminal capable of printing mathematical symbols (for instance the DASI 300 and similar Diablo-based terminals). Most advanced lessons provide additional practice for students who are having trouble in the basic track.

3.7. -ms Script

The -ms script for formatting macros is a short, one-track script. However, the linear style of a single LEARN script is inappropriate for the macros, since the macro package is composed of many independent features, and few users need all of them.

3.8. C Language Script

The script on the language C has been partially converted to follow the order of presentation in The C Programming Language. The C script was never intended to teach C; rather it is a series of exercises for which the computer provides checking and a suggested solution.

SECTION 4
THE SCRIPT INTERPRETER

4.1. General Information

The LEARN program interprets scripts. It provides facilities to capture student responses and their effects, and simplifies the job of passing control to and recovering control from the student. This section describes the operation and use of the driver program, and indicates what is required to produce a new script. Readers interested only in the existing scripts should skip this section.

4.2. File Structure

The file structure used by LEARN is shown below. There is one parent directory named lib containing the script data. Within this directory are subdirectories, one for each subject where a course is available, one for logging (named log), and one where user subdirectories are created (named play). The subject directory contains master copies of all lessons, plus any supporting material for that subject. In a given subdirectory, each lesson is a single text file. Lessons are usually named systematically; the file that contains lesson n is called Ln.

```
lib

        play
                student1
                        files for student1...
                student2
                        files for student2...

        files
                LØ.1a   lessons for files course
                LØ.1b
                ...

        editor
                ...

        (other courses)

        log
```

Directory Structure for LEARN

When LEARN is executed, it makes a private directory for the user to work in, within the LEARN portion of the file system. A fresh copy of all the files used in each lesson is usually made by the lesson script each time a student starts a lesson. The student directory is deleted after each session; any permanent records must be kept elsewhere.

4.3. Requirements

Each lesson must contain the following basic items:

- the text of the lesson
- the set-up commands to be executed before the user gets control
- the data, if any, that the user is supposed to edit, transform, or otherwise process
- the evaluating commands to be executed after the user has finished the lesson, which decide whether the answer is right
- a list of possible successor lessons

LEARN minimizes the work of bookkeeping and installation, so that most of the effort involved in script production is in planning lessons, writing tutorial paragraphs, and coding tests of student performance.

4.4. Sequence of Events

LEARN first creates the working directory. Then, for each lesson, LEARN reads the text for the lesson and processes it a line at a time. The lines in the text are commands to the text interpreter to print something, to create a files, or to test something, text to be printed or put in a file, and other lines that are sent to the shell to be executed. One line in each lesson turns control over to the user, who can run any ZEUS command. The user mode terminates when the user types yes, ok, no, ready, or answer. At this point, the user's work is tested; if the lesson is passed, a new lesson is selected; if not, the old one is repeated.

4.5. Interpreted Script

To illustrate the flow of LEARN, the sample script from Section 2.3 is interpreted here.

Lines that begin with # are commands to the learn script interpreter. For example,

```
#print
```

causes printing of any text that follows, up to the next line that begins with a sharp. The command

```
#print file
```

prints the contents of file; it is the same as catfile. Both forms of #print have the added property that if a lesson is failed, the #print is not executed the second time; this avoids annoying the student by repeating the preamble to a lesson. The command

```
#create file name
```

creates a file of the specified name and copies any subsequent text up to a # in the file. This creates and initializes working files and reference data for the lessons. The command

```
#user
```

gives control to the student; each line typed is passed to the shell for execution. The #user mode is terminated when the student types one of the special keywords yes, ok, no, ready, or answer. At that time, the driver resumes interpretation of the script.

The yes and no responses return control to the script, where the answer can be evaluated with #match. Since ok is an alias for yes, the script writer can also use #match ok when an indication to proceed with the course is the only response needed.

The ready response returns control to the script but cannot be evaluated with #match. Instead, the user's previous responses are evaluated in some way. The answer response prepares for evaluation of the answer given. For instance, if the correct answer is 3, and the user responds with answer 3 then #match3 is used in the script to process that response. Anything the student types between the commands

```
#copyin
#uncopyin
```

is copied onto a file called .copy. This allows for interrogation of the student's responses upon regaining control. Between the commands

```
#copyout
#uncopyout
```

any material typed by the student for any program is copied to the file .ocopy. This allows interrogation of the effect of what the student typed.

Normally the student's input and the script commands are fed to the ZEUS command interpreter (the shell) one line at a time. A sequence of editor commands does not work, since the input to the editor must be handed to the editor, not to the shell. Accordingly, the material between the commands #pipe and #unpipe is fed continuously through a pipe so that such sequences work. If copyout is also desired, the copyout brackets must include the pipe brackets.

There are several commands for setting status after the student has attempted the lesson.

```
#cmp file1 file2
```

is an in-line implementation of cmp that compares two files for identity.

Following the command

```
#match stuff
```

the last line of the student's input is compared to stuff, and the success or fail status is set according to this comparison. Extraneous things like the word answer are stripped before the comparison is made. There can be several #match lines; this provides a convenient mechanism for handling multiple "right" answers. Any text up to a # on subsequent lines after a successful #match is printed, as shown next.

```
#print
What command will move the current line
to the end of the file?  Type
"answer COMMAND", where COMMAND is the command.
#copyin
#user
#uncopyin
#match m$
#match .m$
"m$" is easier.
#log
#next
63.1d 1Ø
```

```
#bad stuff
```

This is similar to #match, except that it corresponds to specific failure answers; this produces hints for particular wrong answers that have been anticipated by the script writer. The commands

```
#succeed
#fail
```

print a message upon success or failure (as determined by some other mechanism).

When the student types one of the "commands" yes,ok, or answer, the driver terminates the #user command, and evaluation of the student's work can begin. This can be done either by the built-in commands, such as #match and #cmp, or by status returned by normal ZEUS commands, typically grep and test. The last command should return status true (Ø) if the task is done successfully and false (nonzero) otherwise; this status return tells the driver whether or not the student has successfully passed the lesson.

Performance can be logged:

```
#log file
```

writes the date, lesson, user name and speed rating, and a success/failure indication on file. The command

```
#log
```

by itself writes the logging information in the logging directory within the LEARN hierarchy, and is the normal form. The commands

```
# cleanup
# nocleanup
```

are for directing the LEARN driver to clean up or ignore the temporary files created in a lesson. By default, learn cleans out the temporary files after each lesson. Specifically, all files that begin with a lowercase letter and are not ".c" files are deleted before the next lesson. The #nocleanup directive enables following lessons to depend on files already created or changed by the user, and #cleanup restores the default action at any time. The command

```
#next
```

is followed by a few lines, each with a successor lesson name and an optional speed rating on it. A typical set reads

```
25.1a   10
25.2a   5
25.3a   2
```

indicating that unit 25.1a is a suitable follow-on lesson for students with a speed rating of 10 units, 25.2a for student with a speed rating of 5 units, and 25.3a for students with a speed rating of 2 units. Speed ratings are maintained for each session per student; the rating is increased by one each time the student gets a lesson right and decreased by four each time the student gets a lesson wrong. Thus, the driver maintains a level at which the users get 80% right answers. The maximum rating is limited to 10, and the minimum is zero. The initial rating is zero unless the student specifies a different rating when starting a session.

If the student passes a lesson, a new lesson is selected, and the process repeats. If the student fails, a false status is returned, and the program reverts to the previous lesson and tries another alternative. If it cannot find another alternative, it skips forward a lesson.

If the student is unable to answer one of the exercises correctly, the driver searches for a previous lesson with a set of alternatives as successors (following the #next line). The program selects an alternative different from the one tried in the previous lesson.

Sophisticated scripts can be written to evaluate the student's speed of response, estimate the subjective merits of the answer, or to provide detailed analysis of wrong answers.

The driver program depends heavily on features of ZEUS that are not available on many other operating systems. Although some parts of LEARN might be transferable to other systems, some generality will be lost.

SECTION 5
CONCLUSIONS

The following are observations about nonprogrammers using LEARN.

A novice must have assistance with the mechanics of communicating with the computer to get through the first or second lesson. Once the first few lessons are passed, people can proceed on their own. Most students enjoy the system, and motivation matters a great deal.

The terminology used in the first few lessons is obscure to those inexperienced with computers. It would help if there were a low-level reference card to supplement the existing manual and reference card. The concept of "substitutable argument" is hard to grasp and requires help.

It takes an hour or two for a novice to get through the script on file handling. The total time for a novice to create new files and manipulate old ones is a few days, with perhaps half of each day spent on the machine.

The normal way of proceeding has been to have students in the same room with someone who knows ZEUS and the scripts. Thus, the student is not brought to a halt by difficult questions. The burden on the counselor is much lower than that on a teacher of a course. The students should be encouraged to proceed with instruction immediately prior to their actual use of the computer. They should exercise the scripts on the same computer and the same kind of terminal that they will later use for their real work, and their first few jobs with the computer should be relatively easy ones.

One disadvantage of training with LEARN is that students come to depend completely on the CAI system and do not try to read manuals or use other learning aids. This is unfortunate, not only because of the increased demands for completeness and accuracy of the scripts but because the scripts do not cover all of the ZEUS system. New users should have manuals (appropriate for their level) and read them; the scripts ought to be altered to recommend suitable documents and to urge students to read them.

From the student's viewpoint, the most serious difficulty is that there are lessons that simply cannot be passed. Sometimes this is due to poor explanations, but just as often it

is some error in the lesson itself, a wrong setup, a missing file, an invalid test for correctness, or some system facility that does not work on the local system as on the development system. It takes knowledge and a certain healthy arrogance on the part of the users to recognize that the fault is not theirs. Permitting the student to continue with the next lesson regardless does alleviate this, and the logging facilities make it easy to watch for lessons that no one can pass.

The biggest problem with some scripts, notably eqn, is that they are very slow. Another potential problem is that it is possible to break LEARN by pushing interrupt at the wrong time, by removing critical files, or any number of similar slips. The defenses against such problems have steadily been improved to the point where most students should not notice difficulties.

One area is more fundamental: LEARN currently does not allow ZEUS global commands to be executed. The most obvious is cd, which changes to another directory. The prospect of a student who is learning about directories moving to some random directory and removing files has prevented lessons on cd.

WRITING PAPERS WITH NROFF USING -ME *

*This information is based on an article written by Eric Allman,
Electronics Research Laboratory,
University of California at Berkeley.

Preface

This document describes the text processing facilities available on the ZEUS operating system via NROFF and the -me macro package. It is assumed that the reader already is generally familiar with the ZEUS operating system and a text editor such as **ex**. This is intended to be a casual introduction, and as such 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 The -me Reference Manual and The NROFF/TROFF Reference Manual.

NROFF, a computer program that runs on the ZEUS operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. 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.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in Section 4. The more complex requests which are not discussed in Section 2 are covered in Section 5. Section 6 discusses things you will need to know if you want to typeset documents. And Section 7 provides a more complete reference section. If you are a novice, you probably won't want to read beyond Section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the ZEUS shell:

nroff -me -T type files

where type describes the type of terminal you are outputting to. Common values are **dtc** for a DTC 300s (daisy-wheel type) printer and **lpr** for the line printer. 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 The NROFF/TROFF Reference Manual.

The word argument is used in this manual to mean a word or number which appears on the same line as a request which 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 which spaces four lines instead of one. Arguments are separated from the request and from each other by spaces.

Table of Contents

SECTION 1
BASICS

NROFF collects words from input lines, fills output lines with those words, justifies the right hand margin by inserting extra spaces in the line, and outputs 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,...
```

Not all input lines are text to be formatted. Some of the input lines are requests describing 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.

Keep the input lines short. Short input lines are easier to edit, and NROFF packs words onto longer lines. It is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. 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 as needed, but cannot 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 will get a space where not wanted, such as "mother- in-law".

1.1. 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 begining with spaces cause a break. For example, if the input is:

```
.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 will 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.

Fancier types of paragraphs, are described later.

1.2. 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 can be used, but avoid backslash and double quote marks. 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.

1.3. Double Spacing

NROFF double spaces output text automatically with the request **.ls 2**. Revert to single spaced mode by typing **.ls 1**.

1.4. Page Layout

Several requests 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 can be replaced with any values required; bold characters represent characters which 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 *N***c** 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 pica typewriter), 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 applies to one line only; after which it reverts 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. To center many lines but not count them, type:

```
.ce 1000
lines 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. To start a new line without performing any other action, use **.br**.

1.5. Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. To underline multiple lines state 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.)

SECTION 2
DISPLAYS

Displays are sections of text to be set off from the body of the paper. Major quotes, 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.

2.1. Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commmands **.(q** and **.)q** to surround the quote. 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,...

2.2. 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
```

to produce:

Alternatives to avoid deadlock are:

- Lock in a specified order
- Detect deadlock and back out one process
- Lock all resources needed before proceeding

2.3. Keeps

A keep is a display of lines kept on a single page if possible. An example of using a keep might be a diagram. Keeps differ from lists in that lists can be broken over a page boundary whereas keeps cannot.

Blocks are the basic kind of keep. They begin with the request **.(b** and end with the request **.)b**. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. Where this is not appropriate, use the alternative, called floating keeps.

Floating keeps move relative to the text. Hence, 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 1.

```
1 "Example of a Floating Keep"
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1.  Example of a Floating Keep.
.hl
.)z
```

The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

2.4. Fancier Displays

Because keeps and lists are normally collected in nofill mode. They are good for tables and such. For a display in fill mode (for text), type **.(l F** (Throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This 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 normally 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 is desireable 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. 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 <u>not</u> keeps, and can 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 <u>inside</u> the keep requests.

SECTION 3
ANNOTATIONS

There are a number of requests that save text for later printing. Footnotes are printed at the bottom of the current page. Delayed text is 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 explicitly called.

3.1. 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
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]

---

[1] Like this.

[2] James R. Ware, The Best of Confucius, Halcyon House, 1950. Page 77.

It is important that the footnote appears inside the quote, to ensure that the footnote appears on the same page as the quote.

3.2. Delayed Text

Delayed text is similar to a footnote except that it is printed when explicitly called. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use ***#** on delayed text instead of ****** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

3.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, as 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 a row of dots.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request can 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.

The **.xp** request prints the index.

For example, the input:

---

*Such as an asterisk.

```
.(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 long index entry 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 ..........................................    9
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. ...    9
```

The **.(x** request can have a single character argument, specifying the "name" of the index; the normal index is **x**. Thus, several "indicies" can be maintained simultaneously (such as a list of tables, table of contents, etc.).

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.

SECTION 4
FANCIER FEATURES

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form **1.2.3** (such as used in this document), and multicolumn output.

4.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. To get left-justified block-style paragraphs use **.lp** instead of **.pp**, as demonstrated by the next paragraph.

To use paragraphs that have the body indented, and the first line exdented (opposite of indented) with a label. Enter 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 there are spaces in the label of a **.ip** request, 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, the label is not separated from the text, and the rest of the text is lined up at the old margin (and not with the first line of text). 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 can be done correctly by saying:

```
.ip longlabel 10
```

to make the paragraph indent 10 spaces for this paragraph only. If there are 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**. Refer to Section 7 for more information.

If **.ip** is used with no argument, 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
which 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 which 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.

4.2. 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.

To specify the section number to begin, place 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 which cause each section to be indented proportionally to the depth of the section. For example, if you enter:

```
.nr si N
```

each section will be 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 Ø.5i
```

After this, sections will be 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 will do a section heading, but will put no number on the section.

4.3. Parts of the Basic Paper

Some requests 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
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

The request **.th** sets up the environment of the NROFF processor to do a thesis, using the rules established at Berkeley. It defines the correct headers and footers (a page number in the upper right hand 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 for a thesis. Also, the header is moved to the foot of the page on the first page of a chapter. Although the **.+c** request was not designed to work only with the **.th** request, it is tuned for the format acceptable for a PhD thesis at Berkeley.

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; for example, **.+c** was used to generate page one of this document.

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, to begin 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 one in lower case Roman numbers. **.+c** can 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.

4.4. Two Column Output

To get two column output automatically, use 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**.

4.5. Defining Macros

A macro is a collection of requests and text which can 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 3 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.**

4.6. Annotations Inside Keeps

Sometimes a footnote or index entry is required inside a keep. For example, to maintain a "list of figures" enter:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which hopefully will give 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
```

This 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**).

SECTION 5
TROFF AND THE PHOTOTYPESETTER

With a little care, documents can be prepared that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

5.1. Fonts

A font is a style of type. Three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text 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. A single word can be set in some font by typing (for example):

```
i word
```

This sets word in italics but does not affect the surrounding text. In NROFF, italic and bold text are underlined.

When setting more than one word in whatever font, surround that word with double quote marks (`"') so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If a quote mark is to appear, quote the entire string (even if a single word), and use two quote marks where one is to appear. For example, to produce the text:

"Master Control"

in italics, type

```
.i """Master Control\|"""
```

The \| produces a very narrow space so that the "|" does not overlap the quote sign in TROFF.

5.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 1Ø points for

most text, 8 points for footnotes. To change the pointsize, 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.

WARNING

Changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

5.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (`"'). This is because it looks better to use grave and acute accents; for example, compare "quote" to ``quote''.

In order to make quotes compatible between the typesetter and terminals, use the sequences ***(lq** and ***(rq** to stand for the left and right quote 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". The material to be quoted must be surrounded with double quote marks if it is more than one word.

SECTION 6
-ME REFERENCE MANUAL

This section describes the features of the **-me** macro package for ZEUS. The reader should understand breaks, fonts, pointsizes, 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.

There are a number of macro parameters that can be adjusted. Fonts can 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 Ø is no font change; the font of the surrounding text is used instead. Notice that fonts Ø 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 Ø.Ø2 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form **$** x can be used in expressions but should not be changed. Macros of the form **$** x perform some function (as described) and can be redefined to change this function. This can 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 can define number registers, strings, and macros, provided that s/he uses single character upper case names or double character names consisting of letters and digits, with at least one upper case letter. In no case should special characters be used in user-defined names.

On daisy wheel type printers in twelve pitch, the **-rx1** flag can be stated to make lines default to one eighth inch (the normal spacing for a newline in twelve-pitch). This is normally too small for easy readability, so the default is to space one sixth inch.

This documentation was NROFF'ed on June 4, 1979 and applies to Version 1.1/20 of the -me macros.

6.1. 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 it is not possible to use any of the following requests: **.sc**, **.lo**, **.th**, or **.ac**. Also, the effects of changing parameters which will 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 to **\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.

.lp *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 non-indented 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.

.np A variant of .ip which numbers paragraphs. Numbering is reset after a **.lp**, **.pp**, or **.sh**. The current paragraph number is in **\n($p**.

6.2. Section Headings

Numbered sections are similiar to paragraphs except that a section number is automatically generated for each one. The section numbers are of the form **1.2.3.** 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($Ø**) 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** [1Øp].

The "name" of the section can be accessed via ***($n.** If **\n(si** is non-zero, 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** [Ø] is added to the section title (but not to the body of the section). The font is set to the paragraph font, so information can occur on the line with the section number and title. **.sh** insures enough room to print the section head plus the beginning of a paragraph (about 3 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 ("_") the section depth and numbering is reset, 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 starts 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.**

.Sp 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.

.$Ø T B N — This macro is called automatically after every call to **.$p**. It is normally undefined, but can be used to automatically put every section title into the table of contents or for some similiar function. T is the section title for the section title just printed, B is the section number, and N is the section depth.

.$1 - .$6 — Traps called just before printing that depth section. Can be defined to (for example) give variable spacing before sections. These macros are called from **.$p**, so in redefining that macro, this feature can be lost.

6.3. 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** [1Øp]. 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 macros **.m1**, **.m2**, **.m3**, and **.m4** are also supplied for compatibility with ROFF documents.

.he 'l'm'r' — Define three-part header, to be printed on 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 hardly every used. Must be followed by a **.bp** or the end of input.

.$h Called at every page to print the header. Can be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the **.he, .fo, .eh,** chapter-style title feature of **.+c.**

.$f Print footer; same comments apply as in **.$h.**

.$H A normally undefined macro which is called at the top of each page (after outputing 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.

6.4. Displays

All displays except centered blocks and block quotes are preceeded 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 will be 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 line-by-line basis. The list is set in font **\n(df** [Ø]. Must be matched by a **.)l**. This macro is similiar **.(b** except that no attempt is made to keep the display on one page.

.)l — End list.

.(q LIST — Begin major quote. These are single spaced, filled, moved in from the text on both sides by **\n(q** [4n], preceeded 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 BLOCK — 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 which should not be broken over a page.) If the block will not fit on the current page, a new page is begun, unless that would leave more than **\n(bt** [Ø] white space at the bottom of the text. If **\n(bt** is zero, 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(b** [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** [Ø].

.)b End block.

.(z m f KEEP 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 relative to the text changes. The floating keep is preceeded and followed by **\n(zs** [1v] space. Also, it defaults to mode **M.**

.)z End floating keep.

.(c CENTER 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 can be nested inside keeps.

.)c End centered block.

6.5. Annotations

.(d DELAYED TEXT Begin delayed text. Everything in the next keep is saved for output later with **.pd,** in a manner similar to footnotes.

.)d n DELAYED TEXT End delayed text. The delayed text number register **\n($d** and the associated string ***#** are incremented if ***#** has been referenced.

.pd DELAYED TEXT Print delayed text. Everything diverted via **.(d** is printed and truncated. This can used at the end of each chapter.

.(f FOOTNOTE 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 preceeded by **\n(fs** [Ø.2v] space, is indented **\n(fi** [3n] on the first line, and is indented **\n(fu** [Ø] from the right margin. Footnotes line up underneath two columned output. If the text of the footnote does not fit on one page it is carried over to the next page.

.)f n FOOTNOTE End footnote. The number register **\n($f** and the associated string ****** are incremented if they have been referenced.

.$s FOOTNOTE — The macro to output the footnote seperator. This macro can be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.

.(x x INDEX — Begin index entry. Index entries are saved in the index x [**x**] until called up with **.xp**. Each entry is preceeded by a **\n(xs** [Ø.2v] space. Each entry is "undented" by **\n(xu** [Ø.5i]; this register tells how far the page number extends into the right margin.

.)x P A INDEX — 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 INDEX — Print index x [**x**]. The index is formated in the font and size, in effect at the time it is printed, rather than at the time it is collected.

6.6. Columned Output

.2c +S N — Enter two-column mode. The column separation is set to +S [4n, Ø.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 **\n($c**. You can test register **\n($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.

6.7. 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.

.l 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. In NROFF, underlines.

.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 won't work correctly if W is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.q W X — Quote W and append X. In NROFF this 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 won't work correctly if W is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.bx W X — Sets W in a box, with X appended. Underlines in NROFF. It won't work correctly right if W is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

6.8. 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**.

.nl — Number lines in margin from one on each page.

.n2 N — Number lines from N, stop if N = Ø.

.sk — Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram which 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 will be output immediately if there is room, and will otherwise be output at the top of the next page. However, if N is greater than the amount of available space on an empty page, no space will ever be output.

6.9. Preprocessor Support

.EQ m T — Begin equation. The equation is centered if m is **C** or omitted, indented **\n(bi** [4n] if m is **I**, and left justified if m is **L**. T is a title printed on the right margin next to the equation. See Typesetting Mathematics - User's Guide by Brian W. Kernighan and Lorinda L. Cherry.

.EN c — End equation. If c is C the equation must be continued by immediately following with another **.EQ**, the text of which can be centered along with this one. Otherwise, the equation is printed, always on one page, with **\n(es** [Ø.5v in TROFF, 1v in NROFF] space above and below it.

.TS h — Table start. Tables are single spaced and kept on one page if possible. For a large table which will 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 Tbl - A Program to Format Tables by M. E. Lesk.

.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 requests such as **.sp** are intermixed with the text of the table. To float it (or if using requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z.**

6.10. Miscellaneous

.re — Reset tabs. Set to every Ø.5i in TROFF and every Ø.8i in NROFF.

.ba +N — Set the base indent to +N [Ø] (saved in **\n($i).** All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The **.sh** request performs a **.ba** request if **\n(si** [Ø] is not zero, and sets the base indent to **\n(si*\n($Ø.**

.xl N — Set the line length to N [6.Øi]. This differs from **.ll** because it only affects the current environment.

.ll +N — Set line length in all environments to N [6.Øi]. 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.

6.11. Standard Papers

.tp Begin title page. Spacing at the top of the page can occur, and headers and footers are supressed. Also, the page number is not incremented for this page.

.th Set thesis mode. This defines the modes acceptable for a doctoral dissertation at Berkeley. 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 which we are entering. The section type is defined by m. **C** means that we are entering the chapter portion of the paper, **A** means that we are entering the appendix portion of the paper, **P** means that the material following should be the preliminary portion (abstract, table of contents, etc.) portion of the paper, **AB** means that we are entering the abstract (numbered independently from 1 in Arabic numerals), and **B** means that we are entering the bibliographic portion at the end of the paper. 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. For the header to have the chapter number in it, use the string **\\\\n(ch.** For example, to number appendixes **A.1** etc., type **.++ RA '''\\\\n(ch.%'**. Each section (chapter, appendix, etc.) should be preceeded by the **.+c** request.

It should be mentioned that 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.

1.+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.

.$c** T — Print chapter number (from **\n(ch**) and T. This macro can be redefined to your liking. It is defined by default to be acceptable for a PhD thesis at Berkeley. This macro calls **$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 which will be 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 can later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro does work correctly in TROFF, since it sets the page length wider than the physical width of the phototypesetter roll.

6.12. 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 (`[').

***]** Unsuperscript. Inverse to *[. For example, to produce a superscript you might type **x*|2*]** which will produce x(2).

***<** Subscript. Defaults to `<' if half-carriage motion not possible.

***>** 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 June 4, 1979. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 4), ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, June is 6), and **\n(yr** (the last two digits of the current year).

***(lq** Left quote marks. Double quote in NROFF.

***(rq** Right quote.

***-** 3/4 em dash in TROFF; two hyphens in NROFF.

6.13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through -me. To reference these characters, 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*'	'a
Grave accent	*`	e*`	e'
Umlat	*:	u*:	u
Tilde	*~	n*~	n

Caret	*^	e*^	e
Cedilla	*,	c*,	c
Czech	*v	e*v	e
Circle	*o	A*o	A
There exists	*(qe		
For all	*(qa		

MM -- Memorandum Macros

based on an article originally written by:
D. W. Smith
J. R. Mashey
E. C Pariser (January 1980 Reissue)

10/14/83

Table of Contents

SECTION 1
INTRODUCTION

1.1. Description

Memorandum Macros, MM, is a general-purpose package of text formatting macros for use with the ZEUS text formatters nroff and troff. The purpose of MM is to provide a unified, consistent, and flexible tool for producing many types of documents. Although the ZEUS operating system provides macro packages for specialized formats, MM is a general-purpose macro package suitable for most documents.

The uses of MM range from single-page letters to documents of several hundred pages in length. Some of the uses of MM are to create:

- Letters
- Reports
- Technical Memoranda
- Release-Papers
- Manuals
- Books

1.2. Conventions

Each section explains a function of MM. In general, the earlier a section occurs, the more common its use. Some of the later sections can be completely ignored if the MM defaults are adequate. Likewise, each section progresses from normal-case to special-case fuctions. Try reading a section in detail until obtaining enough information for the desired format, then skim the rest of it. Some details may be of use to just a few people.

Numbers enclosed in curly brackets **({})** refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (•) at the end of the section heading.

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.

A reference of the form name(N) points to the command name in Section N of the ZEUS Reference Manual

This manual is produced by nroff; troff output would look somewhat different. When the output of the two formatters is different, nroff is described first, and troff follows in parentheses. For example:

The title is underlined (italicized).

means that the title is underlined in nroff and italicized in troff.

1.3. Document Structure

The input for formatting a document in MM can contain four major components, any of which can be omitted. If present, they must occur in the following order:

- Parameter-setting - This sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers, and many other elements of the document. The user can also add macros or redefine existing ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but performs the setup for the rest of the document.

- Beginning - This segment defines those items that occur only once, at the beginning of a document, e.g., title, author's name, date.

- Body - The body can 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 list items. The body may also contain various types of displays, tables, figures, references, and footnotes.

- Ending - The ending contains those items that occur only at the end of a document. Included here are signature(s) and list notations (e.g., "Copy To" lists). Certain macros can be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet.

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed differently for several document types, there is a uniform method of entering commands.

1.4. Definitions

The term formatter refers to either of the text-formatting programs nroff or troff.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly {3.10}, this document contains references to some of them. Full details are given in the NROFF User's Manual in the ZEUS Utilities Manual. For example, the request:

```
.sp
```

inserts a blank line in the output.

A macro is a group of requests shortened to one command. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. MM 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.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by requests and macros. A string can be given a value with the .ds (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for 1-character names) or "*(" (for 2-character names). For example, the string DT in MM normally contains the current date, so that the input line:

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

may result in the following output:

Today is January 22, 1980.

The current date can be replaced, e.g.:

```
.ds DT 01/01/79
```

or by invoking a macro designed for that purpose {6.7.1}.

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

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

See {14.1} regarding naming conventions for requests, macros, strings, and number registers.

SECTION 2
INVOKING THE MACROS

This section tells how to access MM, shows MM command lines appropriate for various output devices, and describes command-line flags for MM.

2.1. The mm Command

The mm(1) command can be used to print documents using nroff and MM; this command invokes nroff with the -cm flag {2.2}. It has options to specify preprocessing by tbl(1) and/or by neqn(1), and for postprocessing by various output filters. Any arguments or flags that are not recognized by mm(1), e.g. -rC3, are passed to nroff or to MM, as appropriate. The options, which can occur in any order but must appear before the file names, are:

-e	invokes neqn(1)
-t	invokes tbl(1)
-c	invokes col(1)
-E	invokes the 'e' option of nroff
-mm	(uncompacted macros) used instead of -cm
-12	uses 12-pitch mode (Be sure that the pitch switch on the terminal is set to 12.)
-T450	output is to a DASI450. This is the default terminal type (unless $TERM is set). It is also equivalent to -T1620.
-T450-12	output is to a DASI450 in 12-pitch mode
-T300	output is to a DASI300 terminal
-T300-12	output is to a DASI300 in 12-pitch mode
-T300S	output is to a DASI300S
-T300S-12	output is to a DASI300Srin 12-pitch mode
-T37	output is to a TELETYPE Model 37
-T382	output is to a DTC-382
-T4000A	output is to a Trendata 4000A
-TX	output is prepared for an EBCDIC lineprinter
-T2631	output is prepared for a HP2631 printer (where -T2631-e and -T2631-ch can 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 troff (see mmt(1)).

2.2. The -cm or -mm Flag

The MM package can also be invoked by including the -cm or -mm flag as an argument to the formatter. The -cm flag loads the pre-compacted version of the macros. The -mm flag reads and processes the file /usr/lib/tmac/tmac.m before any other files. This action defines the MM macros, sets default values for various parameters, and initializes the formatter for processing the files of input text.

2.3. Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4} and in the NROFF User's Manual

- Text without tables or equations:

```
     mm [options] filename ...
or   nroff [options] -mm filename ...
     mmt [options] filename ...
or   troff -mm [options]
```

- Text with tables:

```
     mm -t [options] filename ...
or   tbl filename ... | nroff [options] -cm
     mmt -t [options] filename ...
or   tbl filename ...| troff [options] -cm
```

- Text with equations:

```
     mm -e [options] filename ...
or   neqn filename ... | nroff [options] -cm
     mmt -e [options] filename ...
or   eqn filename ... | troff [options] -cm
```

- Text with both tables and equations:

```
     mm -t -e [options] filename ...
or   tbl filename ... | neqn | nroff [options] -cm
     mmt [options] -t -e filename ...
or   tbl filename ... | eqn | troff [options] -cm
```

When formatting a document with nroff, the output should normally be processed for a specific type of terminal, because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See {2.4} as well as 300(1), 450(1), col(1), and terminals(7) for further information.

- DASI450 in 10-pitch, 6 lines/inch mode, with .75 inch offset, and a line length of 6.0 inches (60 characters) where this is the default terminal type so no -T option is needed (unless $TERM is set to another value):

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

- DASI450 in 12-pitch, 6 lines/inch mode, with .75 inch offset, and a line length of 6.0 inches (72 characters):

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

  or, to increase the line length to 80 characters and decrease the offset to 3 characters:

```
        mm -12 -rW80 -rO3
        nroff -T450-12 -rW80 -rO3 -h -cm filename ...
```

Invoke tbl(1) and eqn(1) or neqn(1), if needed, as shown in the command line prototypes at the beginning of this section.

If two-column processing {12.4} is used with nroff, either the -c option must be specified to mm(1) (note that mm(1) uses col(1) automatically for many of the terminal types {2.1}) or the nroff output must be postprocessed by col(1). In the latter case, the -T37 terminal type must be specified to nroff, the -h option must not be specified, and the output of col(1) must be processed by the appropriate terminal filter (e.g., 450(1)); mm(1) with the -c option handles all this automatically.

2.4. Command Line Parameters

Number registers hold parameter values that control the various output styles. Many of these can be changed within the text files with .nr requests. In addition, some of these registers can be set from the command line itself; a useful feature for those parameters that should not be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register P, see below) must be set on the command line (or before the MM macro definitions are processed). Their meanings are:

-rAln n = 1 has the effect of invoking the .AF macro without an argument {6.7.2}. If n = 2, allows for usage of the Bell Logo, if available, on a printing device (currently available for the Xerox 9700 only).

-rBn defines the macros for the cover sheet and the table of contents. If n is 1, table of contents processing is enabled. If n is 2, then cover sheet processing will occur. If n is 3, both will occur. That is, B having a value greater than 0 defines the .TC {10.1} and/or .CS {10.2} macros. To have any effect, these macros must also be invoked.

-rCn n sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
n = 1 for OFFICIAL FILE COPY
n = 2 for DATE FILE COPY
n = 3 for DRAFT with single-spacing and default paragraph style
n = 4 for DRAFT with double-spacing and 10 space paragraph indent

-rD1 sets debug mode. This flag requests the formatter to attempt to continue processing even if MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 12.3}.

-rEn controls the font of the Subject/Date/From fields. If n is 0, these fields are bold (default for troff), and if n is 1, these fields are regular text (default for nroff).

-rL*k* sets the length of the physical page to k lines.* The default value is 66 lines per page. This parameter is used, for example, when directing output to a Versatec printer.

-rN*n* specifies the page numbering style. When n is Ø (default), all pages get the (prevailing) header {9.2}. When n is 1, the page header replaces the footer on page 1 only. When n is 2, the page header is omitted from page 1. When n is 3, section-page numbering {4.5} occurs (see .FD {8.3} and .RP {11.4} for footnote and reference numbering in sections). When n is 4, the default page header is suppressed; however, a user-specified header is not affected. When n is 5, section-page and section-figure numbering occurs.

n	Page 1	Pages 2 ff.
Ø	header	header
1	header replaces footer	header
2	no header	header
3	section-page as footer	
4	no header	no header unless PH defined

The contents of the prevailing header and footer do not depend on the value of the number register N; N only controls whether and where the header (and, for N = 3 or 5, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of N is irrelevant.

-rO*k* offsets output k spaces to the right.** It is helpful for adjusting output positioning on some terminals. If this register is not set on the command line, the default offset is .75 inches. NOTE: The register name is the capital letter "O", not the digit zero (Ø).

---

* For nroff, k is an unscaled number representing lines or character positions; for troff, k must be scaled.

** For nroff, these values are unscaled numbers representing lines or character positions. For troff, these values must be scaled.

-rP*n* specifies that the pages of the document are to be numbered starting with n. This register may also be set with a .nr request in the input text.

-rS*n* sets the point size and vertical spacing for the document. The default n is 10, i.e., 10-point type on 12-point leading (vertical spacing), giving 6 lines per inch {12.9}. This parameter applies to troff only.

-rT*n* provides register settings for certain devices. If n is 1, then the line length and page offset are set to 80 and 3, respectively. Setting n to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to nroff only.

-rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4}. This parameter applies to nroff only.

-rW*k* page width (i.e., line length and title length) is set to k.** This can be used to change the page width from the default value of 6.0 inches (60 characters in 10 pitch or 72 characters in 12 pitch).

2.5. Omission of -cm or -mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

** For nroff, k is an unscaled number representing lines or character positions; for troff, k must be scaled.

In this case, one must not use the -cm or -mm flag (nor the mm(1) or mmt(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized before the .so request, because their values are meaningful only if set before the 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 8Ø
.nr O 1Ø
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
   .
   .
   .
```

specifies in characters for nroff, a line length of 8Ø, a page offset of 1Ø; section-page numbering, and table of content processing.

3.2. Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Note that omitting an argument is not the same as supplying a null argument (for example, see the .MT macro in {6.6}). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes (").* 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 you must, use two grave accents (``) and/or two acute accents ('') instead. This restriction is necessary because many macro arguments are processed (interpreted) any number of times; for example, headings printed in the text can be (re)printed in the table of contents.

3.3. Unpaddable Spaces

When output lines are justified to give an even right margin, existing spaces in a line can have additional spaces appended to them. This can harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an unpaddable space. There are several ways to accomplish this.

First, one can type a backslash followed by a space ("\ "). This pair of characters directly generates an unpaddable space. Second, one can sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character can be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily recovered by inserting:

---

* A double quote (") is a single character that must not be confused with two apostrophes or acute accents (''), or with two grave accents (``).

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the .tr ~, but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

3.4. Hyphenation

The formatters do not perform hyphenation unless the user requests it. Hyphenation can be turned on in the body of the text by specifying:

```
.nr Hy 1
```

once at the beginning of the document. For hyphenation control within footnote text and across pages, see {8.3}.

If hyphenation is requested, the formatters will automatically hyphenate words where needed. However, the user can specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation-indicator, or can 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 point(s) inside a word. In any case, all occurrences of the hyphenation-indicator disappear on output.

The user can specify a different hyphenation-indicator:

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

```
.HC ^
```

Note that any word containing hyphens or dashes, also known as em dashes, will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, even if the formatter hyphenation function is turned off.

```
.af   .br   .ce   .de   .ds
.fi   .hw   .ls   .nf   .nr
.nx   .rm   .rr   .rs   .so
.sp   .ta   .ti   .tl   .tr
```

The .fp, .lg, and .ss requests are also sometimes useful for troff. Use of other requests without fully understanding their implications can lead to disaster.

SECTION 4
PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in {5}.

4.1. Paragraphs

```
.P [type]
one or more lines of text.
```

This macro is used to begin two kinds of paragraphs. In a left-justified paragraph, the first line begins at the left margin, and is indented five spaces with an indented paragraph (see below).

A document possesses a default paragraph style obtained by specifying .P before each paragraph that does not follow a heading {4.2}. The default style is controlled by the register Pt. The initial value of Pt is Ø, which always provides left-justified paragraphs. All paragraphs can be forced to be indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register Pi, whose default value is 5. To indent paragraphs by 1Ø spaces, insert:

```
.nr Pi 1Ø
```

at the beginning of the document. Of course, both the Pi and Pt register values must be greater than zero for any paragraphs to be indented.

The number register Ps controls the amount of spacing between paragraphs. By default, Ps is set to 1, yielding one blank space (1/2 a vertical space).

WARNING

Values that specify indentation must be unscaled and are treated as character positions, i.e., as a number of ens. In troff, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In nroff, an en is equal to the width of a character.

Regardless of the value of Pt, an individual paragraph can be forced to be left-justified or indented. ".P Ø" always forces left justification; ".P 1" always causes indentation by the amount specified by the register Pi. If .P occurs inside a list, the indent (if any) of the paragraph is added to the current list indent {5}.

Numbered paragraphs can be produced by setting the register Np to 1. This produces paragraphs numbered within first level headings, e.g., 1.Ø1, 1.Ø2, 1.Ø3, 2.Ø1, etc.

A different style of numbered paragraphs is obtained by using the .nP macro rather than the .P macro. This produces paragraphs that are numbered within second level headings and contain a double-line indent so the text of the second line is indented to align with the text of the first line. This causes the number to stand out.

```
.H 1 "FIRST HEADING"
.H 2 "Second Heading"
.nP
one or more lines of text
```

4.2. Numbered Headings

```
.H level [heading-text] [heading-suffix]
zero or more lines of text
```

The .H macro provides seven levels of numbered headings. Level 1 is the most major or highest; level 7 the lowest.

The heading-suffix is appended to the heading-text and can be used for footnote marks which should not appear with the heading text in the table of contents.

In this case, one must not use the -cm or -mm flag (nor the mm(1) or mmt(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized before the .so request, because their values are meaningful only if set before the 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 in characters for nroff, a line length of 80, a page offset of 10; section-page numbering, and table of content processing.

SECTION 3
FORMATTING CONCEPTS

3.1. Basic Terms

The most common function of the formatters is to fill output lines from one or more input lines. The output lines can be justified so that both the left and right margins are aligned. As the lines fill the page they can be hyphenated {3.4} as needed. It is possible to turn any of these modes on and off (see .SA {12.2}, Hy {3.4}, and the formatter .nf and .fi requests). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) introduce a break in the output line currently printing which causes the subsequent text to begin a new output line. This printing of a partially filled output line is known as a break. A few formatter requests and most of the MM macros cause a break.

While formatter requests can be used with MM, one must fully understand the consequences and side-effects that each request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

-- it is much easier to control (and change at any later point in time) the overall style of the document.

-- complicated functions (such as footnotes or tables of contents) can be obtained with ease.

-- the user is insulated from the peculiarities of the formatter language.

It is a good rule to use formatter requests only when absolutely necessary {3.10}.

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full sentence must begin on a new line.

3.2. Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Note that omitting an argument is not the same as supplying a null argument (for example, see the .MT macro in {6.6}). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes (").* 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 you must, use two grave accents (``) and/or two acute accents ('') instead. This restriction is necessary because many macro arguments are processed (interpreted) any number of times; for example, headings printed in the text can be (re)printed in the table of contents.

3.3. Unpaddable Spaces

When output lines are justified to give an even right margin, existing spaces in a line can have additional spaces appended to them. This can harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an unpaddable space. There are several ways to accomplish this.

First, one can type a backslash followed by a space ("\ "). This pair of characters directly generates an unpaddable space. Second, one can sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character can be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily recovered by inserting:

---

* A double quote (") is a single character that must not be confused with two apostrophes or acute accents (''), or with two grave accents (``).

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the .tr ~, but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

3.4. Hyphenation

The formatters do not perform hyphenation unless the user requests it. Hyphenation can be turned on in the body of the text by specifying:

```
.nr Hy 1
```

once at the beginning of the document. For hyphenation control within footnote text and across pages, see {8.3}.

If hyphenation is requested, the formatters will automatically hyphenate words where needed. However, the user can specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation-indicator, or can 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 point(s) inside a word. In any case, all occurrences of the hyphenation-indicator disappear on output.

The user can specify a different hyphenation-indicator:

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

```
.HC ^
```

Note that any word containing hyphens or dashes, also known as em dashes, will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, even if the formatter hyphenation function is turned off.

Using the .hw request, the user can compose a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word printout, one can specify:

```
.hw print-out
```

3.5. Tabs

The macros .MT {6.6}, .TC {10.1}, and .CS {10.2} use the formatter .ta request to set tab stops, and then restore the default values of tab settings.* Thus, setting tabs to other than the default value is the user's responsibility.

Note that a tab character is always 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 mode {3.1}.

Also note that tbl(1) {7.3} changes tab stops, but does not restore the default tab settings.

3.6. Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers {9}, headings {4}, and list marks {5}. Users who include BEL characters in their input text (especially in arguments to macros) will receive mangled output.

3.7. Bullets

A bullet (•) is often obtained on a typewriter terminal by using an o overstruck by a +. For compatibility with troff, a bullet string is provided by MM. Rather than overstriking, use the sequence:

```
\*(BU
```

wherever a bullet is desired. Note that the bullet list (.BL) macros {5.3.3.2} use this string to automatically generate the bullets for the list items.

---

* Every eight characters in nroff; every 1/2 inch in troff.

3.8. Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while nroff does not. Those who intend to use nroff only, can use the minus sign ("-") for all three.

Those who wish mainly to use troff should follow the escape conventions of the NROFF User's Manual.

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

Dash — Type "*(EM" for each text dash for both nroff and troff. This string generates an em dash (-) in troff and generates -- in nroff. Note that the dash list (.DL) macros {5.3.3.3} automatically generates the em dashes for the list items.

Hyphen — Type "-" and use as is for both formatters. Nroff will print it as is, and troff will print a true hyphen.

Minus — Type "\-" for a true minus sign, regardless of formatter. Nroff will effectively ignore the \, while troff will print a true minus sign.

3.9. Trademark String

A trademark string "*(Tm" is now available with MM. This places the letters TM one-half line above the text that it follows. For example, the string

```
System 8000  \*(TM ZEUS Reference Manual
```

places the TM mark after System 8000.

3.10. Use of Formatter Requests

Most formatter requests should not be used with MM. MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than the basic formatter requests {3.1}. However, some formatter requests are useful with MM, namely:

```
.af   .br   .ce   .de   .ds
.fi   .hw   .ls   .nf   .nr
.nx   .rm   .rr   .rs   .so
.sp   .ta   .ti   .tl   .tr
```

The .fp, .lg, and .ss requests are also sometimes useful for troff. Use of other requests without fully understanding their implications can lead to disaster.

SECTION 4
PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in {5}.

4.1. Paragraphs

```
.P [type]
one or more lines of text.
```

This macro is used to begin two kinds of paragraphs. In a left-justified paragraph, the first line begins at the left margin, and is indented five spaces with an indented paragraph (see below).

A document possesses a default paragraph style obtained by specifying .P before each paragraph that does not follow a heading {4.2}. The default style is controlled by the register Pt. The initial value of Pt is Ø, which always provides left-justified paragraphs. All paragraphs can be forced to be indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register Pi, whose default value is 5. To indent paragraphs by 1Ø spaces, insert:

```
.nr Pi 1Ø
```

at the beginning of the document. Of course, both the Pi and Pt register values must be greater than zero for any paragraphs to be indented.

The number register Ps controls the amount of spacing between paragraphs. By default, Ps is set to 1, yielding one blank space (1/2 a vertical space).

WARNING

Values that specify indentation must be unscaled and are treated as character positions, i.e., as a number of ens. In troff, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In nroff, an en is equal to the width of a character.

Regardless of the value of Pt, an individual paragraph can be forced to be left-justified or indented. ".P Ø" always forces left justification; ".P 1" always causes indentation by the amount specified by the register Pi. If .P occurs inside a list, the indent (if any) of the paragraph is added to the current list indent {5}.

Numbered paragraphs can be produced by setting the register Np to 1. This produces paragraphs numbered within first level headings, e.g., 1.Ø1, 1.Ø2, 1.Ø3, 2.Ø1, etc.

A different style of numbered paragraphs is obtained by using the .nP macro rather than the .P macro. This produces paragraphs that are numbered within second level headings and contain a double-line indent so the text of the second line is indented to align with the text of the first line. This causes the number to stand out.

```
.H 1 "FIRST HEADING"
.H 2 "Second Heading"
.nP
one or more lines of text
```

4.2. Numbered Headings

```
.H level [heading-text] [heading-suffix]
zero or more lines of text
```

The .H macro provides seven levels of numbered headings. Level 1 is the most major or highest; level 7 the lowest.

The heading-suffix is appended to the heading-text and can be used for footnote marks which should not appear with the heading text in the table of contents.

WARNING

There is no need for a .P macro after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro. In fact, if a .P follows a .H, the .P is ignored. {4.2.2.2}.

4.2.1. Normal Appearance

The effect of .H 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 (1/2 a vertical space).

.H 1 heading-text	gives a bold heading followed by a single blank line (1/2 a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.
.H 2 heading-text	yields a bold heading followed by a single blank line (1/2 a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.
.H n heading-text	for 3<n<7, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are run-in headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

4.2.2. Altering Appearance of Headings

Users satisfied with the default appearance of headings can skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection

A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (1/2 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 can be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents can be made more manageable if each section starts on a new page. Setting Ej to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to Ej.

4.2.2.2 Spacing After Headings

Three registers control the appearance of text immediately following a .H call. They are Hb (heading break level), Hs (heading space level), and Hi (post-heading indent).

If the heading level is less than or equal to Hb, a break {3.1} occurs after the heading. If the heading level is less than or equal to Hs, a blank line (1/2 a vertical space) is inserted after the heading. Defaults for Hb and Hs are 2. If a heading level is greater than Hb and also greater than Hs, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any stand-alone heading (a heading not run into the following text), the alignment of the next line of output is controlled by the register Hi. If Hi is Ø, text is left-justified. If Hi is 1 (the default value), the text is indented according to the paragraph type as specified by the register Pt {4.1}. Finally, 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.

For example, to cause a blank line (1/2 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 top of the document:

```
.nr Hs  3
.nr Hb  7
.nr Hi  Ø
```

4.2.2.3 Centered Headings

The register Hc can be used to obtain centered headings. A heading is centered if its level is less than or equal to Hc, and if it is also stand-alone {4.2.2.2}. Hc is Ø initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings

Control by Level

Any heading that is underlined by nroff is made italic by troff. The string HF (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for HF are:

Formatter	HF Code 1	2	3	Default HF
nroff	no underline	underline	underline	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined in nroff and italic in troff. Reset HF as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following results in five bold levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

Nroff Underlining Style

Nroff can underline in two ways. The normal style (.ul request) is to underline only letters and digits. The continuous style (.cu request) underlines all characters, including spaces. By default, MM attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined, but is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the -rUl flag when invoking nroff {2.4}.

Heading Point Sizes

The user can also specify the desired point size for each heading level with the HP string (for use with troff 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 bold stand-alone headings are printed 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. Note that the specified values can also be relative point size changes, e.g.:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then those sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then the point sizes for the headings will be relative to the point size of the body, even if the latter is changed.

Omitted or zero values imply that the default point size will be used for the corresponding heading level.

WARNING

Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (with .HX and/or .HZ) can cause overprinting.

4.2.2.5 Marking Styles - Numerals and Concatenation

```
.HM [arg1] ... [arg7]
```

The registers H1 through H7 are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The .HM macro (Heading Mark) overrides this choice, 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 values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

Value	Interpretation
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register Ht (heading-mark type) to 1.

For example, a commonly-used outline style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

4.3. Unnumbered Headings

```
.HU heading-text
```

.HU is a special case of .H; it is handled in the same way as .H, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to

exist at the level given by register Hu, whose initial value is 2. Thus, in the normal case, the only difference between:

```
.HU heading-text
```

and

```
.H 2 heading-text
```

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of Hu should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up Appendices and other sections that do not fit well into the numbering scheme of the main body of a document {14.2.1}.

4.4. Headings and the 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 three things:

- specifying in the register C1 what level headings are to be saved;
- invoking the .TC macro {10.1} at the end of the document;
- and specifying -rBn {2.4} on the command line.

Any heading whose level is less than or equal to the value of the register C1 (Contents level) is saved and later displayed in the table of contents. The default value for C1 is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" diagnostic {Appendix E} will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5. First-Level Headings and the Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it is desirable to use page numbering of the form section-page, where section is the number of the current first-level heading. This page numbering style can be achieved by specifying the -rN3 or -rN5 flag on the command line {9.9}. As a side effect, this also has the effect of setting Ej to 1, i.e., each section begins on a new page. In this style, the page number is printed at the bottom of the page, so that the correct section number is printed.

4.6. User Exit Macros ♦

WARNING

This section is intended only for users who are accustomed to writing formatter macros.

```
.HX dlevel rlevel heading-text
.HY dlevel rlevel heading-text
.HZ dlevel rlevel heading-text
```

The .HX, .HY, and .HZ macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. MM does not define .HX, .HY, and .HZ; they are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. After .HX is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in .HX (such as .ti for temporary indent), to be lost. After the size calculation, .HY is invoked so that the user can respecify these features. All the default actions occur if these macros are not defined. If the .HX, .HY, or .HZ are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (dlevel) and the real level (rlevel) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro {4.3}, dlevel is equal to the contents of register Hu, and rlevel is Ø. In both cases, heading-text is the text of the original invocation.

By the time .H calls .HX, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank line(s) (vertical space) to precede the heading {4.2.2.1}, and accumulated the heading mark, i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

string }Ø — If rlevel is non-zero, this string contains the heading mark. Two unpaddable spaces (to separate the mark from the heading) have been appended to this string. If rlevel is Ø, this string is null.

register ;Ø — This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of Ø 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 (1/2 a vertical space) is to follow the heading.

string }2 — If register ;Ø is Ø, this string contains two unpaddable spaces that will be used to separate the (run-in) heading from the following text. If register ;Ø is non-zero, 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 dlevel equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;Ø 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 can alter the values of }Ø, }2, and ;3 within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format n. to n.Ø:

```
.if \\$1=1 .ds }Ø \\n(H1.Ø\(\(
```

(stands for a space

Separate run-in heading from the text with a period and two unpaddable spaces:

```
.if \\n(;Ø=Ø .ds }2 .\(\(

Assure that at least 15 lines are left on the page
before printing a first-level heading:
.if \\$1=1 .nr ;3 15-\\n(;Ø

Add 3 additional blank lines before each
first-level heading:
.if \\$1=1 .sp 3

Make varying indent for outline style:
.in \\$1*2-2

Indent level 3 run-in headings by 5 spaces:
.if \\$1=3 .ti 5n
```

If temporary string or macro names are used within .HX, care must be taken in the choice of their names {14.1}.

.HY is called after the .ne is issued. Certain features requested in .HX must be repeated. For example:

```
.de HY
.if \\$1=3 .ti 5n
.
```

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections can correspond to chapters of a book, and the user may want to change a page header or footer. For example:

```
.de HZ
.if $1=1 .PF Section $3
.
```

4.7. Hints for Large Documents

A large document is often organized for convenience into one file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers Ø1 through 2Ø rather than 1 through 9 and 1Ø through 2Ø.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register Hl to 1 less than the number of the section just before the corresponding .H 1 call. For example, at the beginning of Section 5, insert:

```
.nr Hl 4
```

WARNING

This is a dangerous practice: it defeats the automatic (re)numbering of sections when sections are added or deleted. Remove such lines as soon as possible.

SECTION 5
LISTS

This section describes many different kinds of lists: automatically-numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, e.g., with terms or phrases to be defined.

5.1. Basic Approach

In order to avoid repetitive typing of arguments to describe the appearance of items in a list, MM provides a convenient way to specify lists. All lists are composed of the following parts:

- A list-initialization macro that controls the appearance of the list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more List Item (.LI) macros, each followed by the actual text of the corresponding list item.
- The List End (.LE) macro that terminates the list and restores the previous indentation.

Lists can be nested up to five 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 that list. In addition, by building on the existing structure, it is possible to create customized sets of list macros with relatively little effort {5.4, Appendix A}.

5.2. Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls {5.3.3} contained therein are examples of the list-initialization macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a "plus" as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized 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 appears at the left margin.
```

Output:

A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

    1. This is a numbered 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. 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 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 appears at the left margin.

5.3. Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1. List Item

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The .LI macro is used with all lists. It normally causes the output of a single blank line (1/2 a vertical space) before its item, although this can be suppressed. If no arguments are given, it labels its item with the current mark, which is specified by the most recent list-initialization macro. If a single argument is given to .LI, 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
But this uses plus as prefix to the bullet.
.LE
```

yields:

• This is a simple bullet item.

+ This replaces the bullet with a plus.

+ ⊕ But this uses plus as prefix to the bullet.

WARNING

The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

If the current mark (in the current list) is a null string, and the first argument of .LI is omitted or null, the resulting effect is a hanging indent, i.e., the first line of the following text is outdented, starting at the same place where the mark would have started {5.3.3.6}.

5.3.2. List End

.LE [1]

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the .LE outputs a blank line (1/2 a vertical space). This option should generally be used only when the .LE is followed by running text, but not when followed by a macro that produces blank lines of its own, such as .P, .H, or .LI.

.H and .HU automatically clear all list information, so one can legally omit the .LE(s) that would normally occur just before either of these macros. Such a practice is not recommended, however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3. List Initialization Macros

The following are the various list-initialization macros. They are actually implemented as calls to the more basic .LB macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists

.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 Li, (initially 6) (5)* spaces from the indent in force when the .AL is called, thus leaving room for a space, two digits, a period, and two spaces before the text.

Spacing at the beginning of the list and between the items can be suppressed by setting the Ls (List space) register. Ls is set to the innermost list level for which spacing is done. For example:

```
.nr Ls  Ø
```

specifies that no spacing will occur around any list items. The default value for Ls is 6 (which is the maximum list nesting level).

The type argument can be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i {4.2.2.5}.** If type is omitted or null, then 1 is assumed. If text-indent is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of Li for this list only. If text-indent is null, then the value of Li will be used.

If the third argument is given, a blank line (1/2 a vertical space) will not separate the items in the list. A blank line (1/2 a vertical space) will occur before the first item, however.

5.3.3.2 **Bullet List**

```
.BL [text-indent] [1]
```

.BL begins a bullet list, in which each item is marked by a bullet (•) followed by one space. If text-indent is non-null, it overrides the default indentation, the amount of paragraph indentation as given in the register Pi {4.1}.***

---

* Values that specify indentation must be unscaled and are treated as character positions, i.e., as the number of ens.

** Note that the ØØØ1 format is not permitted.

*** So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

If a second argument is specified, no blank lines will separate the items in the list.

5.3.3.3 Dash List

.DL [text-indent] [1]

.DL is identical to .BL, except that a dash is used instead of a bullet.

5.3.3.4 Marked List

.ML mark [text-indent] [1]

.ML is much like .BL and .DL, but expects the user to specify an arbitrary mark, which can consist of more than a single character. Text is indented text-indent spaces if the second argument is not null; otherwise, the text is indented one more space than the width of mark. If the third argument is specified, no blank lines will separate the items in the list.

WARNING

The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

5.3.3.5 Reference List

.RL [text-indent] [1]

A .RL call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). Text-indent can be supplied, as for .AL. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list.

5.3.3.6 Variable-Item List

.VL text-indent [mark-indent] [1]

When a list begins with a .VL, 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. Mark-indent gives the number of spaces from the current indent to the beginning of the mark, and it defaults to Ø if omitted or null. Text-indent gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of .VL usage:

```
.tr ~
.VL 2Ø 2
.LI mark~1
Here is a description of mark 1;
mark 1 of the .LI line contains a tilde translated to an
unpaddable space in order to avoid extra spaces between
mark and 1 {3.3}.
.LI second~mark
This is the second mark, also using a tilde translated to
an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space
separates the mark from the text.

.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1 Here is a description of mark 1; mark 1 of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between mark and 1 {3.3}.

second mark This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent: This item shows the effect of a long mark; one space separates the mark from the text.

This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a hanging indent would have been produced. A hanging indent is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

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.

WARNING

The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

5.4. List-Begin Macro and Customized Lists ●

```
.LB text-indent mark-indent pad type [mark]
 [LI-space] [LB-space]
```

The list-initialization macros described above suffice for almost all cases. However, if necessary, one can obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros. Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register Li for automatic lists and from the register Pi for bullet and dash lists.

The combination of mark-indent and pad determines the placement of the mark. The mark is placed within an area (called mark area) that starts mark-indent spaces to the right of the current indent, and ends where the text begins (i.e., ends text-indent spaces to the right of the current

indent).* Within the mark area, the mark is left-justified if pad is Ø. If pad is greater than Ø, say n, then n blanks are appended to the mark; the mark-indent value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively right-justified pad spaces immediately to the left of the text.

Type and mark interact to control the type of marking used. If type is Ø, simple marking is performed using the mark character(s) found in the mark argument. If type is greater than Ø, automatic numbering or alphabetizing is done, and mark is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.3.3.1}. That is:

Type	Mark	Result
Ø	omitted	hanging indent
Ø	string	string is the mark
>Ø	omitted	arabic numbering
<Ø	one of: 1,A,a,I,i	automatic numbering or alphabetic sequencing

Each non-zero value of type from 1 to 6 selects a different way of displaying the items. The following table shows the output appearance for each value of type:

Type	Appearance
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where x is the generated number or letter.

WARNING

The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

* The mark indent argument is typically Ø.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each .LI macro in the list. If omitted, LI-space defaults to 1; the value Ø can be used to obtain compact lists. If LI-space is greater than Ø, the .LI macro issues a .ne request for two lines just before printing the mark.

LB-space, the number of blank lines (1/2 a vertical space) to be output by .LB itself, defaults to Ø if omitted.

There are three reasonable combinations of LI-space and LB-space. The normal case is to set LI-space to 1 and LB-space to Ø, yielding one blank line before each item in the list; such a list is usually terminated with a .LE 1 to end the list with a blank line. In the second case, for a more compact list, set LI-space to Ø and LB-space to 1, and, again, use .LE 1 at the end of the list. The result is a list with one blank line before and after it. If you set both LI-space and LB-space to Ø, and use .LE to end the list, a list without any blank lines will result.

Appendix A illustrates how the user can build upon the supplied list macros to obtain other kinds of lists.

SECTION 6
MEMORANDUM AND RELEASE-PAPER STYLES

MM can be used to prepare memoranda and release-papers which require special formats for the first page and the cover sheet. The information needed for the memorandum or release-paper (title, author, date, case numbers, etc.) is entered in the same way for both styles; an argument to one macro indicates which style is be used. The following sections describe the macros used to provide this data. The required order is shown in {6.9}.

If neither the memorandum nor release-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header {9} followed by the body of the document.

6.1. Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

The arguments for the .TL macro are the charging case number(s) and filing case number(s).* The title of the memorandum or paper follows the .TL macro and is processed in fill mode {3.1}. Multiple charging case numbers are entered as sub-arguments by separating each from the previous with a comma and a space, and enclosing the entire argument within double quotes. Multiple filing case numbers are entered similarly. For example:

```
.TL "12345, 67890" 987654321
On the construction of a table
of all even prime numbers
```

The .br request can break the title into several lines.

When output, the title appears after the word "Subject" in the memorandum style. In the release-paper style, the title is centered and bold.

---

* The charging case is the case number to which time was charged for the development of the project described in the memorandum. The filing case is a number under which the memorandum is to be filed.

If only a charging case number, or a filing case number is given, then it will be separated from the title in the memorandum style by a dash and will appear on the same line. If both cases are given, and the numbers are the same, then "Charging and Filing Case" followed by the number will appear on a line following the title. If the cases are different then separate lines for "Charging Case" and "File Case" will appear following the title.

6.2. Author(s)

```
.AU name [initials] [loc] [dept] [ext] [room] [arg]
[arg] [arg]
```

Information that describes an author is contained in the .AU macro arguments. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given. A separate .AU macro is required for each author.

The .AT macro is used to specify the author's title. Up to nine arguments can be given. Each will appear in the Signature Block for memorandum style {6.11.1} on a separate line following the signer's name. The .AT must immediately follow the .AU for the given author. For example:

```
.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234
.AT Director "Materials Research Laboratory"
```

In the "From" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension number, and room number can be suppressed on the first page of a memorandum by setting the register Au to 0; the default value for Au is 1. Arguments 7 through 9 of the .AU macro, if present, will follow this normal author information in the "From" portion, each on a separate line. Certain organizations have their own numbering schemes for memoranda, engineer's notes, etc. These numbers are printed after the author's name. This can be done by providing more than six arguments to the .AU macro, e.g.:

```
.AU "S. P. Lename" SPL IH 9988 7766 5H-444 3322.11AB
```

The name, initials, location, and department are also included in the Signature Block {6.11.1}. The author information in the "From" portion, as well as the names and initials in the Signature Block appear in the same order as the .AU macros.

The names of the authors in the release-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For authors in different locations, see {6.8}.

6.3. TM Number(s)

```
.TM [number] . . .
```

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers can be specified. Example:

```
.TM 7654321 77777777
```

This macro call is ignored in the release-paper and external-letter styles {6.6}.

6.4. Abstract

```
.AS [arg] [indent]
text of the abstract
.AE
```

Three styles of cover sheet are available: Technical Memorandum, Memorandum for File, and release-paper. On the cover sheet, the text of the abstract follows the author information and is preceded by the centered and underlined (italic) word "ABSTRACT."

The .AS (Abstract Start) and .AE (Abstract End) macros define the abstract (which is optional). When producing the Memorandum for File, no cover sheet will be produced unless an abstract is given.

Combining the first argument to .AS and the .CS macro will produce the cover sheet. If the first argument is 2, a Memorandum for File cover sheet is generated automatically. Any other value for the first argument causes the text of the abstract to be saved until the .CS macro is invoked and then the appropriate cover sheet (either Technical Memorandum or release-paper, depending on the .MT type) is generated. Thus, .CS is not needed for Memorandum for File cover sheets.

Notations {6.11.2}, such as a "Copy To" list, are allowed on Memorandum for File cover sheets. The .NS and .NE macros are given following the .AS 2 and .AE.

The abstract is printed with the standard text margins. An indentation for both margins can be specified as the second argument for .AS.* Note that headings {4.2, 4.3} and displays {7} are not permitted within an abstract.

6.5. Other Keywords

.OK [keyword] ...

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases can be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6. Memorandum Types

.MT [type] [addressee]

The .MT macro controls the format of the top part of the first page of a memorandum or a release-paper, as well as the format of the cover sheet. Legal codes for type and the corresponding values are:

---

* Values that specify indentation must be unscaled and are treated as character positions, i.e., as the number of ens.

Code	Value
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER's NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Release-Paper style
.MT 5	External-Letter style
.MT "string"	string

If type indicates a memorandum style, then value will be printed after the last line of author information. If type is longer than one character, then the string, itself, will be printed. For example:

```
.MT "Technical Note #5"
```

A simple letter is produced by calling .MT with a null (but not omitted!) or zero argument.

The second argument to .MT is used to give the name of the addressee of a letter. The name and page number will be used to replace the ordinary page header on the second and following pages of the letter. For example,

```
.MT 1 "John Jones"
```

produces

```
John Jones - 2
```

This second argument type can not be used if the first argument is 4 (i.e., for the release-paper style) as explained in {6.8}.

In the external-letter style (.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, providing the author's company logotype and address.

6.7. Date and Format Changes

Dates and formats are changed with the following macros.

6.7.1. Changing the Date

By default, the current date appears in the "Date" part of a memorandum. This can be overridden by using:

```
.ND new-date
```

The .ND macro alters the value of the string DT, which is initially set to the current date.

6.7.2. Alternate First-Page Format

One can specify that the words "Subject," "Date," and "From" (in the memorandum style) be omitted and that an alternate company name be used:

```
.AF [company-name]
```

If an argument is given, it replaces "Bell Laboratories," without affecting the other headings. If the argument is null "Bell Laboratories" is suppressed; and extra blank lines are inserted to allow room for stamping the document with the company logo stamp. .AF with no argument suppresses "Bell Laboratories" and the "Subject/Date/From" headings, allowing output on preprinted stationery. The use of .AF with no arguments is equivalent to the use of -rA1 {2.4} except the latter must be used if it is necessary to change the line length and/or page offset (which default to 5.8i and li, respectively, for preprinted forms). The command line options -r0k and -rWk are not effective with .AF.

The only .AF option appropriate for troff is to specify an argument to replace "Bell Laboratories" with another name.

6.8. Release-Paper Style

The release-paper style is obtained by specifying:

```
.MT 4 [1]
```

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF {6.7.2} specifies a different company). If the optional second argument to .MT 4 is given, then the name of each author is followed by the respective company name and location.

Information necessary for the memorandum style but not for the release-paper style is ignored.

If the release-paper style is utilized, most BTL location codes* are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, following the .MT macro, the user can re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the release-paper style is specified.

Authors from non-BTL locations can include their affiliations in the release-paper style by specifying the appropriate .AF and defining a string (with a 2 character name such as XX) for the address before each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.ds XX "22 Maple Avenue, Sometown Ø9999"
.AU "F. Swatter"
.AF "Bell Laboratories"
.AU "Sam P. Lename" " " CB
.MT 4 1
```

6.9. Order of Invocation of 'Beginning' Macros

The macros described in {6.1-6.7}, 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] [arg]
.AT [title] . . .
.TM [number] . . .
.AS [arg] [indent]
one or more lines of text
.AE
.NS [arg]
one or more lines of text
.NE
.OK [keyword] . . .
.MT [type] [addressee]
```

* The complete list is: AK, CP, CH, CB, DR, HO, HOH, HP, IN, IH, MV, MH, PY, RR, RD, WB, WV, and WH.

The only required macros for a memorandum or a release-paper are .TL, .AU, and .MT; all the others (and their associated input lines) can be omitted if the features they provide are not needed. Once .MT has been invoked, none of the above macros can be re-invoked because they are removed from the table of defined macros to save space.

6.10. Example

The input text for a manual title page could begin as follows:

```
.TL
MM-Memorandum Macros
.AU "D. W. Smith" DWS PY . . .
.AU "J. R. Mashey" JRM  WH . . .
.AU "E. C. Pariser (January 1980 Revision)" ECP PY . . .
.AU "N. W. Smith (April 1980 Revision)" NWS PY . . .
.MT 4
```

6.11. Macros for the End of a Memorandum

At the end of a memorandum (but not of a release-paper), the signatures of the authors and a list of notations can be requested. The following macros and their input are ignored if the release-paper style is selected.

6.11.1. Signature Block

```
.FC [closing]
.SG [arg] [1]
```

.FC prints "Yours very truly", as a formal closing. It must be given before the .SG which prints the signer's name. A different closing can be specified as an argument to .FC.

.SG prints the author name(s) after the formal closing (or the last line of text). Each name begins at the center of the page. Three blank lines are left above each name for the actual signature. If no argument is given, the line of reference data* will not appear following the last line.

A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null

* The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens.

argument to print reference data without the typist's initials or the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first author, rather than on the line that has the name of the last author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

6.11.2. 'Copy To' and Other Notations

```
.NS [arg]
zero or more lines of the notation
.NE
```

After the signature and reference data, many types of notations can follow, such as a list of attachments or "Copy To" lists. The .NS macro provides the proper spacing and breaking of notations across pages, if necessary.

The codes for arg and the corresponding notations are:

Code	Notations
.NS ""	Copy To
.NS 0	Copy To
.NS	Copy To
.NS 1	Copy (with att.) To
.NS 2	Copy (without att.) To
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter To
.NS 9	Memorandum To
.NS "string"	Copy (string) To

If arg consists of more than one character, it is placed within parentheses between the words Copy and To. For example:

.NS "with att. 1 only"

will generate "Copy (with att. 1 only) To" as the notation. More than one notation can be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as:

```
Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names
Copy (with att.) To
J. J. Jones

Copy (without att.) To
S. P. Lename
G. H. Hurtz
```

The .NS and .NE macros can also be used at the beginning, following .AS 2 and .AE, to place the notation list on the Memorandum for File cover sheet {6.4}. If notations are given at the beginning without .AS 2, they will be saved and output at the end of the document.

6.12. Approval Signature Line

```
.AV approver's name
```

The .AV macro can be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example,

```
.AV "Jane Doe"
```

produces:

```
APPROVED:


_____________________ ______
Jane Doe              Date
```

6.13. Forcing a One-Page Letter

To lengthen a single-page letter to accommodate the signature or notations that would otherwise be on the next page, use the -rLn option, e.g. -rL90. This changes the format command to 90 lines. This will only work for a single-page letter or memo.

SECTION 7
DISPLAYS

Displays are blocks of text that are to be kept together, not split across pages. MM 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 can result in extra white space at the bottom of the page if the display is too big to fit there. In the floating style, the display floats through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that follows a floating display can precede it in the output text. A queue of floating displays is maintained so that their relative order 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.

Displays and footnotes {8} can never be nested, in any combination whatsoever. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) {4.2, 4.3} can occur within displays or footnotes.

7.1. Static Displays

```
.DS [format] [fill] [rindent]
One or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will not indent them from the prevailing left margin indentation or from the right margin. The indent argument is the number of characters** that the line length should be decreased, i.e., an indentation from the right margin.

---

* Displays are processed in an environment that is different from that of the body of the text. (See the .ev request in the Nroff User's Manual.)

** This number must be unscaled in nroff and is treated as ens. It can be scaled in troff or else defaults to ems.

The format argument to .DS is an integer or letter used to control the left margin indentation and centering with the following meanings:

Code	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

The fill argument is also an integer or letter and can have the following meanings:

Code	Meaning
"	no-fill mode
0 or N	no-fill mode
1 or F	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register Si, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the Pi register {4.1}. Even though their initial values are the same, these two registers are independent of one another.

The display format value 3 (CB) centers the entire display as a block (as opposed to .DS 2 and .DF 2, which center each line individually). That is, all the collected lines are left-justified, and then the display is centered, based on the width of the longest line. This format must be used in order for the eqn(1)/neqn(1) "mark" and "lineup" feature to work with centered equations (see {7.4}).

By default, a blank line (1/2 a vertical space) is placed before and after static and floating displays. These blank lines before and after static displays can be inhibited by setting the register Ds to 0.

The following example shows the usage of all three arguments for displays. This block of text will be filled and

indented 5 spaces from both the left and the right margins (i.e., centered).

```
.DS I F 5
"We the people of the United States, in order to
form a more perfect union, establish justice,
ensure domestic tranquility, 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."
.DE
```

7.2. Floating Displays

```
.DF [format] [fill] [rindent]
one or more lines of text
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as .DS {7.1}, except floating displays, indent, no indent, and centering are always calculated from the initial left margin, because the prevailing indent can change between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (1/2 a vertical space) always occurs both before and after a floating display.

The user can exercise great control over the output positioning of floating displays through the use of two number registers, De and Df. When a floating display is encountered by nroff or troff, it is processed and placed onto a queue of displays waiting to be output. Displays are always removed from the queue and printed in the order that they were entered on the queue, which is the order that they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, then the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the setting of the Df register (see below). The De register (see below) controls whether or not text will appear on the current page after a floating display has been produced.

As long as the 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 has specified top-of-page output. When a display is output it is also removed from the queue.

When the end of a section (when using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This will occur before a .SG, .CS, or .TC is processed.

A display is said to "fit on the current page" if there is enough room to contain the entire display on the page, or if the display is longer than one page in length and less than half of the current page has been used. Also note that a wide (full page width) display will never fit in the second column of a two-column document.

The registers, their settings, and their effects are as follows:

Values for De Register	
Value	Action
Ø	DEFAULT: No special action occurs.
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 values the action performed is for the value 1.

Values for Df Register	
Value	Action
Ø	Floating displays will not be output until end of section (when section-page numbering) or end of document.
1	Output the new floating display on the current page if there is room, otherwise hold it until the end of the section or document.
2	Output exactly one floating display from the queue at the top of a new page or column (when in two-column mode).
3	Output one floating display on current page if there is room. Output exactly one floating display at the top of a new page or column.
4	Output as many displays as will fit (at least one), starting at the top of a new page or column. Note that if register De is set to 1, each display will be followed by a page eject, causing a new top of page to be reached where at least one more display will be output. (This also applies to value 5, below.)
5	DEFAULT: Output a new floating display on the current page if there is room. Output at least one, but as many displays as will fit starting at the top of a new page or column. NOTE: For any value greater than 5 the action performed is for the value 5.

The .WC macro {12.4} can also be used to control handling of displays in double-column mode and to control the break in the text before floating displays.

7.3. Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The .TS (Table Start) and .TE (Table End) macro use the tbl(1) processor. They are used to delimit the text to be examined by tbl(1) as well as to set proper spacing around the table. The display function and the tbl(1) delimiting function are independent of one another, however, to keep together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines, the .TS-.TE block should be enclosed within a display (.DS-.DE). Floating tables can be enclosed inside floating displays (.DF-.DE).

The macros .TS and .TE also permit the processing of tables that extend over several pages. If a table heading is needed for each page of a multi-page table, specify the argument "H" to the .TS macro as above. Following the options and format information, the table heading is typed on as many lines as required and followed by the .TH macro. The .TH macro must occur when ".TS H" is used. Note that this is not a feature of tbl(1), but of the macro definitions provided by MM.

The table header macro .TH can 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
```

causes the table heading to appear at the top of the first table segment, without appearing 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. Use this feature when a single complex table must be broken into segments (for example, too many blocks of filled text). Each segment having its own .TS H\.TH sequence, would have its own header. However, if each table segment after the first uses .TS H\.TH N, then the table header will only appear at the beginning of the table and the top of each new page or column that the table continues onto.

For nroff, the -e option (-E for mm(1) {2.1}) can be used for terminals, such as the 450, that are capable of finer printing resolution. This causes better alignment of features such as the lines forming the corner of a box. Note that -e is not effective with col(1).

7.4. Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

The equation setters eqn(1) and neqn(1) [6.7] can use the .EQ (Equation Start) and .EN (Equation End) macros as delimiters in the same way that tbl(1) uses .TS and .TE; however, .EQ and .EN must occur inside a .DS-.DE pair.

WARNING

There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify eqn/neqn defines, .DS and .DE must not be used; otherwise extra blank lines will appear in the output.

The .EQ macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the vertical center of the general equation. The Eq register can be set to 1 to change the labeling to the left margin.

The equation will be centered for centered displays; otherwise the equation will be adjusted to the opposite margin from the label.

7.5. Figure, Table, Equation, and Exhibit Captions

```
.FG   [title] [override] [flag]
.TB   [title] [override] [flag]
.EC   [title] [override] [flag]
.EX   [title] [override] [flag]
```

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) and .EX (Exhibit Caption) macros are normally used inside .DS-.DE pairs to automatically number and title figures, tables, and equations. They use registers Fg, Tb, Ec, and Ex, respectively (see {2.4} on -rN5 to reset counters in sections). As an example, the call:

```
.FG "This is an illustration"
```

yields:

Figure 1. This is an illustration

.TB replaces Figure with TABLE; .EC replaces Figure with Equation, and .EX replaces Figure with Exhibit. Output 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 can be changed using the .af request of the formatter. The format of the caption can be changed from "Figure 1. Title" to "Figure 1 - Title" by setting the Of register to 1.

The override string is used to modify the normal numbering. If flag is omitted or Ø, override is used as a prefix to the number; if flag is 1, override is used as a suffix; and if flag is 2, override replaces the number. If -rN5 {2.4} is given, section-figure numbering is set automatically and user-specified override string is ignored.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure, equation, and exhibit captions usually occur after the corresponding figures and equations.

7.6. List of Figures, Tables, Equations, and Exhibits

A List of Figures, List of Tables, List of Exhibits, and List of Equations can be obtained. They will be printed after the Table of Contents is printed if the number registers Lf, Lt, Lx, and Le (respectively) are set to 1. Lf, Lt, and Lx are 1 by default; Le is Ø by default.

The titles of these lists can 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
```

SECTION 8
FOOTNOTES

There are two macros that delimit the text of footnotes;* a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1. Automatic Numbering of Footnotes

Footnotes are automatically numbered by typing the three characters "*F" immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

8.2. Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The .FS (Footnote Start) marks the beginning of the text of the footnote, and the .FE marks its end. The label on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes can be intermixed. If a footnote is labeled (.FS label), the text to be footnoted must be followed by label, rather than by "*F". The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are not permitted between the .FS and .FE macros. Automatically-numbered footnotes cannot be used for information such as title and abstract placement on the cover sheet, but labeled footnotes are allowed. Similarly, only labeled footnotes can be used with tables {7.3}. Examples:

---

* Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request in the Nroff User's Manual).

1. Automatically-numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The 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 a unpaddable space {3.3} to "*F" or label when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix C illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3. Format of Footnote Text ●

```
.FD [arg] [1]
```

The user can control the formatting style within the footnote text by specifying text hyphenation, right margin justification, and text indentation. The user can also specify left- or right-justification of the label when text indenting is used. Invoke the .FD macro to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is determined by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests {9}.

Ø	.nh	.ad	text indent	label left justified
1	.hy	.ad	text indent	label left justified
2	.nh	.na	text indent	label left justified
3	.hy	.na	text indent	label left justified
4	.nh	.na	no text indent	label left justified
5	.hy	.ad	no text indent	label left justified
6	.nh	.na	no text indent	label left justified
7	.hy	.na	no text indent	label left justified
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	text indent	label right justified
1Ø	.nh	.na	text indent	label right justified
11	.hy	.na	text indent	label right justified

If the first argument to .FD is out of range, the effect is as if .FD Ø were specified. If the first argument is omitted or null, the effect is equivalent to .FD 1Ø in nroff and to .FD Ø in troff; these are also the respective initial defaults.

If a second argument is specified, automatically-numbered footnotes will begin again with 1 when a first-level heading is present. This is 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.

For long footnotes that must continue onto the following page, the last line of the footnote on the current page is hyphenated. Except for this case (over which the user has control by specifying an even argument to .FD), hyphenation across pages is inhibited by MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In troff, footnotes are set in type that is two points smaller than the point size used in the body of the text.

8.4. Spacing Between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register Fs to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a six-point vertical space) to occur between footnotes.

SECTION 9
PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the page header. Text printed at the bottom of each page is called the 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 can have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways to change them. We use the term header (not qualified by even or odd) to mean the line of the page header that occurs on every page, and similarly for the term footer.

9.1. Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a release-paper, the page header on the first page is automatically suppressed provided a break does not occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do not cause a break and are permitted before the .MT macro call.

9.2. Page Header

```
.PH [arg]
```

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

```
"'left-part'center-part'right-part'"
```

If it is inconvenient to use the apostrophe (') as the delimiter (i.e., because it occurs within one of the parts), it can be replaced uniformly by any other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. The page number contained in the P register is an Arabic number. The format of the number can be changed by the .af request.

If debug mode is set using the flag -rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the Source Code Control System (SCCS) release and level of MM (thus identifying the current version {12.3}), followed by the current line number within the current input file.

9.3. 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. The initial value is a blank line.

9.4. Odd-Page Header

.OH [arg]

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5. Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rCn flag is specified on the command line {2.4}, the type of copy follows the footer on a separate line. In particular, if -rC3 or -rC4 (DRAFT) is specified, the footer also is initialized to contain the date {6.7.1}, instead of being a blank line.

9.6. 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. The initial value is a blank line.

9.7. Odd-Page Footer

.OF [arg]

This macro is the same as .EF, except that it applies to odd-numbered pages.

9.8. Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies .PF and/or .OF before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) replaces the footer on the first page only if the -rN1 flag is specified on the command line {2.4}.

9.9. Default Header and Footer with Section-Page Numbering

Pages can be numbered sequentially within sections {4.5}. To obtain this numbering style, specify -rN3 or -rN5 on the command line. In this case, the default footer is a centered section-page number, e.g. 7-2, and the default page header is blank.

9.10. Use of Strings and Registers in Header and Footer Macros ◆

String and register names are placed in the arguments to the header and footer macros. If the value of the string or register is to be computed when the respective header or footer is printed, the invocation must be escaped by four backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a .tl request during header or footer processing.

For example, the page number register P must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH "'''Page \\\\nP'"
```

creates a right-justified header containing the word Page followed by the page number. Similarly, to specify a footer with the section-page style, one specifies (see {4.2.2.5} for meaning of H1):

```
.PF "'''-\\\\n(H1-\\\\nP- '"
```

As another example, suppose that 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 then be:

```
.PF "''\\\\*(a]'"
```

If only one or two backslashes were used, the footer would print a constant value for a], namely, its value when the .PF appeared in the input text.

9.11. Header and Footer Example

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH ""
.PF ""
.EH "'\\\\nP''Revision 3'"
.OH "'Revision 3''\\\\nP'"
```

9.12. Generalized Top-of-Page Processing ♠

WARNING

This section is intended only for users accustomed to writing formatter macros.

During header processing, MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with no-space mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
.sp 3
.tl \\*()t
.if e 'tl \*()e
.if o 'tl \*()o
.sp
..
```

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 can redefine the .TP macro to cause any desired header processing {12.5}. Note that 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, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro is used to provide text that is to appear at the top of each page after the normal header and that have tab stops to align it with columns of text in the body of the document.

9.13. Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the .BS (Bottom-block Start) and .BE (Bottom-block End) macros will be printed at the bottom of each page,* after the footnotes (if any), but before the page footer. This block of text is removed by specifying an empty block, i.e.:

```
.BS
.BE
```

9.14. Top and Bottom Margins

```
.VM [top] [bottom]
```

.VM (Vertical Margin) allows the user to specify extra space at the top and bottom of the page. This space precedes the page header and follows the page footer. .VM takes two unscaled arguments and treats each as a v. For example:

```
.VM 1Ø 15
```

adds 1Ø blank lines to the default top of page margin, and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page can be decreased by re-defining .TP).

9.15. Proprietary Markings

```
.PM [code]
```

---

* 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 Release-Paper cover sheets.

.PM, for Proprietary Marking, appends to the page footer a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer. The code is:

Code	Meaning
.PM	no proprietary marking
.PM P	PRIVATE disclaimer
.PM N	NOTICE disclaimer
.PM BP	BELL LABORATORIES PROPRIETARY disclaimer
.PM BR	BELL LABORATORIES RESTRICTED disclaimer

The disclaimers are in a form approved for use by the Bell System.

9.16. Private Documents

```
.nr Pv value
```

The word PRIVATE can be printed centered and underlined on the second line of a document (preceding the page header). This is done by setting the Pv register:

Value	Meaning
.nr Pv Ø	do not print PRIVATE (default)
.nr Pv 1	PRIVATE on first page only
.nr Pv 2	PRIVATE on all pages

If Pv is 2, the user definable .TP can not be used because .TP is used by MM to print PRIVATE on all pages except the first page of a memorandum on which .TP is not invoked.

SECTION 10
TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively.

WARNING

This section will refer to cover sheets for Technical Memoranda and Release-Papers only. The mechanism for producing a Memorandum for File cover sheet was discussed earlier {6.4}.

These macros should normally appear only once at the end of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros. They can occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1. Table of Contents

```
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2]
[head3] [head4] [head5]
```

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the C1 register {4.4}. The arguments to .TC control the spacing before each entry, the placement of the 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 two arguments; headings whose level is less than or equal to slevel will have spacing blank lines (halves of a vertical space) before them. Both slevel and spacing default to 1. This means that first-level headings are preceded by one blank line (1/2 a vertical space). Note that slevel does not control what levels of heading have been saved; the saving of headings is the function of the C1 register {4.4}.

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots (leaders) separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to tlevel (default 2), the page numbers are justified at the right margin. In this case, the value of tab determines the character used to separate the heading text from the page number. If tab is Ø (the default value), leaders are used; if tab is greater than Ø, spaces are used. For headings whose level is greater than tlevel, the page numbers are separated from the heading text by two spaces (i.e., they are ragged right).

All additional arguments (e.g., head1, head2, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked (with at most four arguments), then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed; or the user-exit macro .TY is invoked and the word "CONTENTS" is not printed. By defining .TX or .TY and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: `\l'3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

.
.
.

If this macro were defined as .TY rather than .TX, the word "CONTENTS" would not appear. 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 at the left margin. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations can be changed by defining the Ci string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the Cl register. The arguments must be scaled. For example, with Cl =5,

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci Ø 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents, Oc and Cpk. 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 place the page number. Ordinarily the same .PF used in the body of the document will be adequate.

The List of Figures, Tables, etc. pages will be produced separately unless Cp is set to 1 which causes these lists to appear on the same page as the table of contents.

10.2. Cover Sheet

.CS [pages] [other] [total] [figs] [tbls] [refs]

The .CS macro generates a cover sheet in either the Technical Memorandum or release-paper style. See {6.4} for details on the Memorandum for File style cover sheet. All of the other information for the cover sheet is obtained from the data given before the .MT macro call {6.9}. If a Technical Memorandum style is used, the .CS macro generates the cover sheet for Technical Memorandum. The data that appears in the lower left corner of the Technical Memorandum cover sheet (the number of pages of text, the number of other pages, the total number of pages, the number of figures, the number of tables, and the number of references) is generated automatically. These values can be changed by supplying the appropriate arguments to the .CS macro. Any values that are omitted will be calculated automatically (0 is used for other pages). If the release-paper style is used, all arguments to .CS are ignored.

SECTION 11
REFERENCES

There are two macros that delimit the text of references, a string used to automatically number the references, and an optional macro to produce reference pages within the document.

11.1. Automatic Numbering of References

Automatically numbered references can be obtained by typing *(Rf immediately after the text to be referenced. This places the next sequential reference number (in a smaller point size) enclosed in brackets a half-line above the text to be referenced.

11.2. Delimiting Reference Text

The .RS and .RF macros are used to delimit text for each reference.

```
A line of text to be referenced.\*(Rf
.RS [string-name]
reference text
.RF
```

11.3. Subsequent References

.RS takes one argument, a string-name. For example:

```
.RS AA
reference text
.RF
```

The string AA is assigned the current reference number. It can 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 a half-line above the text to be referenced. No .RS/.RF is needed for subsequent references.

11.4. Reference Page

An automatically generated reference page is produced at the end of the document before the table of contents and the cover sheet are output. The reference page is entitled 'References'. This page contains the reference text (RS/RF). The user can change the reference page title by defining the Rp string. For example,

```
.ds Rp "New Title"
```

The optional .RP (Reference Page) macro can be used to produce reference pages anywhere within a document (i.e., within heading sections).

```
.RP [arg1] [arg2]
```

These arguments allow the user to control resetting of reference numbering, and page skipping.

arg1	Meaning
Ø	reset reference counter (default)
1	do not reset reference counter

arg 2	Meaning
Ø	cause a following .SK (default)
1	do not cause a following .SK

.RP need not be used unless the user wishes to produce reference pages elsewhere in the document.

SECTION 12
MISCELLANEOUS FEATURES

12.1. Bold, Italic, and Roman

```
.B [bold-arg] [previous-font-arg] ...
.I [italic-arg] [previous-font-arg] ...
.R
```

When called without arguments, .B changes the font to bold and .I changes to underlining (italic). This condition continues until the occurrence of a .R, when the regular roman font is restored. Thus,

```
.I
here is some text.
.R
```

yields:

here is some text.

If .B or .I is called with one argument, that argument is printed in the appropriate font (underlined in nroff for .I). Then the previous font is restored (underlining is turned off in nroff). If two or more arguments (maximum 6) are given to a .B or .I, the second argument is then concatenated to the first with no intervening space (1/12 space if the first font is italic), but is printed in the previous font; and the remaining pairs of arguments are similarly alternated. For example:

```
.I italic
text
.I right -justified
```

produces:

italic text right-justified

These macros alternate with the prevailing font at the time they are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB
.BI
.IR
.RI
.RB
.BR
```

Each takes a maximum of 6 arguments and alternates the arguments between the specified fonts.

Note that font changes in headings are handled separately {4.2.2.4}.

Anyone using a terminal that cannot underline might wish to insert:

```
.rm ul
.rm cu
```

at the beginning of the document to eliminate all underlining.

12.2. Justification of Right Margin

```
.SA [arg]
```

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used: current and default. .SA Ø sets both flags to no justification, i.e., it acts like the .na request. .SA 1 is the inverse: it sets both flags to cause justification, just like the .ad request. However, calling .SA without an argument causes the current flag to be copied from the default flag, thus performing either a .na or .ad, depending on what the default is. Initially, both flags are set for no justification in nroff and for justification in troff.

In general, use .na 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 lack thereof, for the remainder of the text is specified by inserting .SA Ø or .SA 1 once at the beginning of the document.

12.3. SCCS Release Identification

The string RE contains the SCCS release and level of the current version of MM. For example, typing:

```
This is version \*(RE of the macros.
```

produces:

```
This is version 15.1Ø3 of the macros.
```

This information is useful in analyzing suspected bugs in MM. The easiest way to have this number appear in your output is to specify -rD1 {2.4} on the command line, which causes the string IRE to be output as part of the page header {9.2}.

12.4. Two-Column Output

MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar 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.

It is possible to have full-page width footnotes and displays when in two column mode, although the 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 a macro, .WC (Width Control), which takes the following arguments:

N	Normal default mode (-WF, -FF, -WD)
WF	Wide Footnotes always (even in two column mode)
-WF	DEFAULT: turn off WF (footnotes follow column mode, wide in 1C mode, narrow in 2C mode, unless FF is set)
FF	First Footnote; all footnotes have the same width as the first footnote encountered for that page
-FF	DEFAULT: turn off FF (footnote style follows the settings of WF or -WF)
WD	Wide Displays always (even in two column mode)
-WD	DEFAULT: Displays follow whichever column mode is in effect when the display is encountered

For example: .WC WD FF will cause all displays to be wide, and all footnotes on a page to be the same width, while .WC N will reinstate the default actions. If conflicting settings are given to .WC the last one is used. That is, .WC WF -WF has the effect of .WC -WF.

12.5. Column Headings for Two-Column Output ♦

WARNING

This section is intended only for users accustomed to writing formatter macros.

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page {9}. This is accomplished by redefining the .TP macro {9.12} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl \\nP'OVERALL''
.tl ''TITLE''
.sp
.nf
.ta 16C 31R 34 50C 65R
```

```
left--center--right--left--center--right

            (where -- stands for the tab character)

--first column------second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

12.6. Vertical Spacing

```
.SP [lines]
```

There are several ways of obtaining vertical spacing, all with different effects.

The .sp request spaces the number of lines specified, unless no space (.ns) mode is on, in which case the request is ignored. This mode is typically set at the end of a page header in order to eliminate spacing by a .sp or .bp request that just happens to occur at the top of a page. This mode can be turned off with the restore spacing (.rs) request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row produce not the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many MM macros utilize .SP for spacing. For example, ".LE 1" {5.3.2} immediately followed by .P {4.1} produces only a single blank line (1/2 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. Like .sp, .SP is also inhibited by the .ns request.

12.7. Skipping Pages

.SK [pages]

The .SK macro skips pages, but retains the usual header and footer processing. If pages is omitted, null, or Ø, .SK skips to the top of the next page unless it is currently at the top of a page, in which case it does nothing. .SK n skips n pages. That is, .SK always positions the text that follows it at the top of a page, while .SK 1 always leaves one page that is blank except for the header and footer.

12.8. Forcing an Odd Page

.OP

This macro is used to ensure that the following text begins at the top of an odd-numbered page. If currently at the top of an odd page, no motion takes place. If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page) one blank page is produced, and printing resumes on the page after that.

12.9. Setting Point Size and Vertical Spacing

In troff, the default point size (obtained from the register S {2.4}) is 1Ø, with a vertical spacing of 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing can be changed by invoking the .S macro:

.S [point size] [vertical spacing]

The mnemonics, D for default value, C for current value, and P for previous value, can be used for both point size and vertical spacing arguments.

Arguments can be signed or unsigned. If an argument is negative, the current value is decremented by the specified amount. If the argument is positive, the current value is incremented by the specified amount. If an argument is unsigned, it is used as the new value. .S without arguments defaults to previous (P). If the first argument is specified but the second argument (vertical spacing) is not, then the default (D) value is used. The default value for vertical spacing is always 2p greater than the current point size

value selected.* A null ("") argument for either the first or second argument defaults to the current (C) value. For example (where n is a numeric value):

```
.S          = .SPP
.S""n       = .SCn
.Sn""       = .SnC
.Sn         = .SnD
.S""        = .SCD
.S"" ""     = .SCC
.Snn        = .Snn
```

If a point size argument is greater than 99, the default point size (D) 10 is restored. If a vertical spacing argument is greater than 99, the default vertical spacing (D) +2p is used. For example:

```
.S 12 111       =>      .S 12 14
.S 110          =>      .S 10 12
```

12.10. Producing Accents

The following strings can be used to produce accents for letters when using the troff processor:

	Input	Output
Grave accent	a*`	à
Acute accent	a*'	á
Circumflex	a*^	â
Tilde	n*~	ñ
Cedilla	c*,	ç
Lower-case umlaut	a*:	ä
Upper-case umlaut	A*;	Ä

* Footnotes {8} are printed in a size two points smaller than the point size of the body, with an additional vertical spacing of three points between footnotes.

12.11. Inserting Text Interactively

.RD [prompt] [diversion] [string]

.RD (ReaD insertion) allows a user to stop the standard output of a document and to read text from the standard input until two consecutive newlines are found. When the newlines are encountered, normal output is resumed.

.RD follows the formatting conventions in effect. Thus, the examples below assume that the .RD is invoked in no fill mode (.nf).

The first argument is a prompt which will be printed at the terminal. If no prompt is given, .RD signals the user with a BEL on terminal output.

The second argument, a diversion name, allows the user to save all the entered text typed after the prompt. The third argument, a string name, allows the user to save for later reference the first line following the prompt. For example:

```
.RD Name aa bb
```

produces

```
Name: (user types) J. Jones
16 Elm Rd.,
Piscataway
```

The diversion aa will contain:

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string bb " (" *(bb) contains "J. Jones".

A newline followed by a control-D (EOF) also allows the user to resume normal output.

SECTION 13
ERRORS AND DEBUGGING

13.1. Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which can contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix E.)
- Processing terminates, unless the register D {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

WARNING

The error message is printed by writing it directly to the user's terminal. If an output filter, such as 300(1) or 450(1) is being used to post-process nroff output, the message can be garbled by being intermixed with text held in that filter's output buffer. If either tbl(1) or eqn(1)/neqn(1), or both are being used, and if the -olist option of the formatter causes the last page of the document not to be printed, a harmless broken pipe message results.

13.2. Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .FE or .DE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is

to search backwards from the termination point looking for the corresponding .DS, .DF, or .FS.

The following command:

```
grep -n "^\.[EDFT][EFNQS]" files ...
```

prints all the .DS, .DF, .DE, .FS, .FE, .TS, .TE, .EQ, and .EN macros found in files ..., each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

SECTION 14
EXTENDING AND MODIFYING THE MACROS ●

14.1. Naming Conventions

In this section, the following conventions are used to describe legal names:

n:	digit
a:	lower-case letter
A:	upper-case letter
x:	any letter or digit (any alphanumeric character)
s:	special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that request, macro, and string names are kept by the formatters in a single internal table, so that there must be no duplication among such names. Number register names are kept in a separate table.

14.1.1. Names Used by Formatters

requests:
- aa (most common)
- an (only one, currently: .c2)

registers:
- aa (normal)
- .x (normal)
- .s (only one, currently: .$)
- % (page number)

14.1.2. Names Used by MM

macros:
- AA (most common, accessible to user)
- A (less common, accessible to user)
-)x (internal, constant)
- >x (internal, dynamic)

strings:
- AA (most common, accessible to user)
- A (less common, accessible to user)
-]x (internal, usually allocated to specific functions throughout)
- }x (internal, more dynamic usage)

registers: Aa (most common, accessible to users)
An (common, accessible to user)
A (accessible, set on command line)
:x (mostly internal, rarely accessible, usually dedicated)
;x (internal, dynamic, temporaries)

14.1.3. Names Used by EQN/NEQN and TBL

The equation preprocessors, eqn(1) and neqn(1), use registers and string names of the form nn. The table preprocessor, tbl(1), uses names of the form:

a- a+ a| nn #a ## #- #^ ^a T& TW

14.1.4. User-Definable Names

After the above, what is left for user extensions? To avoid problems, we suggest using names that consist either of a single lower-case letter, or of a lower-case letter followed by anything other than a lower-case letter. The following is a sample naming convention:

macros: aA
Aa

strings: a
a) (or a], or a}, etc.)

registers a
aA

14.2. Sample Extensions

The following paragraphs explain formatting of Appendix headings and hanging extensions.

14.2.1. Appendix Headings

The following gives a way of generating and numbering appendices:

```
.nr Hu 1
.nr a Ø
.de aH
.nr a +1
.nr P Ø
.PH "'''Appendix \\na - \\\\\\\\nP'"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ''.aH "title"'' begins a new page (with the page header changed to Appendix a - n) and generates an unnumbered heading of title, which, if desired, can be saved for the table of contents. Those who wish Appendix titles to be centered must, in addition, set the register Hc to 1 {4.2.2.3}.

14.2.2. Hanging Indent with Tabs

The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the mark. Each argument is separated from the previous one by a tab character; tab settings are defined later. Since the first argument can begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro.* The "\t" is translated by the formatter into a tab character. The "\c" is used to concatenate the line of text that follows the macro to the line of text built by the macro. The macro definition and an example of its use follow:

---

* The two-character sequence "\&" is understood by the formatters to be a zero-width space; it causes no output characters to appear.

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
..
  .
  .
  .
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) can still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\[]c none none no
Hyphenation indicator character is set to "c" or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	-	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) can still be split across lines.
.hy	on	-	no	Hyphenate. Automatic hyphenation is turned on.
.hc	c	none	none	no Hyphenation indicator character is set to c or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

APPENDIX A
DEFINITIONS OF LIST MACROS ●

WARNING

This appendix is intended only for users accustomed to writing formatter macros.

Here are the definitions of the list-initialization macros {5.3.3}:

```
.de AL
.nr !D Ø
.if !@\\$1@@ .if !@\\$1@l@ .if !@\\$1@a@ .if !@\\$1@A@
     \&.if !@\\$1@I@ .if !@\\$1@i@ .)D "AL:bad arg:\\$1"
.if \\n(.$<3 \{.ie \w@\\$2@=Ø .)L \\n(Lin Ø 2n 1 "\\$1"
.el .LB Ø\\$2 Ø 2 1 "\\$1" \}
.if \\n(.$>2 \{.ie \w@\\$2@=Ø .)L \\n(Lin Ø 2n 1 "\\$1" Ø 1
.el .LB Ø\\$2 Ø 2 1 "\\$1" Ø 1 \}
..
.de BL
.nr ;Ø \\n(Pi
.if (\\n(.$>Ø)&(\w@\\$1@>Ø) .nr ;Ø Ø\\$1
.ie \\n(.$<2 .LB \\n(;Ø Ø 1 Ø \\*(BU
.el .LB \\n(;Ø Ø 1 Ø \\*(BU Ø 1
.rr ;Ø
..
.de DL
.nr ;Ø \\n(Pi
.if (\\n(.$>Ø)&(\w@\\$1@>Ø) .nr ;Ø Ø\\$1
.ie \\n(.$<2 .LB \\n(;Ø Ø 1 Ø \(em
.el .LB \\n(;Ø Ø 1 Ø \(em Ø 1
.rr ;Ø
..
.de ML
.if \\n(.$<1 .)D "ML:missing arg"
.nr ;Ø \w@\\$1@u/3u/\\n(.su+1u\" get size in n's
.ie \\n(.$<2 .LB \\n(;Ø Ø 1 Ø "\\$1"
.el .if \\n(.$=2 .LB Ø\\$2 Ø 1 Ø "\\$1"
.if \\n(.$>2 \{.if !\w@\\$2@ .LB \\n(;Ø Ø 1 Ø "\\$1" Ø 1
.       if \w@\\$2@ .LB Ø\\$2 Ø 1 Ø "\\$1" Ø 1 \}
..
.de RL
.nr ;Ø 6
.if (\\n(.$>Ø)&(\w@\\$1@>Ø).nr ;Ø Ø\\$1
.ie \\n(.$<2 .LB \\n(;Ø Ø 2 4
.el .LB \\n(;Ø Ø 2 4 1 Ø 1
```

```
.rr ;Ø
..
.de VL
.if \\n(.$<1 .)D "VL:missing arg"
.ie \\n(.$<3 .LB Ø\\$1 Ø\\$2 Ø Ø
.el .LB Ø\\$1 Ø\\$2 Ø Ø \& Ø 1
..
```

Any of these can be redefined to produce different behavior: e.g., to provide two spaces between the bullet of a bullet item and its text, redefine .BL as follows before invoking it:

```
.de BL
.LB 3  Ø  2  Ø  \\\\*(BU
..
```

APPENDIX B
USER-DEFINED LIST STRUCTURES ♦

WARNING

This appendix is intended only for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to 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, you should know that the number register :g is used by the MM list macros to determine the current list level; it is Ø if there is no currently active list. Each call to a list-initialization macro increments :g, and each .LE call decrements it.

```
.de aL
'\"    register g is used as a local temporary to
'\"         save :g before it is changed below
.nr g \\n(:g
.if \\ng=Ø .AL A \" give me an A.
.if \\ng=1 .LB \\n(Li Ø 1 4 \" give me a [1]
.if \\ng=2 .BL \" give me a bullet
.if \\ng=3 .LB \\n(Li Ø 2 2 a \" give me an a)
.if \\ng=4 .ML + \" give me 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
```

will yield:

A. first line.

[1] second line.

B. third line.

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item:

```
.de bL
.ie \\n(.$ .nr g \\$1 \" if there is an argument,
'\" that is the level
.el .nr g \\n(:g \" if no argument, use current level
.if \\ng-\\n(:g>1 .)D "**ILLEGAL SKIPPING OF LEVEL"
'\" increasing level by more than 1
.if \\ng>\\n(:g \{.aL \\ng-1 \" if g > :g, begin new list
.     nr 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 MM .LC macro (List Clear) 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 Ø". If text is to be resumed at the end of a list, insert the call ".LC Ø" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC Ø
fifth line.
```

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.

APPENDIX C
SAMPLE FOOTNOTES

This appendix lists several footnote styles for both labeled and automatically-numbered footnotes. As shown below, nroff produces one style of footnotes; troff would produce five different styles. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page.

With the footnote style set to the nroff default, we process a footnote* followed by another one.** Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,*** and another.**** The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.*****

---

* This is the first footnote text example (.FD 1Ø). This is the default style for nroff. The right margin is not justified. Hyphenation is not permitted. The text is indented, and the automatically generated label is right-justified in the text-indent space.

** This is the second footnote text example (.FD 1Ø). This is also the default nroff style but with a long footnote label provided by the user.

*** This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is left-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

**** This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

***** This is the fifth footnote example (.FD 6). The right margin is not justified, hyphenation is not permitted, the footnote text is not indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 1Ø
With the footnote style set to the
.I nroff
default, we process a footnote\*F
.FS
This is the first footnote text example (.FD 1Ø).
This is the default style for
.I nroff.
he right margin is
.I not
justified.
Hyphenation is
.I not
permitted.
The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.*****\[   ([ this stands for a space)


.FS *****
This is the second footnote text example (.FD 1Ø).
This is also the default
.I nroff
style but with a long
footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to
hyphenate, right margin justification,
indent, and left justify the label.
Here is a footnote,\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified,
the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example,
hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro,
to no hyphenation, no right margin justification,
```

```
no indentation, and with the label left-justified.
Here comes the final one.\*F\
.FS
This is the fifth footnote example (.FD 6).
The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning
of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid
of their country.
.FE
```

APPENDIX D
SAMPLE LETTER

NOTE

This appendix gives the input for a sample letter which can be output in nroff, troff, or both.

```
.ND "May 31, 1979"
.TL 334455
Out-of Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT Ø
.DS
J. M. Jones:
.DE
.P
Please use the following description for the
Out-of-Hours course "Document Preparation on
the UNIX*
.FS *
UNIX is a Trademark
of Bell Laboratories.
.FE
Time-sharing System:"
.P
The course is intended for clerks, typists, and others
who intend to use the UNIX system
for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system;
using appropriate output terminals.
.LI Files:
how text is stored on the system;
directories;
manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier
to make;
how to use the editing system to
add, delete, and move lines of text;
how to make corrections.
.LI "Text processing:"
basic concepts;
use of general-purpose formatting packages.
```

```
.LI "Other facilities:"
additional capabilities useful to the typist such as the,
.I "typo, spell, diff,"
and
.I grep
commands and a desk-calculator package.
.LE
.SG jrm
.NS
S. P. Lename
H. O. Del
M. Hill
.NE
```

APPENDIX E
ERROR MESSAGES

E.1. MM Error Messages

Each MM error message consists of a standard part followed by a variable part. The standard part is of the form:

ERROR:input line n:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:*

Check TL, AU, AS, AE, MT sequence	The proper sequence of macros for the beginning of a memorandum is shown in {6.9}. Something has disturbed this order.
AL:bad arg:value	The argument to the .AL macro is not one of 1, A, a, I, or i. The incorrect argument is shown as value.
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.4}.
DS:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.

* This list is set up by ".LB 37 0 2 0" {5.4}.

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.
DS:missing DE	.DS or .DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
FE:no FS	.FE has been encountered with no previous .FS to match it.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but value has been supplied instead.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing arg	.H needs at least 1 argument.

HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as value.
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.

E.1.1. 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
: is caused by (a) setting a page width that is negative or extremely short, (b) setting a page length that is negative or extremely short, (c) reprocessing a macro package (e.g. performing a .so to a macro package that was requested from the command line), and (d) requesting the -sl option to troff on a document that is longer than ten pages.

Cannot execute filename
: is given by the .! request if it cannot find the filename.

Cannot open filename
: is issued if one of the files in the list of files to be processed cannot be opened.

Exception word list full
: indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

Line overflow
: means that the output line being generated was too long for the formatter's line buffer. The excess was discarded. See the Word overflow message below.

Non-existent font type
: means that a request has been made to mount an unknown font.

Non-existent macro file
: means that the requested macro package does not exist.

Non-existent terminal type
: means that the terminal options refers to an unknown terminal type.

Out of temp file space
: means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.

Too many page numbers
: is issued when the list of pages specified to the formatter -o option is too long.

Too many string/macro names
: is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.

Too many number registers
: number register names is full. Unneeded registers can be deleted by using the .rr request.

Word overflow
: means that a word being generated exceeded the formatter's word buffer. The excess characters were discarded. A likely cause for this and for the "Line overflow" message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by eqn(1) neqn(1).

APPENDIX F
SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

F.1. MM Macro Names

The following is an alphabetical list of macro names used by MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are not, in general, invoked directly by the user. Rather, they are user exits called from inside header, footer, or other macros.

1C One-column processing {12.4}
.1C

2C Two-column processing {12.4}
.2C

AE Abstract End {6.4}
.AE

AF Alternate Format of Subject/Date/From block {6.7.2}
.AF [company-name]

AL Automatically-incremented List start {5.3.3.1}
.AL [type] [text-indent] [1]

AS Abstract Start {6.4}
.AS [arg] [indent]

AT Author's Title {6.2}
.AT [title] ...

AU Author information {6.2}
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

AV Approval signature {6.11.3}
.AV [name]

B Bold {12.1}
.B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]

BE Bottom End {9.13}
.BE

BI Bold/Italic {12.1}
.BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]

BL Bullet List start {5.3.3.2}
.BL [text-indent] [1]

BR Bold/Roman {12.1}
.BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]

BS Bottom Start {9.13}
.BS

CS Cover sheet {10.2}
.CS [pages] [other] [total] [figs] [tbls] [refs]

DE Display End {7.1}
.DE

DF Display Floating start {7.2}
.DF [format] [fill] [right-indent]

DL Dash List start {5.3.3.3}
.DL [text-indent] [1]

DS Display Static start {7.1}
.DS [format] [fill] [right-indent]

EC Equation Caption {7.5}
.EC [title] [override] [flag]

EF Even-page Footer {9.6}
.EF [arg]

EH Even-page Header {9.3}
.EH [arg]

EN End equation display {7.4}
.EN

EQ Equation display start {7.4}
.EQ [label]

EX Exhibit caption {7.5}
.EX [title] [override] [flag]

FC Formal Closing {6.11}
.FC [closing]

FD Footnote Default format {8.3}
.FD [arg] [1]

FE Footnote End {8.2}
.FE

FG Figure title {7.5}
.FG [title] [override] [flag]

FS Footnote Start {8.2}
.FS [label]

H Heading-numbered {4.2}
.H level [heading-text] [heading-suffix]

HC Hyphenation Character {3.4}
.HC [hyphenation-indicator]

HM Heading Mark style (Arabic or Roman numerals, or letters) {4.2.2.5}
.HM [arg1] ... [arg7]

HU Heading-Unnumbered {4.3}
.HU heading-text

HX * Heading user exit X (before printing heading) {4.6}
.HX dlevel rlevel heading-text

HY Heading user exit Y (before printing heading) {4.6}
.HY dlevel rlevel heading-text

HZ * Heading user exit Z (after printing heading) {4.6}
.HZ dlevel rlevel heading-text

I Italic (underline in nroff) {12.1}
.I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]

IB Italic/Bold {12.1}
.IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]

IR Italic/Roman {12.1}
.IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]

LB List Begin {5.4}
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

LC List-status Clear {Appendix A}
.LC [list-level]

LE List End {5.3.2}
.LE [1]

LI List Item {5.3.1}
.LI [mark] [1]

ML Marked List start {5.3.3.4}
.ML mark [text-indent] [1]

MT Memorandum Type {6.6}
.MT [type] [addressee] or .MT [4] [1]

ND New Date {6.7.1}
.ND new-date

NE Notation End {6.11.2}
.NE

NS Notation Start {6.11.2}
.NS [arg]

nP Double-line indented Paragraphs {4.1}
.nP

OF Odd-page Footer {9.7}
.OF [arg]

OH Odd-page Header {9.4}
.OH [arg]

OK Other Keywords for TM cover sheet {6.5}
.OK [keyword] ...

OP Odd Page {12.8}
.OP

P Paragraph {4.1}
.P [type]

PF Page Footer {9.5}
.PF [arg]

PH Page Header {9.2}
.PH [arg]

PM Proprietary Marking {9.15}
.PM [code]

PX * Page-header user exit {9.12}
.PX

R Return to regular (roman) font (end underlining in nroff) {12.1}
.R

RB Roman/Bold {12.1}
.RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]

RD Read insertion from terminal {12.11}
.RD [prompt] [diversion] [string]

RF Reference end {11.2}
.RF

RI Roman/Italic {12.1}
.RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]

RL Reference List start {5.3.3.5}
.RL [text-indent] [1]

RP Produce Reference Page {11.4}
.RP [arg] [arg]

RS Reference Start {11.2}
.RS [string-name]

S Set troff point size and vertical spacing {12.9}
.S [size] [spacing]

SA Set adjustment (right-margin justification) default {12.2}
.SA [arg]

SG Signature line {6.11.1}
.SG [arg] [1]

SK Skip pages {12.7}
.SK [pages]

SP Space-vertically {12.6}
.SP [lines]

```
TB    Table title {7.5}
      .TB [title] [override] [flag]

TC    Table of Contents {10.1}
      .TC [slevel] [spacing] [tlevel] [tab]  [head1]  [head2]
      [head3] [head4] [head5]

TE    Table End {7.3}
      .TE

TH    Table Header {7.3}
      .TH [N]

TL    Title of memorandum {6.1}
      .TL [charging-case] [filing-case]

TM    Technical Memorandum number(s) {6.3}
      .TM [number] ...

TP *  Top-of-Page macro {9.12}
      .TP

TS    Table Start {7.3}
      .TS [H]

TX *  Table-of-contents user exit {10.1}
      .TX

TY *  Table-of-contents user exit  (suppresses ``CONTENTS'')
      {10.1}
      .TY

VL    Variable-item list start {5.3.3.6}
      .VL text-indent [mark-indent] [1]

VM    Vertical Margins {9.14}
      .VM [top] [bottom]

WC    Width Control {12.4}
      .WC [format]
```

F.2. Strings

The following is an alphabetical list of string names used by MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU Bullet {3.7}
nroff: •
troff: •

Ci Contents indent up to seven args for heading levels (must be scaled) {10.1}

F Footnote numberer {8.1}
nroff: \u\\n+(p\d
troff:\v'-.4m'\\n+(:p\s0\v'.4m'

DT Date (current date, unless overridden) {6.7.1}
Month day, year (e.g., May 20, 1983)

EM Em dash string, produces an em dash for both nroff and troff {3.8}.

HF Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1}
3 3 2 2 2 2 2 (levels 1 and 2 bold, 3-7 underlined in noff, and italic in troff)

HP Heading point size list, up to seven codes for heading levels 1 through 7 {4.2.2.4.3}

Le Title for LIST OF EQUATIONS {7.6}

Lf Title for LIST OF FIGURES {7.6}

Lt Title for LIST OF TABLES {7.6}

Lx Title for LIST OF EXHIBITS {7.6}

RE SCCS Release and Level of MM {12.3}
Release.Level (e.g., 15.103)

Rf Reference numberer {11.1}

Rp Title for References {11.4}

Tm Trademark string places the letters "TM" one half-line above the text that it follows {3.9}

Note that if the release-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}.

Also accent strings are available {12.10}.

F.3. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set only from the command line or before the MM macro definitions are read by the formatter {2.4, 2.5}. See {1.4} for notes on setting and referencing registers.

A * Handles preprinted forms and the Bell Logo {2.4}
Ø, [Ø:2]

Au Inhibits printing of author's location, department, room, and extension in the "From" portion of a memorandum {6.2}
1, [Ø:1]

C * Copy type (Original, DRAFT, etc.) {2.4}
Ø (Original), [Ø:3]

Cl Contents level (i.e., level of headings saved for table of contents) {4.4}
2, [Ø:7]

Cp Placement of List of Figures, etc. {1Ø.1}
1 (on separate pages), [Ø:1]

D * Debug flag {2.4}
Ø, [Ø:1]

De Display eject register for floating dislays {7.2}
Ø, [Ø:1]

Df Display format register for floating displays {7.2}
5, [Ø:5]

Ds Static display pre- and post-space {7.1}
1, [Ø:1]

Ec Equation counter, used by .EC macro {7.5}
Ø, [Ø:?], incremented by 1 for each .EC call.

Ej Page-ejection flag for headings {4.2.2.1}
Ø (no eject), [Ø:7]

Eq Equation label placement {7.4}
Ø (right-adjusted), [Ø:1]

Ex Exhibit counter, used by .EX macro {7.5}
Ø, [Ø:?], incremented by 1 for each .EX call.

Fg Figure counter, used by .FG macro {7.5}
Ø, [Ø:?], incremented by 1 for each .FG call.

Fs Footnote space (i.e., spacing between footnotes) {8.4}
1, [Ø:?]

H1-H7 Heading counters for levels 1-7 {4.2.2.5}
Ø, [Ø:?], incremented by .H of corresponding level or .HU if at level given by register Hu. H2-H7 are reset to Ø by any heading at a lower-numbered level.

Hb Heading break level (after .H and .HU) {4.2.2.2}
2, [Ø:7]

Hc Heading centering level for .H and .HU {4.2.2.3}
Ø (no centered headings), [Ø:7]

Hi Heading temporary indent (after .H and .HU) {4.2.2.2}
1 (indent as paragraph), [Ø:2]

Hs Heading space level (after .H and .HU) {4.2.2.2}
2 (space only after .H 1 and .H 2), [Ø:7]

Ht Heading type (for .H: single or concatenated numbers) {4.2.2.5}
Ø (concatenated numbers: 1.1.1, etc.), [Ø:1]

Hu Heading level for unnumbered heading (.HU) {4.3}
2 (.HU at the same level as .H 2), [Ø:7]

Hy Hyphenation control for body of document {3.4}
Ø (automatic hyphenation off), [Ø:1]

L * Length of page {2.4}
66, [2Ø:?] (11i, [2i:?] in troff **

Le List of Equations {7.6}
Ø (list not produced) [Ø:1]

** For nroff, these values are unscaled numbers representing lines or character positions; for troff, these values must be scaled.

Lf List of Figures {7.6}
1 (list produced) [Ø:1]

Li List indent {5.3.3.1}
6, [Ø:?]

Ls List spacing between items by level {5.3.3.1}
5 (spacing between all levels) [Ø:5]

Lt List of Tables {7.6}
1 (list produced) [Ø:1]

Lx List of Exhibits {7.6}
1 (list produced) [Ø:1]

N * Numbering style {2.4}
Ø, [Ø:5]

Np Numbering style for paragraphs {4.1}
Ø (unnumbered) [Ø:1]

O * Offset of page {2.4}
.75i, [Ø:?] (Ø.5i, [Øi:?] in .nr :p -1 troff **

Oc Table of Contents page numbering style {1Ø.1}
Ø (lowercase Roman), [Ø:1]

Of Figure caption style {7.5}
Ø (period separator), [Ø:1]

P Page number, managed by MM {2.4}
Ø, [Ø:?]

Pi Paragraph indent {4.1}
5, [Ø:?]

Ps Paragraph spacing {4.1}
1 (one blank space between paragraphs), [Ø:?]

Pt Paragraph type {4.1}
Ø (paragraphs always left-justified), [Ø:2]

Pv "PRIVATE" header {9.16}
Ø (not printed), [Ø:2]

** For nroff, these values are unscaled numbers representing lines or character positions; for troff, these values must be scaled.

S * Troff default point size {2.4}
10, [6:36]

Si Standard indent for displays {7.1}
5, [0:?]

T * Type of nroff output device {2.4}
0, [0:2]

Tb Table counter {7.5}
0, [0:?], incremented by 1 for each .TB call.

U * Underlining style (nroff) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]

W * Width of page (line and title length) {2.4}
6i, [10:1365] (6i, [2i:7.54i] in .nr :p -1 troff **

** For nroff, these values are unscaled numbers representing lines or character positions; for troff, these values must be scaled.

TYPING DOCUMENTS ON THE ZEUS SYSTEM USING THE -MS MACROS WITH TROFF AND NROFF*

* This information is based on an article originally written by M.E. Lesk, Bell Laboratories.

Preface

This document describes a set of macros for preparing documents using the ZEUS troff and nroff formatting programs. Documents can be output using either a phototypesetter or a computer terminal without changing the input.

Section 1 describes procedures for creating document files. Section 2 tells how to print the documents. Section 3 outlines the use of macros for producing tables and special symbols. The appendix summarizes the -ms commands.

Refer to the sections on nroff and troff for further information on producing documents.

Table of Contents

SECTION 1
PREPARING THE FILE

1.1. Text

Type normally, except that instead of indenting for paragraphs, place the line

```
.PP
```

before each paragraph. This produces indenting after an extra line space.

Alternatively, the command .LP produces a left-aligned (block) paragraph. The paragraph spacing can be changed (Section 1.16).

1.2. Front Matter

Start front matter as follows:

```
[optional overall format .RP, Section 1.3]
.TL
Title of document (one or more lines)
.AU
Author(s) (one or more lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a document.
Line length is 5/6 of normal; use .ll here to change.
.AE  (abstract end)
text ... (begins with .lp, Section 1.1)
```

To omit some of the standard headings (such as abstract or author's institution), omit the fields and corresponding command lines. Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory; beginning with a .lp command starts the document with an ordinary paragraph.

NOTE

Do not begin a document with a line of text. Some -ms command must precede any text input. When in doubt, use .LP to get proper initialization. The

commands .lp, .LP, .TL, .SH, and .NH are also allowed.

1.3. Cover Sheets and First Pages

The first line of a document signals the general format of the first page. In particular, if the first command is .RP, a cover sheet with title and abstract is generated. The default format of no cover sheet is useful for scanning drafts.

In general, -ms is arranged so that only one form of a document need be stored. The first command gives the format, and unnecessary items for that format are ignored.

NOTE

Do not put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed between them may not be processed as expected. Some -ms command must precede any input text.

1.4. Page Headings

The -ms macros, by default, print a page heading containing a page number. A default page footing is provided only in nroff, where the date is used. Minor adjustments to the page headers/footers are made by redefining the strings LH, CH, and RH (which are the left, center, and right portions of the page headers), and the strings LF, CF, and RF (the left, center, and right portions of the page footer). To get the proper page number in these strings, use a backslash (\) as in:

```
.ds CH "\- \\n(PN \-
```

which defines a center header of the form

```
- 5 -
```

For page number, the number register PN should be used in preference to the register %.

For more complex formats, redefine the macros PT and BT, which are invoked (respectively) at the top and bottom of each page. The margins, taken from registers HM and for the top margin FM for the bottom margin, are normally one inch. The page header/footer is in the middle of that space. If these macros are redefined, be careful with parameters such

as point size or font.

1.5. Multicolumn Formats

The command

```
.MC [column width [gutter width]]
```

makes multiple columns with the specified column and gutter width. The maximum is as many columns as fit across the page. Whenever the number of columns is changed (except going from full width to some larger number of columns), a new page is started.

This feature is more useful for typeset output than for output to the terminal. Placing the command .2C in your document causes it to be printed in double-column format beginning at that point. The command .1C produces one-column format. Changing column format causes a page break.

1.6. Section Headings

Two commands, .NH and .SH, are used to produce section headings. Entering

```
.NH
type section heading here
can be several lines
```

produces a numbered section heading in boldface. The .SH command produces an unnumbered heading.

Every section heading must be followed by a paragraph beginning with .lp or .LP to indicate the end of the heading. Headings can contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a level number, and an appropriate subsection number is generated. Larger level numbers indicate deeper subsections. For example,

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1 Gladstone Branch

2.1.2 Montclair Branch

2.2. Boonton Line

1.7. Indented Paragraphs

Paragraphs with hanging numbers, such as references, are often handled with indented paragraphs.

Example 1. Simple Indentation

```
.IP [1]
Text for first paragraph, typed
normally for as long as necessary
on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as necessary on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs can be followed by an ordinary paragraph by entering .lp or .LP.

Example 2. Block Indentation

More sophisticated uses of .ip are also possible. If the label is omitted, for example, a plain block indent is produced. The lines

```
.IP
This material will
just be turned into a
block indent suitable for quotations.
.LP
```

produce

> This material will just be turned into a block indent suitable for quotations.

Example 3. Nonstandard Indentation

If a nonstandard amount of indenting is required, it is specified after the label (in character positions) and remains in effect until the next .lp or .LP. Thus, the general form of the .ip command contains two additional fields: the label and the indenting length. For example,

```
.ip first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.ip second:
And so forth.
.LP
```

produces the following:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second:
And so forth.

Example 4. Multiple Nested Indentations

It is also possible to produce multiple nested indents. The command .RS indicates that the next .ip starts from the current indentation level. Each .RE takes one level of indenting, so .RS and .RE commands must be balanced. The .RS command can be thought of as "move right" and the .RE command as "move left." For example,

```
.IP 1.
Customer Corporation
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

results in

1. Customer Corporation

    1.1 Murray Hill
    1.2 Holmdel
    1.3 Whippany
        1.3.1 Madison
    1.4 Chester

Example 5. Right Indentation

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) is bracketed with .QS and .QE.

1.8. Emphasis

To produce italics on the typesetter or underlining on the terminal, use

```
.I
as much text as you want
can be typed here
.R
```

as was done for these three words. The .R command restores the normal (usually Roman) font. If only one word is to be italicized or underlined, it can be input on a separate line with the .I command,

```
.I word
```

In this case, no .R is needed to restore the previous font.

Boldface output on the typesetter is produced by

```
.B
Text to be set in boldface
goes here
.R
```

This is also underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on a separate line with the .B command.

Size changes can be specified with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands can be repeated for increased effect.

To specify an underlined word on the typesetter, use the command

```
.UL word
```

There is no way to underline multiple words on the typesetter.

1.9. Footnotes

Material placed between lines with the commands .FS for footnote and .FE for footnote end is collected and placed at the bottom of the current page after an asterisk (*). By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (Section 1.16).

1.10. Displays and Tables

To prepare displays whose lines are not to be rearranged (such as tables), enclose the text in the commands .DS and .DE as follows:

```
table lines, like the
examples here, are placed
between .DS and .DE
```

By default, lines between .DS and .DE are indented and left-adjusted. It is also possible to center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered and not rearranged. Lines bracketed by .DS L and .DE are left-adjusted, not indented, and not rearranged. The command .DS is equivalent to .DS I, which indents and left-adjusts. For example,

```
these lines were preceded
by .DS C and  followed by
      a .DE command;
```

whereas

```
These lines were preceded
by .DS L and  followed by
a .DE command.
```

There is also a variant, .DS B, that makes the display into a left-adjusted block of text, then centers that entire block.

Normally, a display is kept on one page. To produce a long display split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I, respectively. An extra argument to the .DS I or .DS command specifies the amount to indent. There is no command to right-adjust lines.

1.11. Boxing Words or Lines

To draw a rectangular box around a word, use the command

```
.BX word
```

Longer pieces of text can be boxed by enclosing them with .B1 and .B2, as with

```
.B1
text...
.B2
```

Italics are preferred to boxes because boxes are not printed neatly on a terminal. However, col may be used to improve such terminal output. See ZEUS Reference Manual, Section 1.

1.12. Keeping Blocks Together

To keep a table or other block of lines together on a page, use the keep - release commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it begins on a new page. (Lines bracketed by .DS and .DE commands are automatically kept together this way.) There is also a keep floating (.KF) command. If a block preceded by .KF (instead of .KS) does not fit on the current page, it is moved down through the text until the top of the next page. Thus, no large blank space is introduced in the document.

1.13. Nroff/Troff Commands

The following commands from the basic formatting programs work for both typesetter and computer terminal output:

```
.bp   begin new page
.br   "break" stop running text from line to line
.sp n insert n blank lines
.na   do not adjust right margins
```

1.14. Date

By default, documents produced on computer terminals have the date at the bottom of each page, and documents produced on the typesetter do not. To force the date, use the .DA command. To force no date, use the .ND command. To force a fixed date, enter the date after the .DA command; for example,

```
.DA July 4, 1776
```

The command ".ND May 8, 1945" in .RP format places the specified date on the cover sheet and nowhere else. Place this line before the title.

1.15. Signature Line

To obtain a signature line, use the command .SG. The author's name is output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

1.16. Registers

Certain of the registers used by -ms can be altered to change default settings using commands beginning with .nr. For example,

```
/.nr PS 9
```

makes the default point size 9 point. If the effect is needed immediately, use the normal troff command in addition to changing the number register.

Reg.	Defines	Takes Effect	Default
PS	point size	next paragraph	1Ø
VS	line spacing	next paragraph	12 pts
LL	line length	next paragraph	6 inches
LT	title length	next paragraph	6 inches
PD	para. spacing	next paragraph	Ø.3 VS
PI	para. indent	next paragraph	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27 inches
HM	top margin	next page	1 inch
FM	bottom margin	next page	1 inch

It is also possible to alter the strings LH, CH, and RH (left, center, and right headers), and LF, CF, and RF (strings in the page footers). The page number on output is taken from register PN to permit changing its output style. For more complicated headers and footers, the macros PT and BT can be redefined as explained, in Section 1.4.

1.17. Accents

To simplify typing certain foreign words, strings representing common accent marks are defined for use on photocomposition systems and terminals on which strikeover characters have been defined.

Input	Output Character with:
*'e	accute accent
*`e	grave accent
*:u	umlaut (diaeresis)

*^e	circumflex
*~a	tilde
*Ce	hacek (wedge)
*,c	cedilla

SECTION 2
PRINTING THE DOCUMENT

After the document is prepared and stored on a file, it can be displayed on a terminal with the command

```
nroff -ms file
```

If double-column format (2C) is being used, pipe the nroff output through col by making the first line of the input

```
.pi /z/bin/col
```

The document can be printed on the typesetter with the command

```
troff -ms file
```

Many options are possible. In each case, if the document is stored in several files, list all the file names used. If equations or tables are used, eqn and/or tbl must be invoked as preprocessors.

SECTION 3
USING ADVANCED FORMAT OPTIONS

3.1. Special Symbols

To use Greek or mathematics symbols, see eqn for equation setting. To aid eqn users, -ms provides definitions of .EQ and .EN, which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ; the letters C, I, and L indicate centered (default), indented, and left-adjusted equations. If there is both a format argument and an equation number, give the format argument first, as in

```
.EQ L (1.3a)
```

for a left-adjusted equation numbered (1.3a).

3.2. Tables

The macros .TS and .TE are defined to separate tables from text with white space. A very long table with a heading can be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data to repeat the heading. If the table has no heading repeated from page to page, use the ordinary .TS and .TE macros.

APPENDIX A
LIST OF COMMANDS

Return to single-column format

2C	Start double-column format
AB	Begin abstract
AE	End abstract
AI	Specify author's institution
AU	Specify author
B	Begin boldface
DA	Provide the date on each page
DE	End display
DS	Start display (also CD, LD, ID)
EN	End equation
EQ	Begin equation
FE	End footnote
FS	Begin footnote
I	Begin italics
IP	Begin indented paragraph
KE	Release keep
KF	Begin floating keep
KS	Start keep
LG	Increase type size
LP	Left aligned block paragraph
ND	Change or cancel date
NH	Specify numbered heading
NL	Return to normal type size
PP	Begin paragraph
R	Return to regular font (usually Roman)
RE	End one level of relative indenting
RP	Use released paper format
RS	Relative indent increased one level
SG	Insert signature line
SH	Specify section heading
SM	Change to smaller type size
TL	Specify title
UL	Underline one word

Register Names

The following register names are used by -ms internally. Independent use of these names in one's own macros may produce incorrect output. No lowercase letters are used in any -ms internal name.

Number Registers Used in -ms

:	FC	H4	IQ	MF	NS	PO	TC	YY
#T	FL	H5	IR	MM	OI	PQ	TD	ZN
1T	FM	HM	IT	MN	OJ	PS	TN	
1T	FP	HT	KI	MO	PD	PX	TQ	
AV	GW	IF	L1	NA	PE	RO	TV	
CW	H1	IK	LE	NC	PF	ST	VS	
DW	H2	IM	LL	ND	PI	T1		WF
EF	H3	IP	LT	NF	PN	TB		YE

String Registers Used in -ms

'	AI	CS	EM	I	LB	OK	RP	TL
`	AU	CT	EN	I1	LD	PP	RQ	TM
^	B	D	EQ	I2	LG	PT	RS	TQ
~	BG	DA	EZ	I3	LP	PY	RT	TS
:	BT	DE	FA	I4	ME	QF	SO	TT
,	C	DS	FE	I5	MF	R	S1	UL
1C	C1	DW	FJ	ID	MH	R1	S2	WB
2C	C2	DY	FK	IE	MN	R2	SG	WH
A1	CA	E1	FN	IM	MO	R3	SH	WT
A2	CB	E2	FO	IP	MR	R4	SM	XD
A3	CC	E3	FQ	IZ	ND	R5	SN	XF
A4	CD	E4	FS	KE	NH	RC	SY	XK
A5	CF	E5	FV	KF	NL	RE	TA	
AB	CH	EE	FY	KQ	NP	RF	TE	
AE	CM	EL	HO	KS	OD	RH	TH	

NROFF USER'S MANUAL*

* This information is based on an article originally written by Joseph F. Ossanna, Bell Laboratories.

Preface

This document is a reference manual for the nroff text processors. The reader is expected to have some experience with these text processors before using this manual. For an introductory text, see Troff Tutorial.

Each section of this document covers an nroff command or set of related commands, and sections appear in order of use; that is, frequently used commands appear first. At the end of this documment are several summaries and appendixes.

Numerical parameters are indicated in this manual in two ways. +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. N means that an initial algebraic sign is not an increment indicator, but merely the sign of N. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lowercase letters and one/two character arguments are indicated by a pair of lowercase letters. Character string arguments are indicated by multicharacter mnemonics.

NOTE

The version of troff on ZEUS produces output for a Graphic Systems Inc. C/A/T phototypesetter. This device is not presently supported by ZEUS. Since the device is not present, it will always appear to be busy to troff.

Table of Contents

SECTION 1
BASIC INFORMATION

1.1. Introduction

Nroff and troff are text processors under the ZEUS Time-Sharing System that format text for typewriter-like terminals and for phototypesetters, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document with a user-designed style. Nroff and troff offer great freedom in document styling, including:

- Arbitrarily styled headers and footers
- Arbitrarily styled footnotes
- Multiple automatic sequence numbering for paragraphs, sections, etc.
- Multiple column output
- Dynamic font and point-size control
- Arbitrary horizontal and vertical local motions at any point
- A family of automatic overstriking, bracket construction, and line drawing functions

Nroff and troff are compatible with each other; it is possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input destined for either program. Nroff can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

1.2. Usage

The general form of invoking nroff or troff at ZEUS command level is

nroff options files (or troff options files)

where options represents any of a number of option arguments and files represents the list of files containing the document to be formatted. An argument consisting of a single

minus (-) is taken to be a file name corresponding to the standard input. If no file names are given, input is taken from the standard input. The following options can appear in any order as long as they appear before the files:

Option	Effect
-olist	Print only pages whose page numbers appear in a list that consists of comma-separated numbers and number ranges. A number range has the form N-M and means pages N through M; an initial -N means from the beginning to page N; a final N- means from N to the end.
-nN	Number first generated page N.
-sN	Stop every N pages. Nroff halts prior to every N pages (default N=1) to allow paper loading or changing, and resumes upon receipt of a new line. Troff stops the phototypesetter every N pages, produces a trailer to allow changing cassettes, and resumes after the phototypesetter START button is pressed.
-mname	Prepends the macro file **/usr/lib/tmac.**name to the input files.
-raN	Register a (one-character) is set to N.
-i	Read standard input after the input files are exhausted.
-q	Invoke the simultaneous input-output mode of the **rd** request.
	NROFF ONLY
-Tname	Specifies the name of the output terminal type. Currently defined names are: 37 for the Model 37 Teletypewriter **TN300** (default) for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, **450** for the DASI-450 (Diablo Hyterm).
-e	Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF ONLY

-t Direct output to the standard output instead of the phototypesetter.

-f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-b Troff reports whether the phototypesetter is busy or available. No text processing is done. (See NOTE in preface.)

-a Send a printable (ASCII) approximation of the results to the standard output.

-pN Print all characters in point size N, while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

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

nroff -o4,8-10 **-T**300S **-m**abc file1 file2

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named file1 and file2, specifies the output terminal as a DASI-300S, and invokes the macro package abc.

1.3. Form of Input

Input consists of text lines that are destined to be printed, interspersed with control lines that control subsequent processing. Control lines begin with a control character, normally a period (.) or acute accent (') followed by a one or two-character name that specifies a basic request or the substitution of a user-defined macro in place of the control line. The control character ' suppresses the break function (the forced output of a partially filled line) caused by certain requests. The control character can be separated from the request/macro name by white space (spaces and/or tabs). Names must be followed by either a space or a new line. Control lines with unrecognized names are ignored.

Various special functions can be introduced anywhere in the input by means of an escape character, normally a backslash (\). For example, the function **\n**R, causes the interpolation of the contents of the number register R in place of

the function; here R is either a single-character name as in **\nx**, or a left-parenthesis-introduced, two-character name as in **\n(**xx.

1.4. Formatter and Device Resolution

For internal processing, troff uses 432 units per inch, corresponding to the Graphic Systems phototypesetter, which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. Nroff uses 240 units per inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. Troff rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. Nroff rounds numerical input to the actual resolution of the output device indicated by the **-T** option (default Model 37 Teletype).

1.5. Numerical Parameter Input

Both nroff and troff accept numerical input with the appended scale indicators shown in the following table, where S is the current type size in points, V is the current vertical line spacing in basic units, and C is a nominal character width in basic units.

Scale Indicator	Meaning	Number of Basic Units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432x50/127	240x50/127
P	Pica = 1/6 inch	72	240/6
m	Em = S points	6xS	C
n	En = Em/2	3xS	C
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	V	V none
Default, see below			

In nroff, both the em and the en are taken to be equal to the C, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in nroff need not be the same, and constructed characters such as -> are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; the default is **VS** for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; the default is **p** for the **vs** request; the default is **u** for the requests **nr**, **if**, and **ie**. All other requests ignore any scale indicators. When a

number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator u needs 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 prepended to a number N to generate the distance to the vertical or horizontal place N. For vertically-oriented requests and functions, **|**N becomes the distance in basic units from the current vertical place on the page or in a diversion (Section 6.4) to the the vertical place N. For all other requests and functions, **|**N becomes the distance from the current horizontal place on the input line to the horizontal place N. For example,

```
.sp |3.2c
```

spaces in the required direction to 3.2 centimeters from the top of the page.

1.6. Numerical Expressions

Wherever numerical input is expected, the following can be used: an expression involving parentheses, the arithmetic operators +, -, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), 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 scaling differs from the default scaling. 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 1/2 the sum of 4.25 inches + 2 picas + 30 points.

SECTION 2
FONT AND CHARACTER SIZE CONTROL

2.1. Character Set

The troff character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set. Each set has 102 characters. These character sets are shown in the Appendix. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form **\(**xx where xx is a two-character name given in the Appendix. The three ASCII exceptions are mapped as follows:

ASCII Input:		Printed by troff:	
Character	Name	Character	Name
'	acute accent	'	close quote
`	grave accent	`	open quote
-	minus	-	hyphen

The characters ', `, and - can be input by **\'**, **\`**, and **\-**, respectively, or by their names (Section 2). The ASCII characters **@**, **#**, **"**, **'**, **`**, **<**, **>**, ****, **{**, **}**, **~**, **^**, and **-** exist only on the Special Font and are printed as a 1-em space if that font is not mounted.

Nroff recognizes the entire troff character set, but can print only ASCII characters, additional characters as are available on the output device, such characters as are able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ', `, and - print as themselves.

2.2. Fonts

The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4, respectively. The current font can be changed (among the mounted fonts) by use of the **ft** request, or by embedding at any desired point either **\f**x, **\f(**xx, or **\f**N, where x or xx is the name of a mounted font and N is a numerical font

position. It is not necessary to change to the special font; characters on that font are automatically handled. A request for a font that is named but not mounted is ignored. Troff can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation-dependent. In the subsequent discussion of font-related requests, F represents a one or two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

Nroff recognizes font control and (normally) underlines Italic characters.

2.3. Character Size

Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request changes or restores the point size. Alternatively, the point size is changed between any two characters by embedding a **\s**N at the desired point to set the size to N, or a **\s**±N to increment/decrement the size by N; **\s0** restores the previous size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. Nroff ignores type size control.

Request Form	Initial Value	If No Argument
.ps ±N	10 point	previous

Explanation: Point size set to ±N. Alternatively, embed \sN or \s±N. Any positive size value can be requested; if invalid, the next larger valid size results, with a maximum of 36. A paired sequence +N, -N works because the previous requested value is also remembered, but ignored in nroff. Relevant parameters are a part of the current environment

Request Form	Initial Value	If No Argument
.ss N	12/136 em	ignored

Explanation: Space-character size is set to N/36 ems. This size is the minimum word spacing in adjusted text. Ignored in nroff. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.cs	F N M	off

Explanation: Constant character space (width) mode is set on for font F (if mounted); the width of every character is taken to be N/36 ems. If M is absent, the em is that of the character's 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 be still or again in effect when the characters are physically printed. Ignored in nroff.

Request Form	Initial Value	If No Argument
.bd	F N	off

Explanation: The characters in font F are artificially made boldface 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 bold mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in nroff.

Request Form	Initial Value	If No Argument
.bd	S F N	off

Explanation: The characters in the Special Font are made boldface whenever the current font is F. The mode must be still or again in effect when the characters are physically printed.

Request Form	Initial Value	If No Argument
.ft F	Roman	previous

Explanation: Font changed to F. Alternatively, imbed **\fF**. The font name P is reserved to mean the previous font. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.fp N F	R,I,B,S	ignored

Explanation: Font position. This is a statement that a font named F is mounted on position N (1-4). It is a fatal error if F is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip that can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by troff is R, I, B, and S on positions 1, 2, 3 and 4.

SECTION 3
PAGE CONTROL

Top and bottom margins are not automatically provided; it is conventional to define two macros and to set traps for them at vertical positions Ø (top) and -N (N from the bottom). (See Sections 6 and 21.) A pseudo-page transition onto the first page occurs either when the first break occurs or when the first non-diverted text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the current diversion (Section 6.4) mean that the mechanism being described works during both ordinary and diverted output.

The usable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on nroff output are output-device dependent.

Request Form	Initial Value	If No Argument
.pl ±N	11 in	11 in

Explanation: Page length set to ±N. The internal limitation is about 75 inches in troff and about 136 inches in nroff. The current page length is available in the **.p** register. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.bp	±N=1	-

Explanation: Begin page. The current page is ejected and a new page is begun. If ±N is given, the new page number is ±N. Also see request **ns.** The default scale indicator is v (ignored if not specified). This request normally causes a break. The use of " ' " as control character (instead of .) suppresses the break function.

Request Form	Initial Value	If No Argument
.pn ±N	N=1	ignored

Explanation: Page number. The next page (when it occurs) has the page number ±N. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.

Request Form	Initial Value	If No Argument
.po ±N	0;26/127in*	previous

Explanation: Page offset. Values separated by ; are for nroff and troff, respectively. The current left margin is set to (+-N. The troff initial value provides about one inch of paper margin including the physical typesetter margin of 1/27 inch. In troff the maximum (line length)+(page offset) is about 7.54 inches (Section 5). The current page offset is available in the **.o** register. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.ne N	-	N=1 V

Explanation: Need N vertical space. If the distance, D, to the next trap position (Section 6.5) 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 < 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. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.mk R	none	internal

Explanation: Mark the current vertical place in an internal register (both associated with the current diversion level), or in register R, if given. See **rt** request.

Request Form	Initial Value	If No Argument
.rt ±N	none	internal

Explanation: Return upward only to a marked vertical place in the current diversion. If ±N (with respect to current place) is given, the place is ±N from the top of the page or

diversion or, if N is absent, to a place marked by a previous **mk**. The **sp** request (Section 5.3) can be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; for example using the sequence **.mk** R ... **.sp |\nRu.**

SECTION 4
TEXT FILLING, ADJUSTING, AND CENTERING

4.1. Filling and Adjusting

Normally, words are collected from input text lines and assembled into an output text line until some word does not fit. An attempt is then made the 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 then increased to spread out the line to the current line length minus any current indent. A word is any string of characters delimited by the space character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the unpaddable space character "\ " (backslash-space). The adjusted word spacings are uniform and the minimum interword spacing can be controlled with the **ss** request. In nroff, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **-e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation can all be prevented or controlled. The text length on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line (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** can be embedded 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 begins with a control character can be made to look like a regular text line by prefacing it with the nonprinting, zero-width filler character **\&**. Another way to do this is to specify output translation of some convenient character into the control character using **tr**.

4.2. Interrupted Text

The copying of an input line in nofill (non-fill) mode can be interrupted by terminating the partial line with a **\c**. The next encountered input text line is considered to be a continuation of the same line of input text. Similarly, a word within filled text is interrupted by terminating the word (or 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.

Request Form	Initial Value	If No Argument
.br	-	-

Explanation: Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

Request Form	Initial Value	If No Argument
.fi	fill on	-

Explanation: Fill subsequent output lines. The register **.u** is 1 in fill mode and Ø in nofill mode. This request normally causes a break. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.nf	fill on	-

Explanation: No fill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length. This request normally causes a break. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.ad c	adj,both	adjust

Explanation: Line adjustment is begun. If fill mode is not on, adjustment is deferred until fill mode is back on. If the type indicator c is present, the adjustment type is

changed as shown in the following table:

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

Request Form	Initial Value	If No Argument
.na	adjust	-

Explanation: No adjust. Adjustment is turned off; the right margin is ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.ce N	off	N=1

Explanation: Center the next N input text lines within the current (line length minus indent). If N=Ø, 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. Relevant parameters are a part of the current environment.

SECTION 5
VERTICAL SPACING

5.1. Base-Line Spacing

The vertical spacing (V) between the base-lines of successive output lines is set using the **vs** request with a resolution of 1/144 inch = 1/2 point in troff, and to the output device resolution in nroff. V must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set V to 2 points greater than the point size; troff default is 10-point type on 12-point spacing. The current V is available in the **.v** register. Multiple-V line separation (for example, double spacing) is requested with **ls.**

5.2. Extra Line-Space

If a word contains a construct that requires the output line containing it to have extra vertical space before and/or after it, the extra-line-space function (**\x**'N') can be embedded in or attached to that word. In this and other functions that have a pair of delimiters around their parameter (here '), 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 values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of Vertical Space

A block of vertical space is ordinarily requested using **sp,** which honors the no-space mode and which does not space past a trap. A contiguous block of vertical space can be reserved using **sv.**

Request Form	Initial Value	If No Argument
.vs N	1/6in; 12 pts	previous

Explanation: Set vertical base-line spacing size V. Transient extra vertical space available with **\x'N'**. Relevant parameters are a part of the current environment. The default scale indicator is p (ignored if not specified).

Request Form	Initial Value	If No Argument
.ls N	N=1	previous

Explanation: Line spacing set to +N. N-1 Vs (blank lines) are appended to each output text line. Appended blank lines are omitted if the text or previous appended blank line reached a trap position. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.sp N	-	N=1V

Explanation: Space vertically in either direction. If N is negative, the motion is backward (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see **ns**, and **rs** below). This request normally causes a break. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.sv N	-	N=1V

Explanation: Save a contiguous vertical block of size N. If the distance to the next trap is greater than N, N vertical space is output. No-space mode has no effect. If this distance is less than N, no vertical space is immediately output, but N is retained for later output (see **os**). Subsequent **sv** requests overwrite any retained N. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.os	-	-

Explanation: Output saved vertical space. No-space mode has no effect. Used to output a block of vertical space requested by an earlier **sv** request.

Request Form	Initial Value	If No Argument
.ns	space	-

Explanation: No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests without a next page number. The no-space mode is turned off when a line of output occurs, or with **rs.** Relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
.rs	space	-

Explanation: Restore spacing. The no-space mode is turned off. Relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
Blank text line.		-

Explanation: Causes a break and output of a blank line exactly like **sp 1.**

5.4. Line Length and Indenting

The maximum line length for fill mode is set with **ll.** The indent is set with **in;** an indent applicable to only the next output line is set with **ti.** The line length includes indent space but not page offset space. The line length minus the indent is the basis for centering with **ce.** 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.**

Request Form	Initial Value	If No Argument
.ll ±N	6.5 in	previous

Explanation: Line length is set to . In troff the maximum (line length)+(page offset) is about 7.54 inches. Relevant

parameters are a part of the current enviroment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.in ±N	N=Ø	previous

Explanation: Indent is set to ±N. The indent is prepended to each output line. This request normally causes a break. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.ti ±N	-	ignored

Explanation: Temporary indent. The next output text line will be indented a distance ±N with respect to the current indent. The resulting total indent cannot be negative. The current indent is not changed. This request normally causes a break. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

SECTION 6
MACROS, STRINGS, DIVERSIONS, AND POSITION TRAPS

6.1. 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 new line character, that can be interpolated 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 appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a control line beginning .xx 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.

6.2. Copy Mode Input Interpretation

During the definition and extension of strings and macros (not by diversion), the input is read in copy mode. The input is copied without interpretation except that:

- The contents of number registers indicated by **\n** are interpolated
- Strings indicated by ***** are interpolated
- Arguments indicated by **\$** are interpolated
- Concealed new lines indicated by ****(new line) are eliminated
- Comments indicated by **\"** are eliminated
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH, respectively
- \\ is interpreted as \

- \. is interpreted as .

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, \\n copies as \n, which is interpreted as a number register indicator when the macro or string is reread.

6.3. Arguments

When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments can be surrounded by double quotes to permit embedded space characters. Pairs of double quotes can be embedded in double quoted arguments to represent one double quote. If the desired arguments do not fit on a line, a concealed new line 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 Nth argument ($1 \le N \le 9$). If an invoked argument does not exist, a null string results. For example, the macro xx is defined by

```
.de xx  \"begin definition
Today is \\$1 the  \\$2.
..              \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that the **\$** is concealed in the definition with a prepended \. The number of currently available arguments is in the .$ register.

No arguments are available at the top (nonmacro) level in this implementation. Because string referencing is implemented as an input-level push down, no arguments are available from within a string. No arguments are available within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a long string that is interpolated at copy time. It is advisable to conceal string references with an extra \ to delay interpolation until argument reference time.

6.4. Diversions

Processed output can be diverted into a macro for purposes such as footnote processing (Section 21.5) 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** contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in nofill mode, regardless of the current V. Constant-spaced (**cs**) or bold (**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** requests with the transparent mechanism described in Section 9.6.

Diversions can be nested, and certain parameters and registers are associated with the current diversion level. The top nondiversion level can be thought of as the Øth diversion level. These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (**mk** and **rt**), the current vertical place (**.d** register), the current text base-line (**.h** register), and the current diversion name (**.z** register).

6.5. Traps

Three types of trap mechanisms are available: page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps can be planted using **wh** at any page position including the top. This trap position is changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved 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 unless and until the first one is moved (Appendix). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line

of text is output whose vertical size reaches or sweeps past the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion can be planted using **dt**. The .t register works in a diversion; if there is no subsequent trap, a large distance is returned. The following table describes the input-line-count traps:

Request Form	Initial Value	If No Argument
.de xx yy	-	.yy=..

Explanation: Define or redefine the macro xx. The contents of the macro begin on the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with .yy, whereupon the macro yy is called. In the absence of yy, the definition is terminated by a line beginning with two periods (..). A macro can contain **de** requests, provided the terminating macros differ or the contained definition terminator is concealed. The .. can be concealed as **\\..** (which copies as **\..**) and be reread as .. itself.

Request Form	Initial Value	If No Argument
.am xx yy	-	.yy=..

Explanation: Append to macro (append version of **de**).

Request Form	Initial Value	If No Argument
.ds xx string	-	ignored

Explanation: Define a string xx containing string. Any initial double quote in string is stripped off to permit initial blanks.

Request Form	Initial Value	If No Argument
.as xx string	-	ignored

Explanation: Append string to string xx (append version of **ds**).

Request Form	Initial Value	If No Argument
.rm xx	-	ignored

Explanation: Remove request, macro, or string. The name xx is removed from the name list and any related storage space is freed. Subsequent references have no effect.

Request Form	Initial Value	If No Argument
.rn xx yy	-	ignored

Explanation: Rename request, macro, or string xx to yy. If yy exists, it is first removed.

Request Form	Initial Value	If No Argument
.di xx	-	end

Explanation: 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 the current diversion level.

Request Form	Initial Value	If No Argument
.da xx	-	end

Explanation: Divert, appending to xx (append version of **di**). Mode or relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
.wh N xx	-	-

Explanation: Install a trap to invoke xx at page position; 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, if any, is removed. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.ch xx N	-	-

Explanation: Change the trap position for macro xx to be N. In the absence of N, the trap, if any, is removed. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.dt N xx	-	off

Explanation: 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 the current diversion level. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.it N xx	-	off

Explanation: Set an input-line-count trap to invoke the macro xx after N lines of text input have been read (control or request lines do not count). The text can be in-line text or text interpolated by in-line or trap-invoked macros. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.em xx	none	none

Explanation: The 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.

SECTION 7
NUMBER REGISTERS

A variety of parameters are available to the user as predefined, named number registers (Summary and Index). In addition, named registers can be user-defined. Register names are one or two characters long and do not conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register 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 **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified if accessed with an auto-incrementing sequence. If the registers x and xx both contain N and have the auto-increment size M, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
Ø	none	N
Ø	none	N
\n+x	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 decremented by M	N-M

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lowercase Roman, uppercase Roman, lowercase sequential alphabetic, or uppercase sequential alphabetic, according to the format specified by **af**.

Request Form	Initial Value	If No Argument
.nr R±N	M	-

Explanation: The number register R is assigned the value ±N with respect to the previous value, if any. The increment for auto-incrementing is set to M. The default scale indicator is u (ignored if not specified).

Request Form	Initial Value	If No Argument
.af R c	arabic	-

Explanation: Assign format c to register R. The available formats are:

Numbering Format	Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having N digits specifies a field width of N digits (second example above). The read-only registers and the width function are always arabic.

Request Form	Initial Value	If No Argument
.rr R	-	ignored

Explanation: Remove register R. If many registers are being created dynamically, it is necessary to remove unused registers to recapture internal storage space for newer registers.

SECTION 8
TABS, LEADERS, AND FIELDS

8.1. Tabs and Leaders

The ASCII horizontal tab character and the ASCII SOH (leader character) can both generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal tab stops specified with **ta**. The default difference is that tabs generate motion, and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops: left adjusting, right adjusting, and centering. In the following table, D is the distance from the current position on the input line (where a tab or leader was found) to the next tab stop; next-string consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; W is the width of next-string.

Tab Type	Length of Motion or Repeated Characters	Location of Next-String
Left	D	Following D
Right	D-W	Right adjusted with D
Centered	D-W/2	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but can be used as next-string terminators.

Tabs and leaders are not interpreted in copy mode. **\t** and **\a** always generate a noninterpreted tab and leader respectively, and are equivalent to actual tabs and leaders in copy mode.

8.2. Fields

A field is contained between a pair of field delimiter characters, and 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 is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #^xxx^right# specifies a right-adjusted string with the string xxx centered in the remaining space.

Request Form	Initial Value	If No Argument
.ta Nt ...	**Ø.8; Ø.5 in**	**none**

Explanation: Set tab stops and types. t=R, right adjusting; t=C, centering; t absent, left adjusting. Troff tab stops are preset every Ø.5in.; nroff every Ø.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.tc c	none	none

Explanation: The tab repetition character becomes c, or is removed specifying motion. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.lc c		none

Explanation: The leader repetition character becomes c, or is removed specifying motion. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.fc a b	off	off

Explanation: The field delimiter is set to a; the padding indicator is set to the space character or to b, if given. In the absence of arguments, the field mechanism is turned off.

SECTION 9
INPUT/OUTPUT CONVENTIONS AND CHARACTER TRANSLATIONS

9.1. Input Character Translations

The new line 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 **tr**. All others are ignored.

The escape character **(\)** introduces escape sequences and causes the following character to mean another character, or to indicate some function. (A complete list of such sequences is given in the Summary and Index.) The \ is not the ASCII control character ESC of the same name. The escape character can be input with the sequence ****. The escape character can be changed with **ec**, and all that has been said about the default \ becomes true for the new escape character. **\e** prints whatever the current escape character is. If necessary or convenient, the escape mechanism can be turned off with **eo**, and restored with **ec**.

Request Form	Initial Value	If No Argument
.ec c	\	\

Explanation: Set escape character to ****, or to c, if given.

Request Form	Initial Value	If No Argument
.eo	on	-

Explanation: Turn escape mechanism off.

9.2. Ligatures

Five ligatures are available in the current troff character set: **fi**, **fl**, **ff**, **ffi**, and **ffl**. They are input by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in troff, and automatically invokes ligatures during input.

Request Form	Initial Value	If No Argument
.lg N	off; on	on

Explanation: Ligature mode is turned on if N is absent or non-zero, and turned off if N=0. If N=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in copy mode. No effect in nroff.

9.3. Backspacing, Underlining, and Overstriking

Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion the width of the space character.

Nroff automatically underlines characters in the underline font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, Section 2.2). In addition to **ft** and **\fF**, the underline font is selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of reasonable characters.

Request Form	Initial Value	If No Argument
.ul N	off	N=1

Explanation: Underline in nroff (italicize in troff) the next N input text lines. Actually, switch to underline font, saving the current font for later restoration; other font changes within the span of a **ul** 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>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching prevents this. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.cu N	off	N=1

Explanation: A variant of **ul** that causes every character to be underlined in nroff. Identical to **ul** in troff. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.uf F	Italic	Italic

Explanation: Underline font set to F. In nroff, F may not be on position 1 (initially Times Roman).

9.4. Control Characters

Both the 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.

Request Form	Initial Value	If No Argument
.cc c	.	.

Explanation: The basic control character is set to c, or reset to . . Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.c2 c	'	'

Explanation: The nobreak control character is set to c, or reset to ' . Relevant parameters are a part of the current environment.

9.5. Output Translation

One character can be made a stand-in for another character using **tr**. All text processing takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	Initial Value	If No Argument
.tr abcd....	none	-

Explanation: Translate a into b, c into d, etc. 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.

9.6. Transparent Throughput

An input line beginning with a **\!** is read in copy mode and transparently output without the initial **\!**; the text processor makes no other response based on the line's presence. This mechanism is used to pass control information to a post-processor or to embed control lines in a macro created by a diversion.

9.7. Comments and Concealed New Lines

A long input line that must stay one line (for example, a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape ****. The sequence ****(new line) is always ignored--except in a comment. Comments can be embedded at the end of any line by prefacing them with **\"**. The new line at the end of a comment cannot be concealed. A line beginning with **\"** appears as a blank line and behaves like **.sp** 1; a comment can be placed on a line by itself by beginning the line with **.\"**.

SECTION 10
LOCAL MOTIONS AND WIDTH FUNCTION

10.1. Local Motions

The functions **\v'N'** and **\h'N'** are used for local vertical and horizontal motion respectively. The distance N can be negative; the positive directions are rightward and downward. A local motion is one contained within a line. To avoid unexpected vertical dislocations, it is necessary that the net vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion

Command	Effect in TROFF	Effect in NROFF
\v'N'	Move distance N	Move distance N
\u	1/2 em up	1/2 line up
\r	1 em up	1 line up
\d	1/2 em down	1/2 line down

Horizontal Local Motion

Command	Effect in TROFF	Effect in NROFF
\h'N'	Move distance N	Move distance N
\space	Unpaddable space-size space	
\Ø	Digit-size space	
\\|	1/6 em space	ignored
\^	1/12 em	ignored

1Ø.2. Width Function

The width function **\w'string'** generates the numerical width of string (in basic units). Size and font changes can be safely embedded in string, and do not affect the current environment. For example, **.ti|-\w'1.|'u** can temporarily indent leftward 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 (respectively) to the highest and lowest extent of string relative to the baseline; then, for example, the total height of the string is **\n(stu-\n(sbu**. In troff the number register **ct** is set to a value between Ø and 3. Ø means that all of the 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**); and 3 means that both tall characters and characters with descenders are present.

1Ø.3. Mark Horizontal Place

The escape sequence **\kx** causes the current horizontal position in the input line to be stored in register x. As an example, the construction **\kx**word**\h'|\nxu+2u'**word makes "word" bold by backing up to almost its beginning and overprinting it, resulting in word.

SECTION 11
OVERSTRIKE, LINE-DRAWING, AND ZERO-WIDTH FUNCTIONS

11.1. Overstriking

Automatically centered overstriking of up to nine characters is provided by the overstrike function (**\o** "string"). The characters in string are overprinted with centers aligned; the total width is that of the widest character. The string must not contain local vertical motion. For example, **\o'e\''** produces **è**.

11.2. Line Drawing

The function **\l** "Nc" draws a string of repeated c's for a distance N. (\l is \(lowercase L). If c looks like a continuation of an expression for N, it can be insulated from N with a **\&**. If c is not specified, the _ (baseline rule or underline character) is used. If N is negative, a backward horizontal motion of size N is made before drawing the string. Any space resulting from N/(size of c) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected, such as baseline-rule _, underrule _, and root-en, the remainder space is covered by 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

underlined words.

The function **\L'**Nc**'** draws a vertical line consisting of the (optional) character c stacked vertically apart 1em (1 line in nroff), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the box rule | (**\(br**); the other suitable character is the bold vertical | (**\(bv**). The line is begun without 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 1/2-em wide underrule were designed to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \"compensate for next automatic base-line spacing
.nf         \"avoid possibly overflowing word buffer
.fi
..
```

draws a box around some text whose beginning vertical place was saved in number register a (that is, using **.mk a**).

11.3. Zero-Width Characters

The function **\z**c outputs c without spacing over it, and is used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** produces ⊕, and **\(br\z\(rn\(ul\(br** produces the smallest possible constructed box.

SECTION 12
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, the user can specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) nonalphabetic strings are considered candidates for automatic hyphenation. Words that are input containing hyphens (minus), em-dashes (**\(em**), or hyphenation indicator characters, are always subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	Initial Value	If No Argument
.nh	hyphenate	-

Explanation: Automatic hyphenation is turned off. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.hy N	on,N=1on,	N=1

Explanation: Automatic hyphenation is turned on for N $\geq$1, or off for N=Ø. If N=2, last lines (ones that cause a trap) are not hyphenated. For N=4 and 8, the last and first two characters of a word are not split off. These values are additive; for example, N=14 invokes all three restrictions. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.hc c	\%	\%

Explanation: 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.

Request Form	Initial Value	If No Argument
.hw	word1 ...	ignored

Explanation: Specify hyphenation points in words with embedded minus signs. Versions of a word with terminal s are implied; for example, dig-it implies dig-its. This list is examined initially and after each suffix stripping. The space available is small--about 128 characters.

SECTION 13
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 specified with **lt**. **tl** can be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	Initial Value
.tl 'left'center'right' -	

Explanation: The strings left, center, and right are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings can 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. The format is assigned to register **%**. Any character can be used as the string delimiter.

Request Form	Initial Value	If No Argument
.pc c	%	off

Explanation: The page number character is set to c, or removed. The page-number register remains **%**.

Request Form	Initial Value	If No Argument
.lt ±N	6.5 in	previous

Explanation: Length of title set to ±N. The line-length and the title-length are independent. Indents do not apply to titles; page-offsets do. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

SECTION 14
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 prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length can be used to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are not numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later **.nm|+Ø**. In addition, a line number indent I, and the number-text separation S can be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number M are to be printed. The others appear as blank number fields.

Request Form	Initial Value	If No Argument
.nm ±N M S I	M=1, S=1, I=Ø	off

Explanation: Line number mode. If ±N is given, line numbering is turned on, and the next output line numbered is numbered ±N. Default values are M=1, S=1, and I=Ø. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.nn N	-	N=1

Explanation: The next N text output lines are not numbered. Relevant parameters are a part of the current environment.

As an example, the paragraph portions of this section are numbered with M=3: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +Ø** was placed in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w"ØØØØ"u**) to keep the right side aligned. Another example is **.nm +5 5 x 3**, which turns on

numbering with the line number of the next line to be 5
24 greater than the last numbered line, with M=5, with
spacing S untouched, and with the indent I set to 3.

SECTION 15
CONDITIONAL ACCEPTANCE OF INPUT

In the following, c is a one-character, built-in condition name, **!** signifies not, N is a numerical expression, string1 and string2 are strings delimited by any nonblank, non-numeric character not in the strings, and anything represents what is conditionally accepted.

Request
Form

.if c anything

Explanation: If condition c true, accept anything as input; in multi-line case use \{anything\}.

Request
Form

.if !c | anything

Explanation: If condition c false, accept anything.

Request
Form

.if N anything

Explanation: If expression N > 0, accept anything. The default scale indicator is u (ignored if not specified).

Request
Form

.if !N anything

Explanation: If expression N ≤ 0, accept anything. The default scale indicator is u (ignored if not specified).

Request
Form

.if 'string1'string2'anything

Explanation: If string1 is identical to string2, accept anything.

Request
Form

.if !'string1'string2'anything

Explanation: If string1 is not identical to string2, accept anything.

Request
Form

.ie c anything

Explanation: If portion of if-else; all above forms (like **if**). The default scale indicator is u (ignored if not specified).

Request
Form

.el anything

Explanation: Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is troff
n	Formatter is nroff

If the condition c is true, or if the number N is greater than zero, or if the strings compare identically (including motions and character size and font), anything is accepted as input. If a **!** precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of anything are skipped over. The anything is either a single input line (for example, text or macro) 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 **(\})**.

The request **ie** (if-else) is identical to **if**, except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie**|-|**el** pairs can be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp Ø.5i
.tl 'Page %'''
'sp |1.2i \}
.el .sp|2.5i
```

which treats page 1 differently from other pages.

SECTION 16
ENVIRONMENT SWITCHING

A number of the parameters that control the text processing are gathered together into an environment that can be switched by the user. The environment parameters are those associated with requests noting E in their Notes column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	Initial Value	If No Argument
.ev N	N=Ø	previous

Explanation: Environment switched to environment Ø $\leq$ N $\leq$ 2. Switching is done in push-down fashion so that restoring a previous environment must be done with **.ev** rather than specific reference.

SECTION 17
INSERTIONS FROM THE STANDARD INPUT

The input can be temporarily switched to the system standard input with **rd**, which switches back when two new lines in a row are found (the extra blank line is not used). This mechanism is intended for insertions in form-letter types of documentation. On ZEUS, the standard input is the user's keyboard, a pipe, or a file.

Request Form	Initial Value	If No Argument
.rd prompt	-	prompt=BEL

Explanation: Read insertion from the standard input until two new lines in a row are found. If the standard input is the user's keyboard, prompt (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments can be placed after prompt.

Request Form	Initial Value	If No Argument
.ex	-	-

Explanation: Exit from nroff/troff. Text processing is terminated exactly as if all input had ended.

If insertions are 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 prompts 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 are prepared by entering the insertions for all the copies in one file used as the standard input, and causing the file containing the letter to reinvoke itself using **nx**; the process is ended by an **ex** in the insertion file.

SECTION 18
INPUT/OUTPUT FILE SWITCHING

Request Form	Initial Value	If No Argument
.so	filename	-

Explanation: Switch source file. The top input (file reading) level is switched to filename. A so encountered in a macro does not take effect until the input level returns to the file level. When the new file ends, input is again taken from the original file. so's can be nested.

Request Form	Initial Value	If No Argument
.nx filename		**end-of-file**

Explanation: Next file is filename. The current file is considered ended, and the input is immediately switched to filename.

Request Form	Initial Value	If No Argument
.pi program	-	

Explanation: Pipe output to program (nroff only). This request must occur before any printing occurs. No arguments are transmitted to program.

SECTION 19
MISCELLANEOUS

Request Form	Initial Value	If No Argument
.mc c N	-	off

Explanation: 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 nofill mode) the character is appended to the line. If N is not given, the previous N is used. The initial N is Ø.2 inches in nroff and 1 em in troff. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.tm string	-	newline

Explanation: After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user's terminal.

Request Form	Initial Value	If No Argument
.ig yy	-	.yy=..

Explanation: Ignore input lines. Ig behaves like **de** except that the input is discarded. The input is read in copy mode, and any auto-incremented registers are affected.

Request Form	Initial Value	If No Argument
.pm t	-	all

Explanation: Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if t is given, only the total of the sizes is printed. The size is given in blocks of 128 characters.

Request Form	Initial Value	If No Argument
.fl	-	-

Explanation: Flush output buffer. Used in interactive debugging to force output. This request normally causes a break.

SECTION 2Ø
OUTPUT AND ERROR MESSAGES

The output from **tm**, **pm**, and the prompt from **rd**, as well as various error messages are written onto the standard message output. The standard message output is different from the standard output, where nroff formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various error conditions can occur during the operation of nroff and troff. Certain less serious errors that have only local impact do not cause processing to terminate. Two examples are word overflow, caused by a word that is too large to fit into the word buffer (in fill mode), and line overflow, caused by an output line that grows too large to fit in the line buffer; in both cases, a message is printed, the excess is discarded, and the affected word or line is marked at the point of truncation with a * in nroff and a <= in troff. Processing continues, if possible, since output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

SECTION 21
EXAMPLES

21.1. Introduction

It is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into nroff and troff. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations. (Most documents can be prepared with either the -ms or -man macro sets.)

The following examples are intended to be useful and realistic, but do not cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers are used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization.

21.2. Page Margins

Header and footer macros are defined to describe the top and bottom page margin areas. A trap is planted at page position Ø for the header, and at -N (N from the page bottom) for the footer. The simplest such definitions are

```
.de hd          \"define header
'sp li
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh Ø hd
.wh -li fo
```

which provide blank one-inch top and bottom margins. The header only occurs on the first page if the definition and trap exist prior to the initial pseudo-page transition. In fill mode, the output line that springs the footer trap is forced out because some part or whole word does not fit on it. If anything in the footer and header that follows causes a break, that word or part word is forced out. In this and other examples, requests like **bp** and **sp**, which

normally cause breaks, are invoked using the no-break control character ' to avoid this problem. When the header/footer design contains material requiring independent text processing, the environment can be switched, avoiding most interaction with the running text.

Another example is

```
.de hd                          \"header
.if t .tl '\(rn''\(rn'          \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i-1                     \"tl base at 0.5i
.tl ''- % -''                   \"centered page number
.ps                             \"restore size
.ft                             \"restore font
.vs  \}                         \"restore vs
'sp |1.0i                       \"space to 1.0i
.ns                             \"turn on no-space mode
..
.de fo                          \"footer
.ps 10                          \"set footer/header size
.ft R                           \"set font
.vs 12p                         \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu-0.5i-1             \"tl base 0.5i up
.tl ''- % -'' \}                \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If troff is used, a cut mark is drawn in the form of root-ens at each margin. The **sps** refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing sweeps past the trap position by as much as the base-line spacing. The no-space mode is turned on at the end of **hd** to render ineffective and accidental occurrences of **sp** at the top of the running text.

This 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 restore both the current and previous values for size as shown in the following:

```
.de fo
.nr s1 \\n(.s              \"current size
.ps
.nr s2 \\n(.s              \"previous size
. ---                      \"rest of footer
..
.de hd
. ---                      \"header
.ps \\n(s2                 \"restore previous size
.ps \\n(s1                 \"restore current size
..
```

Page numbers are 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 -Ø.5i-1v bn            \"tl base Ø-5i up
```

21.3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph is collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for more than one line, and requests a temporary indent.

```
.de pg                  \"paragraph
.br                     \"break
.ft R                   \"force font,
.ps 1Ø                  \"size,
.vs 12p                 \"spacing,
.in Ø                   \"and indent
.sp Ø.4                 \"prespace
.ne 1+\\n(.Vu           \"want more than 1 line
.ti Ø.2i                \"temp indent
..
```

The first break in **pg** forces out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is a defense against prior error and permits things like section heading macros to set parameters only once. The prespacing parameter is suitable for troff; a larger space, at least as big as the output device vertical resolution, is more suitable in nroff. The choice of remaining space to test for in ne is the smallest amount greater than one line.

A macro to automatically number section headings looks like:

```
.de sc                      \"section
. ---                       \"force font, etc.
.sp Ø.4                     \"prespace
.ne 2.4+\\n(.Vu             \"want 2.4+ lines
.fi
\\n+S.
..
.nr S Ø 1                   \"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, Ø.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 is set by **af**.

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp                      \"labeled paragraph
.pg
.in Ø.5i                    \"paragraph indent
.ta Ø.2i Ø.5i               \"label, paragraph
.ti Ø
\t\\$1\t\c                  \"flow into paragraph
..
```

The intended usage is "**.lp** label"; label begins at Ø.2 inch, and cannot exceed a length of Ø.3 inch without intruding into the paragraph. The label is right-adjusted against Ø.4 inch by setting the tabs instead with **.ta Ø.4iR Ø.5i**. The last line of **lp** ends with **\c** so that it becomes a part of the first line of the text that follows.

21.4. Multiple Column Output

The production of multiple column pages requires the footer macro to determine whether it was invoked by other than the last column, so that it begins a new column rather than produce the bottom margin. The header initializes a column register that the footer increments and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd                  \"header
. ---
.nr cl Ø 1              \"init column count
.mk                     \"mark top of text
..
.de fo                  \"footer
.ie \\n+(cl<2 \{\
.po +3.4i               \"next column; 3.1+Ø.3
.rt                     \"back to mark
.ns \}                  \"no-space mode
.el \{\
.po \\nMu               \"restore left margin
. ---
'bp \}
..
.ll 3.1i                \"column width
.nr M \\n(.o            \"save left margin
```

Typically, a portion of the top of the first page contains full-width text; the request for the narrower line length, as well as another **.mk** is made where the two-column output begins.

21.5. Footnote Processing

The footnote mechanism is used by embedding the footnotes in the input text at the point of reference, demarcated by an initial **.fn** and a terminal **.ef**:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote does not completely fit in the available space.

```
.de hd                  \"header
. ---
.nr x 0 1               \"init footnote count
.nr y 0-\\nb            \"current footer place
.ch fo -\\nbu           \"reset footer trap
.if \\n(dn .fz          \"leftover footnote
..
.de fo                  \"footer
.nr dn 0                \"zero last diversion size
.if \\nx \{\
.ev 1                   \"expand footnotes in ev1
.nf                     \"retain vertical size
.FN                     \"footnotes
.rm FN                  \"delete it
.if "\\n(.z"fy" .di     \"end overflow diversion
.nr x 0                 \"disable fx
.ev  \}                 \"pop environment
. ---
'bp
..
.de fx                  \"process footnote overflow
.if \\nx .di fy         \"divert overflow
..
.de fn                  \"start footnote
.da FN                  \"divert (append) footnote
.ev 1                   \"in environment 1
.if \\n+x=1 .fs         \"if first, include separator
.fi                     \"fill mode
..
.de ef                  \"end footnote
.br                     \"finish output
.nr z                   \\n(.v\"save spacing
.ev                     \"pop ev
.di                     \"end diversion
.nr y -\\n(dn           \"new footer position,
.if                     \\nx=1 .nr y -(\\n(.v-\\nz) \
                          \"uncertainty correction
.ch fo \\nyu            \"y is negative
.if ( \\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v       \"it didn't fit
..
.de fs                  \"separator
\l'1i'                  \"1 inch rule
.br
..
.de fz                  \"get leftover footnote
.fn
.nf                     \"retain vertical size
.fy                     \"where fx put it
.ef
..
```

```
.nr b 1.Øi              \"bottom margin size
.wh Ø hd                \"header trap
.wh 12i fo              \"footer trap, temp position
.wh -\\nbu fx           \"fx at footer position
.ch fo -\\nbu           \"conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**. **y** is then decremented by the size of the footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments to prevent the late triggering of the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes are 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 **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading and finishing before reaching the **fx** trap.

A good exercise is to combine the multiple-column and footnote mechanisms.

21.6. Last Page

After the last input file has ended, nroff and troff invoke the end macro, if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. To start another page, use the end-macro

```
.de en                  \"end-macro
\c
'bp
..
.em en
```

to deposit a null partial word, and effect another last page.

APPENDIX A
SUMMARY AND INDEX

A.1. Summary

* Values separated by ; are for nroff and troff, respectively.

\# Notes are explained at the end of this Summary and Index.

† No effect in nroff.

‡ The use of ` as control character (instead of .) suppressess the break functon.

1. General Explanation

2. Font and Character Size Control

Request Form	Initial Value	If No Argument	Notes	Explanation
.ps±N	1Ø point	previous	E	Point size; also \s±N.
.ss N	12/36 em	ignored	E	Space-character size set to N/36 em.†
.cs F N M	off	-	P	Constant character space (width) mode (font F).†
.bd F N	off	-	P	Embolden font F by N-1 units.†
.bd S F N	off	-	P	Embolden Special Font when current font is F.†
.ft F	Roman	previous	E	Change to font F=x, xx, or 1-4. Also \fx,\f(xx, \fN.
.fp N F	R,I,B,S	ignored	-	Font named F mounted on physical position 1≤N≤4.

3. Page Control

Request Form	Initial Value	IfNo Argument	Notes	Explanation
.pl ±N	11in	11in	v	Page length.
.bp ±N	N=1	-	B,v	Eject current page; next page number N.
.pn ±N	N=1	ignored	-	Next page number N.
.po ±N	0;26/27in	previous	v	Page offset.
.ne N	-	N=1V	D,v	Need N vertical space (V=vertical spacing).
.mk	none	internal	D	Mark current vertical place in register R.
.rt ±N	none	internal	D,v	Return (upward only) to marked vertical place.

4. Text Filling, Adjusting, and Centering

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

5. Spacing

Request Form	Initial Value	If No Argument	Notes	Explanation
.vs N	1/6in;	previous	E,p	Vertical base line

	12pts			spacing (V).
.ls N	N=1	previous	E	Output N-1 Vs after each text output line.
.sp N	-	N=1V	B,v	Space vertical distance N in either direction.
.sv N	-	N=1V	B,v	Save vertical distance N.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
.ll ±N	6.5in	previous	E,m	Line length.
.in ±N	N=Ø	previous	B,E,m	Indent.
.ti ±N	-	ignored	B,E,m	Temporary indent.

6. Macros, Strings, Diversion, and Position Traps

Request Form	Initial Value	If No Argument	Notes	Explanation
.de xx yy	-	.yy=..	-	Define or redefine macro xx; end at call of yy.
.am xx yy	-	.yy=..	-	Append to a macro.
.ds xx string	-	ignored	-	Define a string xx containing string.
.as xx string	-	ignored	-	Append string to string xx.
.rm xx	-	ignored	-	Remove request, macro, or string.
.rn xx yy	-	ignored	-	Rename request, macro, or string xx to yy.
.di xx	-	end	D	Divert output to macro xx.
.da xx	-	end	D	Divert and append to xx.
.wh N xx	-	-	v	Set location trap; negative is with respect to page bottom.
.ch xx N	-	-	v	Change trap location.

.dt N xx	-	off	D,v	Set a diversion trap.
.it N xx	-	off	E	Set an input-line count trap.
.em xx	none	none	-	End macro is xx.

7. Number Registers

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

8. Tabs, Leaders, and Fields

Request Form	Initial Value	If No Argument	Notes	Explanation
.ta Nt ...	Ø.8;Ø.5in	none	E,m	Tab settings; left type, unless t=R(right), C(centered).
.tc c	none	none	E	Tab repetition character.
.lc c	.	none	E	Leader repetition character.
.fc a b	off	off	-	Set field delimiter a and pad character b.

9. Input and Output Conventions and Character Translations

Request Form	Initial Value	If No Argument	Notes	Explanation
.ec c	\	\	-	Set escape char-

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

1Ø. Hyphenation

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

11. Three Part Titles.

Request Form	Initial Value	If No Argument	Notes	Explanation
.tl	'left'center'right'-		-	Three-part title.
.pc c	%	off	-	Page number character.

Request Form	Initial Value	If No Argument	Notes	Explanation
.lt ±N	6.5in	previous	E,m	Length of title.

12. Output Line Numbering.

Request Form	Initial Value	If No Argument	Notes	Explanation
.nm ±N M S I		off	E	Number mode on or off, set parameters.
.nn N	-	N=1	E	Do not number next N lines.

13. Conditional Acceptance of Input

Request Form	Initial Value	If No Argument	Notes	Explanation
.if	c anything	-	-	If condition c true, accept anything as input, for multi-line use \{anything\}.
.if	!c anything	-	-	If condition c false, accept anything.
.if	N anything	-	u	If expression N>Ø, accept anything.
.if	!N anything	-	u	If expression N ≤Ø, accept anything.
.if	'string1' string2' anything	-	-	If string 1 identical to string2, accept anything.
.if	! 'string1 'string2'anything	-	-	If string1 not identical to string2, accept anything.
.ie	c anything	-	u	If portion of if-else; all above forms (like if).
.el	anything		-	- if-else.

14. Environment Switching

Request Form	Initial Value	If No Argument	Notes	Explanation
.ev N	N=Ø	previous	-	Environment switched (pushed down).

15. Insertions from the Standard Input

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

16. Input/Output File Switching

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

17. Miscellaneous

Request Form	Initial Value	If No Argument	Notes	Explanation
.mc c N	-	off	E,m	Set margin character c and separation N.
.tm string	-	newline	-	Print string on terminal (ZEUS stand ard message output).
.ig yy	-	.yy=..	-	Ignore till call

Request Form	Initial Value	If No Argument	Notes	Explanation of yy.
.pm t	-	all	-	Print macro names and sizes; if t present, print only total of sizes.
.fl	-	-	B	Flush output buffer.

18. Output and Error Messages

NOTES

B Request normally causes a break.

D Mode or relevant parameters associated with current diversion level.

E Relevant parameters are a part of the current environment.

O Must stay in effect until logical output.

P Mode must be still or again in effect at the time of physical output.

v,p,m,u Default scale indicator; if not specified, scale indicators are ignored.

A.2. Alphabetical Request and Section Number Cross Reference

Request Form			Initial Value		If No Argument		Notes		Explanation
ad	4	dt	6	ig	19	nn	14	rs	5
af	7	ec	9	in	5	nr	7	rt	3
am	6	ei	15	it	6	ns	5	so	18
as	6	em	6	lc	8	nx	18	sp	5
bd	2	eo	9	lg	9	os	5	ss	2
bp	3	ev	16	li	9	pc	13	sv	5
br	4	ex	17	ll	5	pi	18	ta	8
c2	9	fc	8	ls	5	pl	3	tc	8
cc	9	fi	4	lt	13	pm	19	ti	5
ce	4	fl	19	mc	19	pn	3	tl	13
ch	6	fp	2	mk	3	po	3	tm	19

cs	2	ft	2	na	4	ps	2	tr	9
cu	9	hc	12	ne	3	rd	17	uf	9
da	6	hw	12	nf	4	rm	6	ul	9
de	6	hy	12	nh	12	rn	6	vs	5
di	6	ie	15	nm	14	rr	7	wh	6
ds	6	if	15						

A.3. Escape Sequences for Characters, Indicators, and Functions

Section Reference	Escape Sequence	Meaning
9.1	\\	\ (to prevent or delay the interpretation of \)
9.1	\e	Printable version of the current escape character.
2.1	\'	\' (acute accent); equivalent to \(aa
2.1	\`	` (grave accent); equivalent to \(ga
2.1	\-	- Minus sign in the current font
6	\.	Period (dot) (see de)
10.1	\(space)	Unpaddable space-size space character
10.1	\0	Digit-width space
10.1	\\|	1/6 em narrow space character (zero-width in nroff)
10.1	\^	1/12 em half-narrow space character (zero width in nroff)
4.1	\&	Nonprinting, zero width character
9.6	\!	Transparent line indicator
9.7	\"	Beginning of comment
6.3	\$N	Interpolate argument $1 \leq N \leq 9$
12	\%	Default optional hyphenation character
2.1	\(xx 1	Character named xx
6.1	*x,\ (xx	Interpolate string x or xx
8.1	\a	Noninterpreted leader character
11.2	\b'abc...'	Bracket building function
4.2	\c	Interrupt text processing
10.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in nroff)
2.2	\fx,\f(xx,\fN	Change to font named x or xx or position N
10.1	\h'N'	Local horizontal motion; move right N (negative left)
10.3	\kx	Mark horizontal input place in register x

11.3	\l'Nc'	Horizontal line drawing function (optionally with c)
11.3	\L'Nc'	Vertical line drawing function (optionally with c)
8	\nx,\n(xx	Interpolate number register x or xx
11.1	\o'abc...'	Overstrike characters a, b, c, ...
4.1	\p	Break and spread output line
10.1	\r	Reverse 1 em vertical motion (reverse line in nroff)
2.3	\sN,\s±N	Point-size change function
8.1	\t	Noninterpreted horizontal tab
10.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in nroff)
10.1	\v'N'	Local vertical motion; move down N (negative up)
10.2	\w'string'	Interpolate width of string
5.2	\x'N'	Extra line-space function (negative before, positive after)
11.4	\zc	Print c with zero width (without spacing)
15	\{	Begin conditional input
15	\}	End conditional input
9.7	\(newline)	Concealed (ignored) new line
-	\X	X, any character not listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and (new line) are interpreted in copy mode (Section 7.2).

A.4. Predefined General Number Registers

Section Reference	Register Name	Description
3	%	Current page number
10.2	ct	Character type (set by width function)
6.4	dl	Width (maximum) of last completed diversion
6.4	dn	Height (vertical size) of last completed diversion
-	dw	Current day of the week (1-7)
-	dy	Current day of the month (1-31)
10.3	hp	Current horizontal place on input line
14	ln	Output line number
-	mo	Current month (1-12)
4.1	nl	Vertical position of last printed text base-line
10.2	sb	Depth of string below base line (generated by width function)
10.2	st	Height of string above base line (gen-

		erated by width function)
-	yr	Last two digits of current year

A.5. Predefined Read-Only Number Registers

Section Reference	Register Name	Description
6.3	$	Number of arguments available at thecurrent macro level
-	A	Set to 1 in troff if -a option used; always 1 in nroff
10.1	H	Available horizontal resolution in basic units
-	T	Set to 1 in nroff, if -T option used; always 0 in troff
10.1	V	Available vertical resolution in basic units
5.2	a	Post-line extra line-space most recently utilized using ex "N"
-	c	Number of lines read from current input file
6.4	d	Current vertical place in current diversion; equal to nl, if no diversion
2.2	f	Current font as physical quadrant (1-4)
4	h	Text base-line mark on current page or diversion
5	i	Current indent
5	l	Current line length
4	n	Length of text portion on previous output line
3	o	Current page offset
3	p	Current page length
2.3	s	Current point size
6.5	t	Distance to the next trap
4.1	u	Equal to 1 in fill mode and 0 in nofill mode
5.1	v	Current vertical line spacing
10.2	w	Width of previous character
-	x	Reserved version-dependent register
-	y	Reserved version-dependent register
6.4	z	Name of current diversion

APPENDIX B
SUMMARY OF RECENT CHANGES TO NROFF/TROFF

B.1. Command Line Options

Removed Options

+n, -n Use the -o option instead.

Modified Options

-sn As well as stopping the output every n pages, this option also causes the bell control character to be output to the terminal when stopping between pages. (In troff, a message, "page stop", is output to the terminal).

Additional Options

-h (nroff only) Use output tabs during horizontal spacing to speed up output as well as to reduce output byte count. Device tabs settings are assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-z Efficiently suppresses formatted output. Only message output will occur (from **.tm** requests and diagnostics).

-cname Use the compacted version of macro package name, if it exists. If it doesn't, nroff/troff will try the equivalent -mname option instead. This option should be used instead of **-m** because it makes nroff/troff execute significantly faster.

-kname Produce a compacted macro package from this invocation of nroff/troff. This option has no effect if no **.co** request is used in the nroff/troff input. Otherwise, the compacted output is produced in files **d.**name and **t.**name.

-un (nroff only) Set the emboldening factor (number of character overstrikes in nroff) for the third font position (bold) to be n (zero if n is missing).

B.2. Requests

Removed Requests

.li The transparent input mode request has been removed.

Modified Requests

.ad c The adjustment type indicator c may now also be a number obtained from the **.j** register (see Appendix B.3).

.so name The contents of file name will be interpolated at the point the **.so** request is encountered. Previously, the interpolation was done upon return to the file-reading input level.

.bd The emboldening request now works in nroff, and causes overprinting of bold characters. The default setting for font position 3 (the bold font) is 3 (causing each bold character to be printed 4 times in the same position). The -u command line option may be used to change the emboldening factor for the bold font.

Additional Requests

.ab text 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.

.co If the **-k** name command line option was given, compact the current state of nroff/troff. If the **-k** name wasn't used, **.co** has no effect.

.! cmd args The UNIX command cmd is executed and its output becomes nroff/troff input. The standard input for cmd is closed.

B.3. Additional Predefined Number Registers

.k Read-only. Contains the horizontal size of the text portion (without indent) of the current partially-collected output line, if any, in the current environment.

.j Read-only. Indicates the current adjustment mode and type. Can be saved and later given to the **.ad** request to restore a previous mode.

.P Read-only. Contains the value 1 if the current page is being printed, and is zero otherwise, i.e., if the current page did not appear in the -o option list.

.L Read-only. Contains the current line-spacing parameter (the value of the most recent **.ls** request).

.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.

.R Number of number registers that remain available for use.

.b Emboldening factor of the current font (nroff and troff).

B.4. Additional Escape Sequences

\jx,**\j(**xx Mark the current horizontal output position in register x or xx.

\gx,**\g(**xx Return the **.af -type** format of the register x or xx. Returns nothing if x (xx) has not yet been referenced.

B.5. New Feature - Compacted Macros

A. User Information

The time required to read a macro package by nroff/troff may be greatly lessened by using a pre-processed version of that macro package, called compacted macros. The compacted version of a macro package is completely equivalent to the non-compacted version, except that a compacted macro package cannot be read by the **.so** request.

A compacted version of a macro package, name, is used by the -cname command line option, while the uncompacted version is used by the -mname option. Because -cname defaults to -mname if the name macro package hasn't been compacted, the user should always use -c rather than -m.

The next section describes how to build a compacted version of a macro package.

B. Building a Compacted Macro Package

If one has a macro package and wishes to make a compacted version of it, the following simple steps should be followed:

1. Separate the Compactable from Non-compactable Parts

   Only the following nroff/troff entities can be compacted: macro, string, diversion, number register and trap settings and definitions. For example, the following are not compactable: environment settings, end macro setting, or any commands that interact (during package interpretation) with command line settings, (like a reference to \nP, which could be set on the command line).

   All the non-compactable material should be placed at the end of the macro package, with a **.co** request separating the compactable from non-compactable parts:

```
Compactable Material
        .
        .
        .
.co
Non-Compactable Material
        .
        .
        .
```

   The **.co** request indicates to nroff/troff when to compact its current internal state.

2. Produce Compacted Files

   Once compactable and non-compactable segments have been set up as above, nroff/troff may be run with the **-k**name option to build the compacted files.

For example, if the macro file produced by step 1 is called **mac**, then the following can be used to build the compacted files:

```
$ nroff -kmac mac
or
$ troff -kmac mac
```

Each of these commands causes nroff/troff to create two files in the current directory, **d.mac** and **t.mac**.

3. Install Compacted Files

The two compacted files produced in step 2 must be installed into the system macro library **(/usr/lib/macros)** with the proper names.

If the files were produced by nroff, **cmp.n.** must be prepended to their names. If produced by troff, **cmp.t.** must be prepended to their names.

Still assuming that the macro package with which we are dealing is called **mac**, the two (nroff) compacted files may be installed by:

```
$ cp d.mac /usr/lib/macros/cmp.n.d.mac
$ cp t.mac /usr/lib/macros/cmp.n.t.mac
```

4. Install Non-compactable Segment

The non-compactable segment from the original macro package must also be installed on the system as

/usr/lib/macros/ucmp.[nt].name

where **n** of **[nt]** means the nroff version, and **t** means the troff version. The non-compactable segment must be produced by hand, for example, by using the editor.

Again using the **mac** package as an example, the following could be used to install the (nroff) non-compactable segment:

```
$ ed mac
/^\.co$/+,$w /usr/lib/macros/ucmp.n.mac
```

C. Warnings

Compacted macro packages depend heavily on the particular version of nroff/troff that produced them. This means that each package needs to be compacted separately by both nroff and troff. It also means that any compacted macro packages must be recompacted when a new version of nroff or troff is installed.

If nroff/troff discovers that a macro package was produced by a different version of nroff/troff than that attempting to read it, the -c will be abandoned, and the equivalent -m option attempted instead.

If nroff/troff actually reads a compacted package that was produced by a different version of nroff/troff (because the version number of nroff/troff was not updated), very peculiar action will result.

B.6. Other New Features

A. nroff/troff now accepts multiple -m/-c macro package requests on the command line.

B. The ASCII **esc** and **bel** characters are treated as regular characters.

C. The third font position (bold) causes overprinting in nroff.

D. Hyphenation is off by default.

B.7. Notable Changes

A. The conditionally accepted part of an **.ie** or **.if** request is now completely ignored if the test failed, rather than being read in copy mode, as was previously the case.

B. The **.cu** request has been enhanced to provide up to about three lines of continuously underlined text, and the underlining is not lost when the **.cu** is used inside a diversion.

SOURCE CODE CONTROL SYSTEM USER'S GUIDE

Table of Contents

List of Illustrations

List of Tables

SECTION 1
INTRODUCTION

The Source Code Control System SCCS is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It provides facilities for storing, updating, and retrieving any version of a file of text, for controlling updating privileges to that file, for identifying the version of a retrieved file, and for recording who made each change, when and where it was made, and why. SCCS is a collection of programs that run under the ZEUS system.

This document, is a user's guide to SCCS. The following topics are covered:

- How to get started with SCCS.
- The scheme used to identify versions of text kept in an SCCS file.
- Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- Protection and auditing of SCCS files, including the differences between the use of SCCS by individual users on one hand, and groups of users on the other.

Neither the implementation of SCCS nor the installation procedure for SCCS are described here.

The Source Code Control System SCCS is a collection of commands developed under the UNIX Tm based PWB (Programmers Workbench) timesharing system that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, records who made each change, when and where it was made, and why. This is important in environments in which programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes

unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the changes; each set of changes is called a "delta".

This manual contains the following sections:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands, with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

SECTION 2
SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a PWB system, create files, and use one of the available text editors (see vi(1), ex(1), or ed(1) in the System 8000 ZEUS Reference Manual). A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions in Section 1 of the System 8000 ZEUS Reference Manual should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the SCCS IDentification string SID, composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2. Creating an SCCS File - The 'admin' Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pascal
fortran
cobol
```

```
algol
```

We wish to give custody of this file to SCCS. The following admin command (which is used to administer SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin  -ilang  s.lang
```

All SCCS files must have names that begin with "s.", hence, "s.lang". The -i keyletter, together with its value "lang", indicates that admin is to create a new SCCS file and initialize it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The admin command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below. In the following examples, this warning message is not shown, although it may actually be issued by the various command.

The file "lang" should be removed (because it can be easily reconstructed by using the get command, below):

```
rm  lang
```

2.3. Retrieving a File - The 'get' Command

The command:

```
get  s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
No id keywords (cm7)
```

This means that get retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is

created.

The above get command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the delta command (see below), the get command must be informed of your intention to do so. This is done as follows:

```
get  -e  s.lang
```

The **-e** keyletter causes get to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the p-file, that will be read by the delta command. The get command prints the same messages as before, except that the SID of the version to be created through the use of delta is also issued. For example the command:

```
get  -e  s.lang
```

outputs:

```
1.1
new delta  1.2
5 lines
```

The file "lang" may now be changed, for example, by:

```
ed  lang
29
$a
snobol
ratfor
.
w
43
q
```

2.4. Recording Changes - The 'delta' Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta  s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

```
comments? added more languages
```

Delta then reads the p-file, and determines what changes were made to the file "lang". It does this by doing its own get to retrieve the original version, and by applying diff(1) to the original version and the edited version.

NOTE

> All references of the form name(N) refer to an entry for the command name found in section N of the System 8000 ZEUS Reference Manual.

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", delta outputs:

```
No id keywords (cm7)
1.2
2 inserted
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

2.5. More about the 'get' Command

As we have seen:

```
get  s.lang
```

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

```
get  s.lang

get  -r1  s.lang

get  -r1.2  s.lang
```

The numbers following the **-r** keyletter are SIDs (see Section

2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the get command:

```
get  -e  -r2  s.lang
```

Because release 2 does not exist, get retrieves the latest version before release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to delta via the p-file. Get then outputs:

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version delta will create. If the file is now edited, for example, by:

```
ed  lang
43
/cobol/d
w
37
q
```

and delta executed:

```
delta  s.lang
comments? deleted cobol from list of languages
```

we will see, by delta's output, that version 2.1 is indeed created:

```
No id keywords (cm7)
2.1
Ø inserted
```

```
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6. The 'help' Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (col)
```

The string "col" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the help command:

```
help col
```

This produces the following output:

```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, help is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

SECTION 3
HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta, to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 3-1.

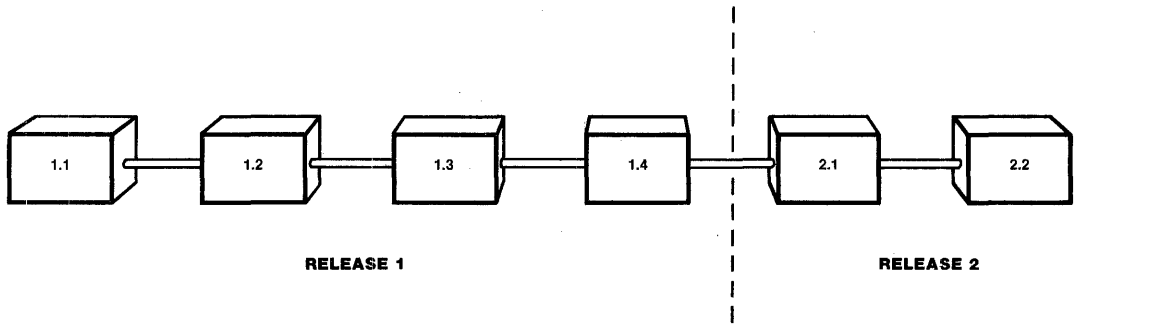


Figure 3-1 Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal sequential development of an SCCS file, in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree, in that changes applied as

part of a given delta are not dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 3-1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of four components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The sequence number is assigned, in order, to each delta on a "particular branch". Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 3-2.

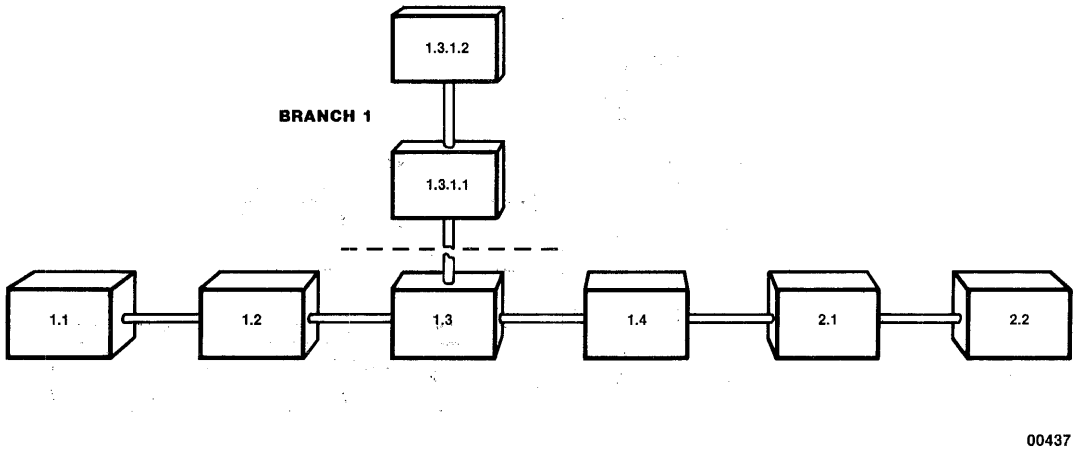


Figure 3-2 Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Figure 3-3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all of the

deltas between it and its trunk ancestor (1.3).

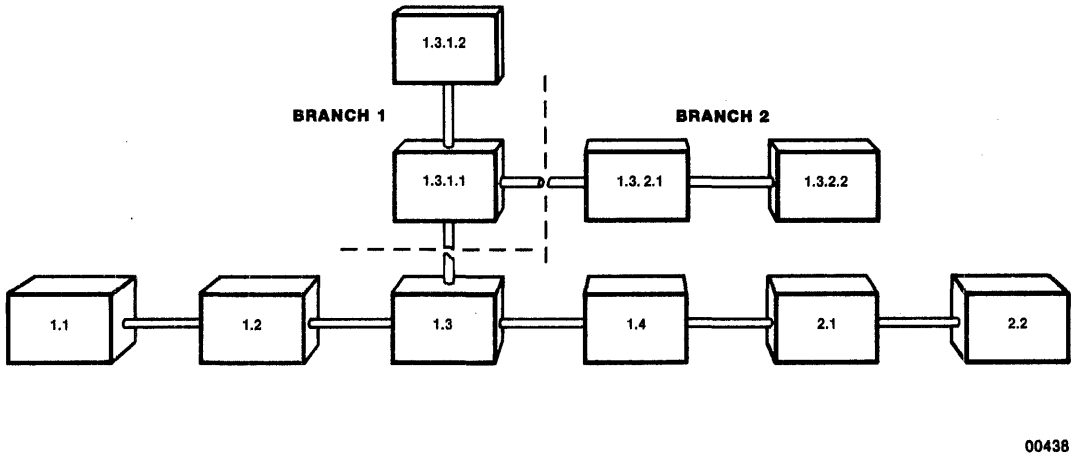


Figure 3-3 Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

SECTION 4
SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands, except as indicated below. SCCS commands accept two types of arguments: keyletter arguments and file arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (-), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable (because of permission modes (see chmod(1)) files in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name "-" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the find(1) or ls(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right.

Somewhat different argument conventions apply to the help, what, sccsdiff, and val commands (see Sections 5.5, 5.8, 5.9, and 5.11).

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see admin(1).

The distinction between the "real user" (see passwd(1)) and the "effective user" of a ZEUS system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a ZEUS system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the x-file, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the x-file is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the x-file is renamed to be the SCCS file. The x-file is created in the directory containing the SCCS file, is given the same mode (see chmod(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a lock-file, called the z-file, whose name is formed by replacing the "s." of the SCCS file name with "z.". The z-file contains the "process number" of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding z-file exists. The z-file is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore x-files and z-files; they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on standard output) of the form:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses may be used as an argument to the help command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

SECTION 5
SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the System 8000 ZEUS Reference Manual, and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

Because the commands get and delta are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get — Retrieves versions of SCCS files.

delta — Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.

admin — Creates SCCS files and applies changes to parameters of SCCS files.

prs — Prints portions of an SCCS file in user specified format.

help — Gives explanations of diagnostic messages.

rmdel — Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.

cdc — Changes the commentary associated with a delta.

what — Searches file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the get command.

sccsdiff — Shows the differences between any two versions of an SCCS file.

comb — Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.

val Validates an SCCS file.

sact Prints current SCCS file editing activity.

unget Undoes a previous get of an SCCS file.

5.1. get

The get command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the g-file; its name is formed by removing the "s." from the SCCS file name. The g-file is created in the current directory and is owned by the real user. The mode assigned to the g-file depends on how the get command is invoked, as discussed below.

The most common invocation of get is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).

2. This version has 67 lines of text.

3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated g-file (file "abc") is given mode 444 (read-only), since this particular way of invoking get is intended to produce g-files only for inspection, compilation, etc., and not for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

5.1.1. ID Keywords: In generating a g-file to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the g-file, so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. "Identification (ID) keywords" appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the g-file. Thus, executing get on an SCCS file that contains the PL/I declaration:

```
static char Version[] = "%M%  %I% %H%";
```

gives (for example) the following:

```
static char Version[] = "filename 2.3 7/07/77";
```

When no ID keywords are substituted by get, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by get, although the presence of the **i** flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see get(1).

5.1.2. Retrieval of Different Versions: Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **-r** keyletter of get.

The **-r** keyletter is used to specify an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get -r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a two- or four-component SID is specified as a value for the **-r** keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, get retrieves the trunk delta with the highest level number within the highest-

numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. If the given branch does not exist, an error message results. This might result in the following output:

```
4.3.2.8
89 lines
```

The -t keyletter is used to retrieve the latest ("top") version in a particular release (i.e., when no -r keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3. Retrieval with Intent to Make a Delta: Specification of the -e keyletter to the get command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes get to check:

1. The "user list" (which is the list of login names and/or "group IDs" of users allowed to make deltas (see Section 6.2)) to determine if the login name or group ID of the user executing get is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.

2. That the release (R) of the version being retrieved satisfies the relation:

   floor <= R <= ceiling

   to determine if the release being accessed is a protected release. The floor and ceiling are specified as flags in the SCCS file.

3. That the release (R) is not locked against editing. The lock is specified as a flag in the SCCS file.

4. Whether or not "multiple concurrent edits" are allowed for the SCCS file as specified by the j flag in the SCCS file (multiple concurrent edits are described in Section 5.1.5).

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the -e keyletter causes the creation of a g-file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g-file already exists, get terminates with an error. This is to prevent inadvertent destruction of a g-file that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the g-file are not substituted by get when the -e keyletter is specified, because the generated g-file is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, get does not need to check for the presence of ID keywords within the g-file, so that the message:

```
No id keywords (cm7)
```

is never output when get is invoked with the -e keyletter.

In addition, the -e keyletter causes the creation (or updating) of a p-file, which is used to pass information to the delta command (see Section 5.1.4).

The following is an example of the use of the -e keyletter:

```
get  -e  s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the -r and/or -t keyletters are used together with the -e keyletter, the version retrieved for editing is as specified by the -r and/or -t keyletters.

The keyletters -i and -x may be used to specify a list (see get(1) for the syntax of such a list) of deltas to be included and excluded, respectively, by get. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be not applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, get checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file (two deltas can interfere, for example, when each one changes the same line of the retrieved g-file). Any interference is indicated by a warning that shows the range of lines within the retrieved g-file in which the problem may exist. The user is expected to examine the g-file to determine whether a problem actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

The -i and -x keyletters should be used with extreme care.

The -k keyletter is provided to facilitate regeneration of a g-file that may have been accidentally removed or ruined subsequent to the execution of get with the -e keyletter, or to simply generate a g-file in which the replacement of ID keywords has been suppressed. Thus, a g-file generated by the -k keyletter is identical to one produced by get executed with the -e keyletter. However, no processing related to the p-file takes place.

5.1.4. Concurrent Edits of Different SIDs: The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of get commands with the -e keyletter may be executed on the same file, provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed, see Section 5.1.5).

The p-file (which is created by the get command invoked with the -e keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The p-file contains the following information for each delta that is still "in progress":

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing get.

The first execution of "get -e" causes the creation of the p-file for the corresponding SCCS file. Subsequent executions only update the p-file by inserting a line containing the above information. Before inserting this line, however, get checks that no entry already in the p-file specifies as already retrieved the SID of the version to be retrieved, unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of get should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a writable g-file, which is an SCCS error condition. In practice, such multiple executions are performed by different users, (see Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files) so that this problem does not arise, since each user normally has a different working directory.

Table 5-1 shows, for the most useful cases, what version of an SCCS file is retrieved by get, as well as the SID of the version to be eventually created by delta, as a function of the SID specified to get.

Table 5-1. Determination of New SIDs

Case	SID Specified @	-b Keyletter Used %	Other Conditions	SID Retrieved	SID of Delta to be Created
1.	none #	no	R defaults to mR	mR.mL	mR.(mL+1)
2.	none #	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
3.	R	no	R > mR	mR.mL	R.1 &
4.	R	no	R = mR	mR.mL	mR.(mL+1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
7.	R	-	R < mR and R does not exist	hR.mL **	hR.mL.(mB+1).1
8.	R	-	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1
9.	R.L	no	No trunk successor	R.L	R.(L+1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
11.	R.L	-	Trunk successor in release > R	R.L	R.L.(mB+1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)

15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
16.	R.L.B.S	-	Branch successor	R.L.B.S	R.L.(mB+1).1

@ "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

% The **-b** keyletter is effective only if the **b** flag (see admin(1)) is present in the file. In this table, an entry of "-" means "irrelevant".

This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

& This case is used to force the creation of the first delta in a new release.

** "hR" is the highest existing release that is lower than the specified, nonexistent, release.

5.1.5. Concurrent Edits of the Same SID: Under normal conditions, gets for editing (**-e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, delta must be executed before a subsequent get for editing is executed at the same SID as the previous get. However, multiple concurrent edits (defined to be two or more successive executions of get for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get  -e  s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of delta. In this case, a delta command corresponding to the first get produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the delta command corresponding to the second get produces delta 1.1.1.1.

5.1.6. Keyletters That Affect Output: Specification of the **-p** keyletter causes get to write the retrieved text to the standard output, rather than to a g-file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create g-files with arbitrary names:

```
get  -p  s.abc  >  arbitrary-filename
```

The **-s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to standard error. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the **-p** keyletter to "pipe" the output of get, as in:

```
get  -p  -s  s.abc  |  nroff
```

The **-g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get  -g  -r4.3  s.abc
```

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the **-g** keyletter is in regenerating a p-file that may have been accidentally destroyed:

```
get  -e  -g  s.abc
```

The **-l** keyletter causes the creation of an l-file, which is named by replacing the "s." of the SCCS file name with "l.".

This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in get(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get -r2.3 -l s.abc
```

generates an l-file showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the -l keyletter, as in:

```
get -lp -r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the l-file. Note that the -g keyletter may be used with the -l keyletter to suppress the actual retrieval of the text.

The -m keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated g-file to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The -n keyletter causes each line of the generated g-file to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The -n keyletter is most often used in a pipeline with grep(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the -m and -n keyletters are specified, each line of the generated g-file is preceded by the value of the %M% ID keyword and a tab (this is the effect of the -n keyletter), followed by the line in the format produced by the -m keyletter. Because use of the -m keyletter and/or the -n keyletter causes the contents of the g-file to be modified, such a g-file must not be used for creating a delta. Therefore, neither the -m keyletter nor the -n keyletter may be specified together with the -e keyletter.

See get(1) for a full description of additional get keyletters.

5.2. delta

The delta command is used to incorporate the changes made to a g-file into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the delta command requires the existence of a p-file (see Sections 5.1.3 and 5.1.4). Delta examines the p-file to verify the presence of an entry containing the user's login name. If none is found, an error message results. Delta also performs the same permission checks that get performs when invoked with the **-e** keyletter. If all checks are successful, delta determines what has been changed in the g-file, by comparing it (via diff(1)) with its own, temporary copy of the g-file as it was before editing. This temporary copy of the g-file is called the d-file (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal get at the SID specified in the p-file entry.

The required p-file entry is the one containing the login name of the user executing delta, because the user who retrieved the g-file must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed get with the **-e** keyletter more than once on the same SCCS file), the **-r** keyletter must be used with delta to specify an SID that uniquely identifies the p-file entry. The SID specified may be either the SID retrieved by get, or the SID delta is to create. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of delta is:

```
delta  s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a newline character. The user's response may be up to 512 characters long, with newlines not intended to terminate the response escaped by "\".

If the SCCS file has a **v** flag (see Section 5.3.2), delta first prompts with:

```
MRs?
```

on the standard output (again, this prompt is printed only if the standard output is a terminal). The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

The **-y** and/or **-m** keyletters are used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input. For example:

```
delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when delta is executed from within a "shell procedure" (see sh(1)).

The commentary (comments and/or MR numbers), whether solicited by delta or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to all SCCS files processed by the same invocation of delta. This implies that if delta is invoked with more than one file argument, and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, delta outputs (on the standard output) the SID of the created delta (obtained from the p-file entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by delta do not agree with the user's perception of the changes applied to the g-file. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the g-file, and delta is likely to find

a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited g-file.

If, in the process of making a delta, delta finds no ID keywords in the edited g-file, the message:

```
No id keywords (cm7)
```

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a g-file that was created by a get without the **-e** keyletter (recall that ID keywords are replaced by get in that case), or by accidentally deleting or changing the ID keywords during the editing of the g-file. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an i flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding p-file entry is removed from the p-file. All updates to the p-file are made to a temporary copy, the q-file, whose use is similar to the use of the x-file, which is described in Section 4 above. If there is only one entry in the p-file, then the p-file itself is removed.

In addition, delta removes the edited g-file, unless the **-n** keyletter is specified. Thus:

```
delta  -n  s.abc
```

will keep the g-file upon completion of processing.

The **-s** ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes delta neither to read the standard input nor to write the standard output.

The differences between the g-file and the d-file (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by diff(1).

5.3. admin

The admin command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the admin command upon that file.

5.3.1. Creation of SCCS Files: An SCCS file may be created by executing the command:

```
admin -ifirst s.abc
```

in which the value ("first") of the **-i** keyletter specifies the name of a file from which the text of the initial delta of the SCCS file "s.abc" is to be taken. Omission of the value of the **-i** keyletter indicates that admin is to read the standard input for the text of the initial delta. Thus, the command:

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by admin as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the -i keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the **-i** keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **-r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **-i** keyletter.

5.3.2. Inserting Commentary for the Initial Delta: When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments using the **-y** keyletter and/or MR numbers using the **-m** keyletter (the creation of an SCCS file may sometimes be the direct result of an MR) in exactly the same manner as for delta. If comments **(-y** keyletter) are omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers **(-m** keyletter), the **v** flag must also be set (using the **-f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" (see sccsfile(5)) in the SCCS file. Thus:

```
admin -ifirst -mmrnuml -fv s.abc
```

Note that the **-y** and **-m** keyletters are only effective if a new SCCS file is being created.

5.3.3. Initialization & Modification of SCCS File Parameters: The portion of the SCCS file reserved for "descriptive text" (see Section 6.2) may be initialized or changed through the use of the **-t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the **-t** keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an existing SCCS file, the -t keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the -t keyletter as in:

```
admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags (see Section 6.2) of an SCCS file may be initialized and changed, or deleted through the use of the -f and -d keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See admin(1) for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the get command. The -f keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fi -fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the get command will use to replace the **%M%** ID keyword. In the absence of the **m** flag, the name of the g-file is used as the replacement for the **%M%** ID keyword. Note that several -f keyletters may be supplied on a single invocation of admin, and that -f keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The -d keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several -d keyletters may be supplied on a single invocation of admin, and may be intermixed with -f keyletters.

SCCS files contain a list ("user list") of login names and/or group IDs of users who are allowed to create deltas

(see Sections 5.1.3 and 6.2). This list is empty by default, which implies that anyone may create deltas. To add login names and/or group IDs to the list, the **-a** keyletter is used. For example:

```
admin -axyz -awql -a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **-a** keyletter may be used whether admin is creating a new SCCS file or processing an existing one, and may appear several times. The **-e** keyletter is used in an analogous manner if one wishes to remove ("erase") login names or group IDs from the list.

5.4. prs

Prs is used to print on the standard output all or parts of an SCCS file (see Section 6.2) in a format, called the output "data specification", supplied by the user via the **-d** keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with "get ID" keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently being processed, and **:C:** is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see prs(1).

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the **-r** keyletter. For example:

```
prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc:  1.4 comment line is: THIS IS A COMMENT
```

If the -r keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the -l or -e keyletters. The -e keyletter substitutes data keywords for the SID designated via the -r keyletter and all deltas created earlier. The -l keyletter substitutes data keywords for the SID designated via the -r keyletter and all deltas created later. Thus, the command:

```
prs  -d:I:  -r1.4  -e  s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command:

```
prs  -d:I:  -r1.4  -l  s.abc
```

may produce:

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the -e and -l keyletters.

5.5. help

The help command prints explanations of SCCS commands and of messages that these commands may print. Arguments to help, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given,

help prompts for one. Help has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces:

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
   rmdel -rSID file ...
```

5.6. rmdel

The rmdel command is provided to allow removal of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The -r keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, rmdel checks that the release number (R) of the given SID satisfies the relation:

```
floor <= R <= ceiling
```

Rmdel also checks that the SID specified is not that of a version for which a get for editing has been executed and whose associated delta has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified can not be locked against editing (i.e., if the l flag is set (see admin(1)), the release specified must not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7. cdc

The cdc command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the rmdel command, except that the delta to be processed is not required to be a leaf delta. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by cdc in the same manner as that of delta. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing cdc and the time of its execution.

Cdc also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted
          correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

5.8. what

The what command is used to find identifying information within any ZEUS file whose name is given as an argument to what. Directory names and a name of "-" (a lone minus sign) are not treated specially, as they are by other SCCS commands, and no keyletters are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the **%Z%** ID keyword (see get(1)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), newline, or (non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char  id[] = "%Z%%M%:%I%";
```

and then the command:

```
get  -r3.4  s.prog.c
```

is executed, and finally the resulting g-file is compiled to produce "prog.o" and "a.out", then the command:

```
what  prog.c  prog.o  a.out
```

produces:

```
prog.c:
     prog.c:3.4
prog.o:
     prog.c:3.4
a.out:
     prog.c:3.4
```

The string searched for by what need not be inserted via an ID keyword of get; it may be inserted in any convenient manner.

5.9. sccsdiff

The sccsdiff command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are

specified by using the -r keyletter, whose format is the same as for the get command. The two versions must be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the pr(1) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are not acceptable to sccsdiff.

The differences are printed in the form generated by diff(1). The following is an example of the invocation of sccsdiff:

```
sccsdiff -r3.4 -r5.6 s.abc
```

5.10. comb

Comb generates a "shell procedure" (see sh(1)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is not recommended that comb be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, comb preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The -p keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The -c keyletter specifies a list (see get(1) for the syntax of such a list) of deltas to be preserved. All other deltas are discarded.

The **-s** keyletter causes the generation of a shell procedure, which, when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that comb be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by comb is not guaranteed to save any space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

5.11. val

Val is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

Val checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag respectively (see admin(1) for a description of the flags).

Val treats the special argument "-" differently from other SCCS commands (see Section 4). This argument allows val to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end-of-file. This capability allows for one invocation of val with different values for the keyletter and file arguments. For example:

```
val -
-yc -mabc s.abc
-mxyz -ypll s.xyz
```

first checks if file "s.abc" has a value "c" for its type flag and value "abc" for the "module name" flag. Once processing of the first file is completed, val then processes the remaining files, in this case "s.xyz", to determine if they meet the characteristics specified by the keyletter arguments associated with them.

Val returns an 8-bit code which is a disjunction of the possible errors detected. That is, each bit set indicates the occurrence of a specific error (see val(1) for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the

-s keyletter. A return code of "Ø" indicates all named files met the characteristics specified.

5.12. sact

Sact checks for SCCS files currently being edited. This means that a "get -e" was performed without a subsequent execution of delta. For example, if the SCCS file, s.abc was being edited, the command

```
sact s.abc
```

would output information similar to the following (the data is from the p-file):

```
1.1 1.2 bill 82/11/24 15:Ø2:ØØ
```

The first field is the SID of the latest delta made for s.abc, the second field specifies the SID for the new delta, the third field contains the logname of the user who performed the "get -e", the fourth and fifth fields contain the date and time that the "get -e" was executed.

5.13. unget

If a "get -e" has been executed on an SCCS file, this can be "undone" by the unget command. Of course, this is executed before the delta command. The simplest form of the unget command is:

```
unget s.abc
```

The program responds with the SID of the delta that would have been created. So, for the above example, unget would output

```
1.2
```

The keyletter, **-s**, suppresses the output of the intended delta's SID. Another keyletter, **-n**, retains the file being edited; otherwise, the file is removed. A specific SID can be indicated by using the **-r** keyletter.

SECTION 6
SCCS FILES

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

6.1. Protection

SCCS relies on the capabilities of the ZEUS operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the "release lock" flag, the "release floor" and "ceiling" flags, and the "user list" (see Section 5.1.3).

New SCCS files created by the admin command are given mode 444 (read only). It is recommended that this mode not be changed, as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755, which allows only the owner of the directory to modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files (see Section 6.3). The contents of directories should correspond to convenient logical groupings, e.g., sub-systems of a large project.

SCCS files must have only one link (name). The reason for this is that those commands that modify SCCS files do so by creating a temporary copy of the file (called the x-file, see Section 4) and, upon completion of processing, remove the old file and rename the x-file. If the old file has more than one link, removing it and renaming the x-file would break the link. Rather than process such files, SCCS commands produce an error message. All SCCS files must have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly

without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the admin command). This user is termed the "SCCS administrator" for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the get, delta, and, if desired, rmdel and cdc commands.

The interface program must be owned by the SCCS administrator, and must have the set user ID on execution bit on (see chmod(1)), so that the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the interface program for the duration of that command's execution. In this manner, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the "user list" for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of delta and, possibly, rmdel and cdc. The project-dependent interface program, as its name implies, must be custom-built for each project.

For more information on this interface program, see Appendix A "Function and Use of an SCCS Interface Program".

6.2. Format

SCCS files are composed of lines of ASCII text. Previous versions of SCCS used non-ASCII files. Therefore, files created by earlier versions of SCCS are incompatible with this version of SCCS. The SCCS files are arranged in six parts, as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (not including this checksum itself).
Delta Table	Information about each delta, such as its type, its SID, date and time of creation,

and commentary.

User Names — List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.

Flags — Indicators that control certain actions of various SCCS commands.

Descriptive Text — Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.

Body — Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in sccsfile(5); the checksum is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various ZEUS commands, such as vi(1), ed(1), grep(1), and cat(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when it is desired to simply "look" at the file.

NOTE

Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

6.3. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file, or portions of it (i.e., one or more "blocks") can be destroyed. SCCS commands (like most ZEUS commands) issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed (possibly by having lost one or more blocks, or by having been modified with, for example, ed(1)). No SCCS command will process a corrupted SCCS file except the admin command with the **-h** or **-z** keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the admin command with the -h keyletter on all SCCS files:

```
admin  -h  s.file1  s.file2  ...
        or
admin  -h  directory1  directory2  ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
ERROR [s.filename]:  corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the ls(1) command on that directory, and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the system administrator to request a restoral of the file from a backup copy. In the case of minor damage, repair through use of the editor ed(1) may be possible. In the latter case, after such repair, the following command must be executed:

```
admin  -z  s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

APPENDIX A
FUNCTION AND USE OF AN SCCS INTERFACE PROGRAM

ABSTRACT

This appendix discusses the use of a Source Code Control System Interface Program to allow more than one user to use SCCS commands upon the same set of files.

A.1. Introduction

In order to permit ZEUS users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This memorandum discusses the creation and use of such an interface program.

A.2. Function

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. However, there are situations (for example, in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the admin command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the get, delta, and, if desired, rmdel, cdc, and unget commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are generally reserved for use only by the administrator.

The interface program must be owned by the SCCS administrator, must be executable by non-owners, and must have the set user ID on execution bit on (see chmod(1)), so that, when executed, the "effective" user ID is the user ID of the administrator. This program's function is to invoke the

desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the "user list" (this is the list of login names of users who are allowed to modify an SCCS file by adding or removing deltas; the login names are specified using the admin(1) command) for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of delta and, possibly, rmdel and cdc.

A.3. A Basic Program

When a ZEUS program is executed it is passed (as argument Ø) the name by which it is invoked, followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), it may alter its processing depending upon which link is used to invoke it. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke (see exec(2)).

A generic interface program ("inter.c", written in C) is shown in Attachment I. Note the reference to the (unsupplied) function "filearg". This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

A.4. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program "inter.c" resides in directory "/z/xyz/sccs". Thus, the command sequence:

```
cd  /z/xyz/sccs
cc  ...  inter.c  -o  inter  ... -lpw
```

compiles "inter.c" to produce the executable module "inter" (the ellipses represent other arguments that may be required). The proper mode and the set user ID on execution bit are set by executing:

```
chmod 4755 inter
```

Finally, new links are created (names of the links may be arbitrary, provided the interface program is able to determine from them the names of SCCS commands to be invoked) by (for example):

```
ln inter get
ln inter delta
ln inter rmdel
```

Subsequently, any user whose C shell parameter, path (see csh(1)) or Bourne shell parameter PATH (see sh(1)) specifies directory "/z/xyz/sccs" as the one to be searched first for executable commands, may execute, for example:

```
get -e /z/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes "/usr/bin/get" (the actual SCCS get command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname "/z/xyz/sccs", so that the user would only have to specify:

```
get -e s.abc
```

to achieve the same results.

A.5. Conclusion

An SCCS interface program (Table A-1) is used to permit users having different user IDs to use SCCS commands upon the same files. Although this is its primary purpose, such a program may also be used as a pre-processor to SCCS commands since it can perform operations upon its arguments.

Table A-1. SCCS Interface Program "inter.c"

```
#define LENGTH 80
main(argc, argv)
int argc;
char *argv[];
{
        register int i;
        char cmdstr[LENGTH]
        char *filearg(), *sname();

        /*
        Process file arguments (those that don't begin with `-').
        */
        for (i = 1; i < argc; i++)
                if (argv[i][0] != '-')
                        argv[i] = filearg(argv[i]);

        /*
        Get `simple name' of name used to invoke this program
        (i.e., strip off directory-name prefix, if any).
        */
        argv[0] = sname(argv[0]);

        /*
        Invoke actual SCCS command, passing arguments.
        */
        sprintf(cmdstr, "/usr/bin/%s", argv[0]);
        execv(cmdstr, argv);
}

char
*filearg(s)
char *s;
{
        .
        .
        .
}
```

SED

A Noninteractive Text Editor*

* This information is based on an article originally written by Lee E. McMahon, Bell Laboratories.

Preface

This document is for users of sed, a noninteractive context editor that runs on the ZEUS Operating System. It is assumed that the user has some familiarity with string matching and substitution features of vi, the interactive screen-oriented editor of ZEUS.

Section 1 provides an introduction to sed. The format of sed editing commands appears in Section 2. Section 3 gives the available sed commands and use of arguments.

Examples appear throughout the text. Except where otherwise noted, the examples use the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Table of Contents

SECTION 1
INTRODUCTION

Sed is a noninteractive context editor designed for three cases:

1. Editing files too large for efficient interactive editing.

2. Editing any size file when the sequence of editing commands is too complicated to be efficiently typed in interactive mode.

3. Performing multiple "global" editing functions efficiently in one pass through the input.

Sed is a descendant of the editor, ed. Because of the differences between interactive and noninteractive operation, considerable changes have been made between ed and sed. Even experienced users of ed will be surprised if they use sed without reading Sections 3 and 4 of this document. The most striking resemblance between the two editors is in the class of patterns or regular expressions they recognize. The code for matching patterns is copied almost verbatim from the code for ed, and the description of regular expressions in Section 3 is copied almost verbatim from the writeup for ed in the ZEUS Reference Manual (Ø3-3255).

SECTION 2
COMMAND OPERATION

2.1. General Operation

Sed copies the standard input to the standard output, and can perform one or more editing commands on each line before writing it to the output. This action can be modified by flags on the command line (Section 2.2).

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses can be omitted. Any number of blanks or tabs can separate the addresses from the function. The function must be present. The arguments can be required or optional, according to which function is given. Tab characters and spaces at the beginning of lines are ignored.

2.2. Command Line Flags

Three flags are recognized on the command line:

-n: tells sed not to copy all lines, but only those specified by p functions or p flags after s functions (Section 4.4)

-e: tells sed to take the next argument as an editing command

-f: tells sed to take the next argument as a file name; the file should contain one editing command to a line

2.3. Flow of Edit Commands

For more efficient execution, all the editing commands are first compiled in the order they are encountered. This is generally the order in which they are attempted at execution time. During the execution phase, the commands are applied one at a time, and the input to each command is the output of all preceding commands.

The linear order of application of editing commands can be changed by the flow-of-control commands, t and b (Section 4.7). Even when the order of application is changed by

these commands, the input line to any command is the output of any previously applied command.

2.4. Pattern Space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the N command (Section 4.5).

SECTION 3
LINE SELECTION

3.1. Selecting Lines for Editing

Lines in an input file can be selected by addresses. Addresses can be either line numbers or context addresses.

The application of a group of commands can be controlled by one address or address-pair by grouping the commands with braces ({ })(Section 4.7).

3.2. Line Number Addresses

A line number is a decimal integer. As each line is read from the input, a line number counter is incremented. A line number address matches the input line, which causes the internal counter to equal the address line number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

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

3.3. Context Addresses

A context address is a pattern "regular expression" enclosed in slashes (/). The following regular expressions are recognized by sed:

1. An ordinary character (not one of the special characters discussed in this section) is a regular expression, and matches itself.

2. A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.

3. A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.

4. The characters \n match an embedded new line character, but not the new line at the end of the pattern space.

5. A period (.) matches any character except the terminal new line of the pattern space.

6. A regular expression followed by an asterisk (*) matches any number (including none) of adjacent occurrences of the regular expression it follows.

7. A string of characters in square brackets ([ ]) matches any character in the string, and no others. If the first character of the string is circumflex (^), the regular expression matches any character except the characters in the string and the terminal new line of the pattern space.

8. A concatenation of regular expressions is itself a regular expression. It matches the concatenation of strings that match the components of the regular expression.

9. A regular expression between the sequences \(and \) is identical to the regular expression, but has side-effects described in Section 4.3.

10. The expression \d means the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. Here d is a single digit. The string specified begins with the dth occurrence of \(, counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.

11. The null regular expression standing alone (for example, //) is equivalent to the last regular expression compiled.

To use one of the special characters:

```
^
$
.
*
[ ]
\
/
```

as a literal to match an occurrence of itself in the input, precede the special character with a backslash (\).

If a context address is to match the input, the whole pattern within the address must match some portion of the pattern space.

3.4. Number of Addresses

The commands in the next section can have zero, one, or two addresses. Two addresses are separated by a comma. Under each command, the maximum number of allowed addresses is given. It is an error for a command to have more addresses than the maximum allowed.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines that match that address.

If a command has two addresses, it is applied to the first line that matches the first address, and to all subsequent lines until and including the first subsequent line that matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated.

Examples:

```
/an/            matches lines 1, 3, 4 in the sample text
/an.*an/        matches line 1
/^an/           matches no lines
/./             matches all lines
/\./            matches line 5
/r*an/          matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/    matches line 1
```

SECTION 4
FUNCTIONS

4.1. General Information

All functions are named by a single character. In this section, the command format shows the maximum number of allowable addresses enclosed in parentheses, the single character function name, and possible arguments enclosed in angle brackets (< >). The angle brackets around the arguments are not part of the argument and must not be typed in actual editing commands. An expanded English translation of the single character name and a description of each function also appear.

4.2. Whole Line Functions

Within the text output by these functions, leading blanks and tabs disappear. To include leading blanks and tabs in the output, precede the first desired blank or tab with a backslash. The backslash does not appear in the output.

(2)d -- delete lines

The d function deletes from the file all those lines matched by its address(es).

It also has the effect that no further commands are attempted on the deleted lines. As soon as the d function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.

(2)n -- next line

The n function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the n command.

(1)a\
<text> -- append lines

The a function writes the argument <text> to the output after the line matched by its address. The a command is inherently multiline; a must appear at the end of a line, and <text> can contain any number of lines. The interior new lines must immediately follow a backslash character (\). The <text> argument is terminated by the first new line not immediately preceded by a backslash.

Once an a function is successfully executed, <text> is written to the output. The triggering line can be deleted entirely, but <text> is still written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line number counter.

```
(1)i\
<text> -- insert lines
```

The i function behaves like the a function, except that <text> is written to the output before the matched line. All other comments about the a function apply to the i function.

```
(2)c\
<text> -- change lines
```

The c function deletes the lines selected by its address(es) and replaces them with the lines in <text>. Like a and i, c must be followed by a new line entered after a backslash. Interior new lines in <text> must follow backslashes.

The c command can have two addresses, and thereby select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output. As with a and i, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter.

After a line has been deleted by a c function, no further commands are attempted on it.

If text is appended after a line by a or r functions, and the line is subsequently changed, the text inserted by the c function is placed before the text of the a or r functions.

NOTE

Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in sed commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to the standard input produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect is produced by either of the two following command lists:

```
n               n
i\              c\
XXXX            XXXX
d
```

4.3. Substitute Functions

A substitute function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The s function replaces the part of a line selected by <pattern> with <replacement>. It is also read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (Section 3.3). The only difference between <pattern> and a context address is that the context

address must be delimited by slash (/) characters and <pattern> can be delimited by any character other than space or new line.

By default, only the first string matched by <pattern> is replaced.

The <replacement> argument begins immediately after the second delimiting character of <pattern> and must be followed immediately by another instance of the delimiting character. Thus, there are three instances of the delimiting character.

The <replacement> is not a pattern, and the characters that are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

\d (where d is a single digit) is replaced by the dth substring matched by parts of <pattern> enclosed in \(and \). If nested substrings occur in <pattern>, the dth string is determined by counting opening delimiters. As in patterns, special characters can be made literal by preceding them with a backslash (\).

The <flags> argument can contain the following flags:

g -- substitute <replacement> for all nonoverlapping instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters. Characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The p flag prints the line to the output if a substitution was actually made by the s function. If several s functions, each followed by a p flag, successfully substitute in the same input line, multiple copies of the line are written to the output--one for each successful substitution.

w <filename> -- write the line to a file if there was a successful replacement. The w flag causes lines that are actually substituted by the s function to be written to a file named by <filename>. If <filename> exists before sed is run, it is overwritten; if not, it is created.

A single space must separate w and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for p.

A maximum of ten different file names can be mentioned after w flags and w functions.

Examples:

Applied to the standard input, the following command,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file "changes":

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

In nocopy mode, the command:

```
/X/s/an/AN/p
```

produces

```
In XANadu did Kubhla Khan
```

and the command:

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

produces:

In XANadu did Kubhla KhAN

4.4. Input/Output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the p function is encountered, regardless of what subsequent editing commands do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands do to them.

One space must separate the w and <filename>. A maximum of ten different files can be mentioned in write functions and w flags after s functions.

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

The read function reads the contents of <filename> and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched its address. If r and a functions are executed on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed.

One space must separate the r and <filename>. If a file mentioned by an r function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Since there is a limit to the number of files that can be opened simultaneously, take care not to mention more than ten files in w functions or flags. The number is reduced to nine if any r functions are present.

Examples:

Assume that the file notel has the following contents:

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

The following command:

```
/Kubla/r notel
```

produces:

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

4.5. Patterns with New Line

Three functions, all entered as capital letters, deal specifically with pattern spaces containing embedded new lines. They provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space and the two input lines are separated by an embedded new line. Pattern matches can extend across the embedded new line(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first new line character in the current pattern space. If the pattern space becomes empty (the only new line is the terminal new line), read another line from the input. Begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first new line in the pattern space.

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

4.6. Hold and Get Functions

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

(2)h -- hold pattern space

The h function copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.

(2)H -- Hold pattern space

The H function appends the contents of the pattern space to the contents of the hold area. The former and new contents are separated by a new line.

(2)g -- get contents of hold area

The g function copies the contents of the hold area into the pattern space, destroying the previous contents of the pattern space.

(2)G -- Get contents of hold area

The G function appends the contents of the hold area to the contents of the pattern space. The former and new contents are separated by a new line.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example:

The commands

```
lh
ls/ did.*//
lx
G
s/\n/  :/
```

applied to the standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

4.7. Flow-of-Control Functions

These functions control the application of functions to the lines selected by the address portion. They do no editing on the input lines.

(2)! -- Don't

The Don't command causes the next command written on the same line to be applied to input lines not selected by the address part.

(2){ -- Grouping

The grouping command, a left brace ({), causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping command can appear on the same line as the {, or on the next line.

The group of commands is terminated by a right brace (}) standing on a line by itself.

Groups can be nested.

(Ø):<label> -- place a label

The label function marks a place in the list of editing commands that can be referred to by b and t functions. The <label> can be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile-time diagnostic is generated, and no execution is attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after a colon function with the same <label> is encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile-time diagnostic is produced, and no execution is attempted.

A b function with no <label> is a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The t function tests whether any successful substitutions have been made on the current input line. If so, it branches to <label>; if not, it does nothing. The flag indicating that a successful substitution has been executed is reset by:

1. reading a new input line, or
2. executing a t function.

4.8. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The q function writes the current line to the output, writes any appended or read text, and terminates execution.

An Introduction to the ZEUS Shell*

* This information is based on an article originally written by S.R. Bourne, Bell Laboratories.

10/14/83

Preface

The shell is both a command language and a programming language that provides an interface to the ZEUS operating system. This document describes, with examples, the ZEUS shell.

This version of the shell is also referred to as the "Bourne Shell" after its original author, S. R. Bourne of Bell Laboratories. There are at least four other shell programs in moderately widespread use. Users can select the shell they feel most comfortable with. Most of Zilog's internal users use the C Shell, which has additional features for interactive work.

The first section covers most requirements of terminal users. Some familiarity with ZEUS is an advantage when reading this section. ZEUS for Beginners in this manual provides the basis for this familiarity.

Section 2 describes features of the shell primarily intended for use within shell procedures. These include control-flow primitives and string-valued variables. Knowledge of a programming language is helpful when reading this section.

The last section describes more advanced features of the shell. References of the form "pipe (2)" refer to a section of the ZEUS Reference Manual.

List of Illustrations

```
cc pgm.c &
```

calls the C compiler to compile the file pgm.c. The trailing & is an operator that instructs the shell to run the command in the background. To help keep track of such a process, the shell reports its process number following its creation. A list of currently active processes can be obtained using the ps command.

1.4. Input/Output Redirection

Most commands produce output on the standard output device (the terminal). This output can also be sent to a file by writing, for example,

```
ls -l >file
```

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

```
ls -l >>file
```

The standard input of a command can be taken from a file instead of the terminal by entering

```
wc <file
```

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

```
wc -l <file
```

is used.

1.5. Pipelines and Filters

The standard output of one command can be connected to the standard input of another by entering the pipe operator, (|), as in,

```
ls -l | wc
```

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

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

except that no file is used. Instead, the two processes are connected by a pipe (2) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting wc when there is nothing to read and halting ls when the pipe is full.

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

```
ls | grep old
```

prints those lines of the output from ls that contain the string old. Another useful filter is sort. For example,

```
who | sort
```

prints an alphabetically sorted list of logged-in users.

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

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

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

1.6. File Name Generation

Many commands accept arguments that are file names. For example,

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

prints information relating to the file main.c.

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

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

generates, as arguments to ls, all file names in the current directory that end in .c. The character * is a pattern that matches any string including the null string. 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 minus matches any character lexically between the pair.

`[!...]` Matches all but the specified character(s).

For example,

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

matches all names in the current directory beginning with one of the letters a through z. For example,

```
ls *[!o]
```

lists all files not ending in 'o'.

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern, the pattern is passed unchanged as an argument.

This mechanism saves typing, selects names according to some pattern, and finds files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all core files in subdirectories of **/usr/fred.** (Echo is a standard ZEUS command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all subdirectories of **/usr/fred.**

There is one exception to the general rules given for patterns. A single period (.) at the start of a file name must be explicitly matched. For example,

```
echo *
```

echoes all file names in the current directory not beginning with ".".

```
echo .*
```

echoes all those file names that begin with ".". This avoids matching the name "." (the current directory) with ".." (the parent directory). The ls command suppresses information for the "." and ".." files unless a user is logged in as "zeus".

1.7. Quoting

Characters that have a special meaning to the shell, such as <, >, *, ?, |, &, and , are called metacharacters. (A complete list of metacharacters is given in Appendix B.) Any character preceded by a backslash (\) is quoted and loses its special meaning. The \ itself is not echoed, so

```
echo \?
```

echoes a single **?** and

```
echo \\
```

echoes a single ****. To allow long strings to be continued over more than one line, the sequence **\new line** is ignored.

The **** is convenient for quoting single characters, but clumsy and error prone when more than one character needs quoting. A string of characters can be quoted by enclosing the string between single quotes. For example,

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

echoes

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

The quoted string can contain new lines, which are preserved; it cannot contain a single quote. This quoting mechanism is the simplest and is recommended.

A third quoting mechanism uses double quotes (Section 3.5).

1.8. Prompting

When the shell is used from a terminal, it issues a prompt before reading a command. By default, this prompt is a dollar sign (**$**); it can be changed by the PS1 command. For example,

```
PS1=yesdear
```

sets the prompt to be the string yesdear.

If a new line is typed and further input is needed, the shell issues the > prompt. Sometimes this can be caused by mistyping a quote mark. If it is unexpected, an interrupt (DEL) returns the shell to read another command. This prompt can be changed by the PS2 command. For example,

```
PS2=more
```

1.9. The Shell and Login

Following login (1), the shell is called to read and execute commands typed at the terminal. If an **/etc/profile** file exists and if the user's login directory contains the file **.profile**, those files are assumed to contain commands and are read by the shell before any commands are read from the terminal.

In addition, the shell will execute the commands contained in both /etc/profile and .profile whenever invoked with an argument zero (that is, $Ø) containing a "-" as the first character. An example is

```
su - name
```

When invoked, the shell searches the environment for the variable SHELL. If SHELL is found, and has an "r" in the simple part of its value (the part after the last "/"), then the shell becomes restricted. Restricted logins, such as games, should include the following in their .profile files:

```
SHELL=/usr/rsh
export SHELL
```

1.10. Summary

ls
Print the names of files in the current directory.

ls >file
Put the output from ls into file.

ls | wc -l
Print the number of files in the current directory.

ls | grep old

Print those file names containing the string old.

ls | grep old | wc -l
Print the number of files whose names contain the string old.

cc pgm.c &
Run cc in the background.

Preface

The shell is both a command language and a programming language that provides an interface to the ZEUS operating system. This document describes, with examples, the ZEUS shell.

This version of the shell is also referred to as the "Bourne Shell" after its original author, S. R. Bourne of Bell Laboratories. There are at least four other shell programs in moderately widespread use. Users can select the shell they feel most comfortable with. Most of Zilog's internal users use the C Shell, which has additional features for interactive work.

The first section covers most requirements of terminal users. Some familiarity with ZEUS is an advantage when reading this section. ZEUS for Beginners in this manual provides the basis for this familiarity.

Section 2 describes features of the shell primarily intended for use within shell procedures. These include control-flow primitives and string-valued variables. Knowledge of a programming language is helpful when reading this section.

The last section describes more advanced features of the shell. References of the form "pipe (2)" refer to a section of the ZEUS Reference Manual.

Table of Contents

List of Illustrations

SECTION 1
BASIC TASKS

1.1. Introduction

The shell is a command programming language that provides an interface to the ZEUS operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. 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 file names or flags, can be passed to a command. A return code that is set by commands can be used to determine control flow, and the standard output from a command can be used as shell input.

The shell modifies the environment in which commands run. Input and output can be redirected to files, and processes that communicate through pipes can be invoked. Commands are found by searching directories in the file. Commands can be read either from the terminal or from a file.

1.2. Simple Commands

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

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

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

1.3. Background Commands

To execute a command, the shell normally creates a new process and waits for it to finish. A command can also be run in the background, that is, without waiting for the process to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file pgm.c. The trailing & is an operator that instructs the shell to run the command in the background. To help keep track of such a process, the shell reports its process number following its creation. A list of currently active processes can be obtained using the ps command.

1.4. Input/Output Redirection

Most commands produce output on the standard output device (the terminal). This output can also be sent to a file by writing, for example,

```
ls -l >file
```

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

```
ls -l >>file
```

The standard input of a command can be taken from a file instead of the terminal by entering

```
wc <file
```

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

```
wc -l <file
```

is used.

1.5. Pipelines and Filters

The standard output of one command can be connected to the standard input of another by entering the pipe operator, (|), as in,

```
ls -l | wc
```

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

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

except that no file is used. Instead, the two processes are connected by a pipe (2) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting wc when there is nothing to read and halting ls when the pipe is full.

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

```
ls | grep old
```

prints those lines of the output from ls that contain the string old. Another useful filter is sort. For example,

```
who | sort
```

prints an alphabetically sorted list of logged-in users.

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

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

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

1.6. File Name Generation

Many commands accept arguments that are file names. For example,

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

prints information relating to the file main.c.

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

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

generates, as arguments to ls, all file names in the current directory that end in .c. The character * is a pattern that matches any string including the null string. 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 minus matches any character lexically between the pair.

`[!...]` Matches all but the specified character(s).

For example,

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

matches all names in the current directory beginning with one of the letters a through z. For example,

```
ls *[!o]
```

lists all files not ending in 'o'.

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern, the pattern is passed unchanged as an argument.

This mechanism saves typing, selects names according to some pattern, and finds files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all core files in subdirectories of **/usr/fred.** (Echo is a standard ZEUS command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all subdirectories of **/usr/fred.**

There is one exception to the general rules given for patterns. A single period (.) at the start of a file name must be explicitly matched. For example,

```
echo *
```

echoes all file names in the current directory not beginning with ".".

```
echo .*
```

echoes all those file names that begin with ".". This avoids matching the name "." (the current directory) with ".." (the parent directory). The ls command suppresses information for the "." and ".." files unless a user is logged in as "zeus".

1.7. Quoting

Characters that have a special meaning to the shell, such as <, >, *, ?, |, &, and , are called metacharacters. (A complete list of metacharacters is given in Appendix B.) Any character preceded by a backslash (\) is quoted and loses its special meaning. The \ itself is not echoed, so

```
echo \?
```

echoes a single **?** and

```
echo \\
```

echoes a single \. To allow long strings to be continued over more than one line, the sequence **\new line** is ignored.

The \ is convenient for quoting single characters, but clumsy and error prone when more than one character needs quoting. A string of characters can be quoted by enclosing the string between single quotes. For example,

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

echoes

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

The quoted string can contain new lines, which are preserved; it cannot contain a single quote. This quoting mechanism is the simplest and is recommended.

A third quoting mechanism uses double quotes (Section 3.5).

1.8. Prompting

When the shell is used from a terminal, it issues a prompt before reading a command. By default, this prompt is a dollar sign ($); it can be changed by the PS1 command. For example,

```
PS1=yesdear
```

sets the prompt to be the string yesdear.

If a new line is typed and further input is needed, the shell issues the > prompt. Sometimes this can be caused by mistyping a quote mark. If it is unexpected, an interrupt (DEL) returns the shell to read another command. This prompt can be changed by the PS2 command. For example,

```
PS2=more
```

1.9. The Shell and Login

Following login (1), the shell is called to read and execute commands typed at the terminal. If an **/etc/profile** file exists and if the user's login directory contains the file **.profile**, those files are assumed to contain commands and are read by the shell before any commands are read from the terminal.

In addition, the shell will execute the commands contained in both /etc/profile and .profile whenever invoked with an argument zero (that is, $0) containing a "-" as the first character. An example is

```
su - name
```

When invoked, the shell searches the environment for the variable SHELL. If SHELL is found, and has an "r" in the simple part of its value (the part after the last "/"), then the shell becomes restricted. Restricted logins, such as games, should include the following in their .profile files:

```
SHELL=/usr/rsh
export SHELL
```

1.10. Summary

ls
Print the names of files in the current directory.

ls >file
Put the output from ls into file.

ls | wc -l
Print the number of files in the current directory.

ls | grep old

Print those file names containing the string old.

ls | grep old | wc -l
Print the number of files whose names contain the string old.

cc pgm.c &
Run cc in the background.

SECTION 2
SHELL PROCEDURES

2.1. Introduction

The shell reads and executes commands contained in a file. For example,

```
sh file [ args... ]
```

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

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

ZEUS files have three independent attributes: read, write, and execute. The ZEUS command chmod (1) can be used to make a file executable. For example,

```
chmod +x wg
```

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

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case, a new process is created to run the command.

In addition to providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special shell parameter, **$***, substitutes for all positional parameters except **$0**. This provides some default arguments, as in

```
nroff -T450 -ms $*
```

which prepends some arguments to those already given.

2.2. 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**, which contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of tel is

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

The command

```
tel fred
```

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

```
tel fred bert
```

prints those lines containing fred followed by those for bert.

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

```
for name in w1 w2 ...
do command-list
done
```

A command-list is a sequence of one or more simple commands separated or terminated by a new line or semicolon. Reserved words like do and done are only recognized following a new line or semicolon. Name is a shell variable that is set to the words w1 w2... in turn each time the command-list following do is executed. If in w1 w2... is omitted, the loop is executed once for each positional parameter, that is, "in $*" is assumed.

Another example of the loop is the create command, for which the text is

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

The command

```
create alpha beta
```

ensures that two files, alpha and beta, exist and are empty. The notation >file can be used on its own to create or clear the contents of a file. A semicolon (or a new line) is required before done.

2.3. Control Flow--Case

The **case** notation provides a multiple branch. For example,

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

is an append command. When called with one argument as

```
append file
```

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

```
append file1 file2
```

appends the contents of file1 to file2. If more than two arguments are supplied to append, a message is printed indicating improper usage.

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

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

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

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 next example, the commands following the second * are never executed.

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

Another example of the **case** construction is distinguishing 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/cØ $i ... ;;
     *) echo 'unexpected argument $i' ;;
     esac
done
```

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

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

esac
```

is equivalent to

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

esac
```

The usual quoting conventions apply so that

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

matches the ? character.

2.4. Here Documents

The shell procedure tel in Section 2.2 uses the file **/usr/lib/telnos** to supply the data for grep. An alternative includes this data within the shell procedure as a here document, as in,

```
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. The string **!** is arbitrary; the document is terminated by a line that consists of the string following <<, whatever that is.

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
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

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

and changes all occurrences of string1 in file to string2. Substitution is prevented if \ is used to quote the special character **$**, as in

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

This version of edg is equivalent to the first except that ed prints a ? if there are no occurrences of the string **$1**. Substitution within a here document is prevented entirely by quoting the terminating string. For example,

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

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

2.5. Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables can be given values with commands such as

```
user=fred box=mØØØ acct=mhØØØØ
```

which assigns values to the variables **user**, **box**, and **acct**. A variable can be set to the null string by entering, for example,

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

echoes fred.

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

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

moves the file pgm from the current directory to the directory **/usr/fred/bin.** A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

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

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

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

```
ps -a >$tmpa
```

substitutes the value of the variable tmpa.

Except for $?, the following are set initially by the shell. $? is set after executing each command.

$? 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 return codes is dealt with under 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 flags, such as -x and -v.

The following variables have a special meaning to the shell and must 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 it was last looked at, the shell prints the message you have mail before prompting for the next command. This variable is typically set in the file **.profile**, in the user's login directory. For example,

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

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

```
cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin.**

```
cat wn
```

prints on the terminal the file wn in this directory. The command cd with no argument is equivalent to

```
cd $HOME
```

This variable is generally set in the user's login profile.

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

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

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

$PS1 The primary shell prompt string, by default, $.

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

$IFS The set of characters used by blank interpretation (Section 3.5).

2.6. Test Command

The test command is generally used by shell programs. For example,

```
test -f file
```

returns zero exit status if file exists and nonzero exit status otherwise. In general, test evaluates a predicate and returns the result as its exit status. Some of the more frequently used test arguments are given here. (Test (1) contains a complete specification.)

test s	true if the argument s is not the null string
test -f file	true if file exists
test -r file	true if file is readable
test -w file	true if file is writable
test -d file	true if file is a directory

2.7. 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; their actions are determined by the exit status returned by commands. A while loop has the general form

```
while command-list1
do command-list2
done
```

Loops such as:

```
while echo hi
do

    continue
done
```

echo an infinite number of hi's.

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

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

is equivalent to

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

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

Another use 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
```

loops until file exists. Each time around the loop, it waits for five minutes before trying again.

2.8. Control Flow--If

Also available is a general conditional branch of the form,

```
if command-list
then    command-list
else    command-list
fi
```

which tests the value returned by the last simple command following if.

The if command can be used in conjunction with the test command to test for the existence of a file as in

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

A multiple test if command of the form

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

can be written using an extension of the if notation as,

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

The following example is the touch command, which changes the "last modified" time for a list of files. The command can be used in conjunction with make (1) to force recompilation of a list of files.

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

The -c flag in this command forces subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable flag is set to some non-null string if the -c argument is encountered. The commands

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

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

The sequence

```
if command1
then    command2
```

can be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

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

2.9. Command Grouping

Commands can be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

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

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

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

The commands

```
cd x; rm junk
```

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

2.10. Debugging Shell Procedures

The shell provides two tracing mechanisms to help debug shell procedures. The first is invoked within the procedure, as with

```
set -v  (v for verbose)
```

and causes lines of the procedure to be printed as they are read. This is useful to help isolate syntax errors. It can be invoked without modifying the procedure by using

```
sh -v proc ...
```

where proc is the name of the shell procedure. This flag can be used in conjunction with the -n flag, which prevents execution of subsequent commands. (Using set -n at a terminal renders the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

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

```
set -
```

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

The command

```
set --
```

indicates that the flags are to remain unchanged and is useful in setting $1 to a string beginning with a minus (-).

2.11. The man Command

The man command can be used used to print sections of a document. It is called, for example, as

```
man sh
man -t ed
man 2 fork
```

In the first line, a section of the sh manual is printed. Since no section is specified, Section 1 is used. The second example typesets (-t option) a section of the manual ed. The last prints the fork manual page from Section 2.

A more elaborate example of the man command appears in Figure 2-1.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in
     [1-9*]     s=$i ;;

     -t)        N=t ;;

     -n)        N=n ;;

     -*)        echo unknown flag \'$i\' ;;

      *)        if test -f man$s/$i.$s

               then        ${N}roff manØ/${N}aa man$s/$i.$s
              else : 'look through all manual sections'
                  found=no
                  for j in 1 2 3 4 5 6 7 8 9
                  do if test -f man$j/$i.$j
                       then man $j $i
                            found=yes
                       fi
                  done
                  case $found in
                       no) echo '$i: manual page not found'
            esac
            fi
     esac
done
```

Figure 2-1 A Version of the man Command

SECTION 3
KEYWORD PARAMETERS

3.1. Introduction

Shell variables are given values by assignment or by invoking a shell procedure. 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
```

executes command with user set to fred. The **-k** flag for the set command causes arguments of the form name=value to be interpreted in this way anywhere in the argument list. Such names are called keyword parameters. If any arguments remain, they are available as positional parameters **$1**, **$2**, and so on.

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

```
set - *
```

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

3.2. Parameter Transmission

When a shell procedure is invoked, both positional and keyword parameters can be supplied with the call. Keyword parameters are implicitly available to a shell procedure by specifying in advance that such parameters are to be exported. For example, the command

```
export user box
```

marks the variables **user** and **box** for export. When a shell procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. A shell procedure cannot modify the state of its caller without explicit request on

the part of the caller. Shared file descriptors are an exception to this rule.

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

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.3. Parameter Substitution

If a shell parameter is not set, the null string is substituted for it. For example, if the variable **d** is not set

```
echo $d
```

or

```
echo ${d}
```

echoes nothing. A default string can be given as in

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

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

```
echo ${d-'*'}
```

echoes * if the variable **d** is not set. Similarly,

```
echo ${d-$1}
```

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

```
echo ${d=.}
```

which substitutes the same string as

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

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

If there is no default, the notation

```
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 is abandoned. If message is absent, a standard message is printed. An example of a shell procedure that requires some parameters to be set starts as follows:

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

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

3.4. Command Substitution

The standard output from a command can be substituted in a manner similar to parameter substitution. 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 command

```
d= `pwd`
```

is equivalent to

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

The entire string between grave accents 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 ` must be escaped using a \. For example,

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

is equivalent to

```
ls $1
```

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

For example,

```
basename main.c
```

prints the string main. Its use is illustrated by the following fragment from a cc command.

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

Here, **B** is set to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples:

- **for i in `ls -t`; do ...**
  The variable **i** is set to the names of files in time order, most recent first.

- **set `date`; echo $6 $2 $3, $4**
  prints, for example, 1981 Nov 1, 23:59:59

3.5. Evaluation and Quotation

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

Commands are parsed initially according to the grammar given in Appendix A. Before a command is executed, the following substitutions occur:

- parameter substitution; for example, **$user**

- command substitution; for example, **`pwd`**

  Only one evaluation occurs, so that if the value of the variable X is the string $y, then

  ```
  echo $X
  ```

  echoes $y.

- blank interpretation

Following the above substitutions, the resulting characters are broken into nonblank words (blank interpretation). For this purpose "blanks" are the characters of the string **$IFS**. By default, this string consists of blank, tab, and new line. The null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

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

```
echo $null
```

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

- file name generation

  Each word is then scanned for the file pattern characters comma, question mark, and [...], and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

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

In addition to the quoting mechanisms described previously, a third quoting mechanism is provided that uses double quotes. Within double quotes, parameter and command substitution occurs, but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and are quoted using \.

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

For example,

```
echo "$x"
```

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

```
echo "$*"
```

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

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

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

The command

```
echo "$@"
```

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

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

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

	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

Where more than one evaluation of a string is required, the built-in command eval is used. If the variable **X** has the value $y, and if **y** has the value pqr, then

```
eval echo $X
```

echoes the string pqr.

The eval command evaluates its arguments and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

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

3.6. Error Handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by gtty (2)). A shell invoked with the -i flag is also interactive.

Execution of a command (Section 3.8) can fail for any of the following reasons:

- Input/output redirection fails, 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" (see Figure 3-1 for a complete list of ZEUS signals)
- The command terminates normally but returns a nonzero exit status

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

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

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

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

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

Figure 3-1 ZEUS Signals

3.7. Fault Handling

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

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

sets a trap for signal 2 (terminal interrupt), and if this signal is received, executes the commands

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

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.

ZEUS signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. They can be left to cause termination of the process without any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (Section 3.8), trap commands and the signal are ignored.

The use of trap is illustrated by the modified version of the touch command in Figure 3-2. 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 terminate without removing the file.

Since there is no signal Ø in ZEUS, it is used by the shell to indicate the commands executed on exit from the shell procedure.

A procedure can itself ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the nohup command:

```
trap " 1 2 3 15
```

which causes hangup, interrupt, quit, and kill to be ignored both by the procedure and by invoked commands.

```
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
     esac
done
```

Figure 3-2 The touch Command

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

Figure 3-3 The scan Command

Traps can be reset by entering

```
trap 2 3
```

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

```
trap
```

The procedure scan (Figure 3-3) is an example of the use of

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

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

3.8. Command Execution

To run other than a built-in 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, and is established in the child process before the command is executed. The built-in command exec, used in the rare cases when no fork is required, replaces the shell with a new command. For example, a simple version of the nohup command looks like

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

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

In the following, word is subject only to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

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

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

> 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, output is appended by seeking to the end; otherwise, the file is created.

< word — The standard input (file descriptor Ø) 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, no interpretation of the document occurs. If word is not quoted, parameter and command substitution occur and \ is used to quote the characters \, $, and `, and the first character of word. In the latter case, **\new line** is ignored.

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

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

<&- — The standard input is closed.

>&- — The standard output is closed.

Any of the above can be preceded by a digit to create the file descriptor specified by the digit instead of the default Ø or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to file. Also,

```
... 2>&1
```

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

The environment for a command run in the background such as

```
list *.c | lpr &
```

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

```
ed file &
```

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

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

3.9. Invoking the Shell

The following flags 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**.

-c string
: If the -c flag is present, commands are read from string.

-s
: If the -s flag is present or if no arguments remain, commands are read from the standard input. Shell output is written to standard error (file descriptor 2).

-i
: If the -i flag is present or if the shell input and output are attached to a terminal (as told by gtty), this shell is interactive. In this case, TERMINATE is ignored so that **kill Ø** does not kill an interactive shell, and INTERRUPT is caught and ignored so that **wait** is interruptable. In all cases, QUIT is ignored by the shell.

APPENDIX A
GRAMMAR

item:
word
input-output
name = value

simple-command:
item
simple-command item

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

pipeline:
command
pipeline | command

andor:
pipeline
andor && pipeline
andor || pipeline

command-list:
andor
command-list ;
command-list &
command-list ; andor
command-list & andor

input-output:
> file
< file
>> word
<< word

file:
word
& digit
& -

case-part:
pattern) command-list ;;

pattern:
word
pattern | word

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

empty:

word: a sequence of nonblank characters

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

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

APPENDIX B
BOURNE SHELL MESSAGES

B.1. The Full List

- arg list too long
- argument expected
- bad directory
- bad file number
- bad number
- bad option(s)
- bad substitution
- bad trap
- cannot create
- cannot execute
- cannot fork: too many processes
- cannot fork: no swap space
- cannot make pipe
- cannot open
- cannot shift
- core dumped
- illegal io
- invalid shell script
- is not an identifier
- is read only
- Node may not be removed
- no more processes, waiting for current ones to complete
- no space
- not a login shell
- not found
- not in environment
- parameter not set
- parameter null or not set
- restricted
- syntax error
- text busy
- too big
- you have mail

APPENDIX C
METACHARACTERS AND RESERVED WORDS

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

Patterns

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

Substitution

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

Quoting

****	quote the next character
'...'	quote the enclosed characters except for '

''...'' quote the enclosed characters except for $, `, \, or "

Reserved Words

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

TBL - A PROGRAM TO FORMAT TABLES*

*This information is based on an article written by M. E. Lesk

10/14/83

Table of Contents

SECTION 1
INTRODUCTION

Tbl is a document formatting preprocessor for troff or nroff which makes even fairly complex tables easy to specify and enter. Tables are made up of columns which can be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings can be placed over single columns or groups of columns. Horizontal or vertical lines can be drawn as desired in the table, and any table or element can be enclosed in a box.

The input to tbl is text for a document, with tables preceded by a .TS (table start) command and followed by a .TE (table end) command. Tbl processes the tables, generating nroff formatting commands, and leaves the remainder of the text unchanged. The .TS and .TE lines are copied, too, so that nroff page layout macros (such as the memo formatting macros) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the .TS or .TE lines are copied but otherwise ignored, and can be used by document layout macro commands.

The format of the input is as follows:

```
.TS
options;
format.
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, can be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

1.1. Options

There can be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

center
     center the table (default is left-adjust);

expand
     make the table as wide as the current line length;

box
     enclose the table in a box;

allbox
     enclose each item in the table in a box;

doublebox
     enclose the table in two boxes;

tab(x)
     use x instead of tab to separate data items.

linesize(n)
     set lines or rules (e.g. from box) in n point type;

delim(xy)
     Recognize x and y as the eqn delimiters.

1.2. Format

The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any, and each line contains a key-letter for each column of the table.)

It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

l or L to indicate a left-adjusted column entry;

r or R to indicate a right-adjusted column entry;

c or C to indicate a centered column entry;

n or N to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;

a or A to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column;

s or S to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or

^ to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

When numerical alignment 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 units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& can be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output.

NOTE

If numerical data are used in the same column with wider L or r type table entries, the widest number is centered relative to the wider L or r items (L is used instead of l for readability; they have the same meaning as key-letters).

Alignment within the numerical items is preserved. This is similar to the behavior of a type data, as explained above. However, alphabetic subcolumns (requested by the a key-letter) are always slightly indented relative to L items; if necessary, the column width is increased to force this. This is not true for n type entries.

WARNING

The n and a items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. A simple format might appear as:

```
c  s  s
l  n  n
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data.

Horizontal lines

A key-letter can be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line.

If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines

A vertical bar can be placed between column key-letters. This will cause 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.

Space between columns

A number can follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in ens (one en is about the width of the letter 'n').

If the "expand" option is used, then 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 worst case (largest space requested) governs.

Vertical spanning

Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

Font changes

A key-letter can be followed by a string containing a font name or number preceded by the letter **f** or **F**.

This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes

A key-letter can be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number can be a signed digit, in which case 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 can be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number can 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 can be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a 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, its width is used.

The width is also used as a default line length for included text blocks. Normal nroff units can be used to scale the width value; if none are used, the default is 'ens'. If the width specification is a unitless integer the parentheses can be omitted. If the width value is changed in a column, the last one given controls.

Equal width columns

A key-letter can be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

NOTE

The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
npl2w(25i)fI  6
```

Alternative notation

Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats can be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

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 silently.

1.3. Data

The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option tabs. There are a few special cases:

Nroff commands within tables

An input line beginning with a '.' followed by anything but a number is assumed to be a command to nroff and is passed through unchanged, retaining its position in the table. So, for example, space within a table can be produced by ".sp" commands in the data.

Full width horizontal lines

An input line containing only the character _ (underscore) or = (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 character _ or = 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, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines

An input table entry containing only the string _ is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters

An input table entry containing only a string of the form **\R**x where x is any character is replaced by repetitions of the character x as wide as the data in the column. The sequence of x's is not extended to meet adjoining columns.

Vertically spanned items

An input table entry containing only the character string **\^** indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks

In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus the sequence

```
. . . T{
block of
text
T} . . .
```

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the **T}** end delimiter must begin a line; additional columns of data can follow after a tab on the same line.

If more than twenty or thirty text blocks are used in a table, various limits in the nroff program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by nroff, and replaced in the table as a solid block. If no line length is specified in the block of text itself, or in the table format, the default is to use L x C/(N+1) where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the block of text are those in effect at the beginning of the table (including the effect of the ".TS" macro) and any table format specifications of size, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course.

However, nroff commands within the table data but not within the text block do not affect that block.

WARNING

Although any number of lines can be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, can be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting can arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ".TS" command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data.

Therefore, although arbitrary nroff requests can be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

1.4. Additional Command Lines

If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ".T&" (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

Using this procedure, each table line can be close to its corresponding format line.

WARNING

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 equal width.

1.5. Usage

On ZEUS, tbl can be run on a simple table with the command

```
tbl input-file | nroff
```

but for more complicated use, where there are several input files, and they contain equations and ms memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff -ms
```

and, of course, the usual options can be used on the nroff and eqn commands.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special -TX command line option to tbl which produces output that does not have fractional line motions in it. The only other command line options recognized by tbl are -ms and -mm which are turned into commands to fetch the corresponding macro files; ususlly it is more convenient to place these arguments on the troff part of the command line, but they are accepted by tbl as well.

Note that when eqn and tbl are used together on the same file tbl should be used first. If there are no equations within tables, either order works, but it is usually faster to run tbl first, since eqn normally produces a larger expansion of the input than tbl. However, if there are equations within tables (using the delim mechanism in eqn), tbl must be first or the output will be scrambled. Users must also beware of using equations in n-style columns; this is nearly always wrong, since tbl attempts to split numerical format items into two parts and this is not possible with equations. Use the delim(xx) table option to prevent splitting of numerical columns within the delimiters.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns can fail because of limits in nroff, producing the 'too many number registers' message. Nroff number registers used by tbl must be avoided by the

user within tables; these include two-digit names from 31 to 99, and names of the forms #x, x+, x|, ^x, and x-, where x is any lower case letter. The names ##, and #- and #^ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register, so they cannot be used in the same column.

For aid in writing layout macros, tbl defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and can be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the ms macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro.

If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is not a feature of tbl, but of the ms layout macros.

List of Tbl Command Characters and Words

Command	Meaning
a A	Alphabetic subcolumn
allbox	Draw box around all items
b B	Boldface item
box	Draw box around table
c C	Centered column
center	Center table in page
doublebox	Doubled box around table
e E	Equal width columns
expand	Make table full line width
f F	Font change
i I	Italic item
l L	Left justified column
n N	Numerical column
nnn	Column separation
p P	Point size change
r R	Right adjusted column
s S	Spanned item
t T	Vertical spanning at top
tab (x)	Change data separator character
T{ T}	Text block
v V	Vertical spacing change
w W	Minimum width value
.xx	Included troff command
|	Vertical line
||	Double vertical line
^	Vertical span
\^	Vertical span
=	Double horizontal line
-	Horizontal line
_	Short horizontal line
\Rx	Repeat character

A TROFF TUTORIAL*

* This information is based on an article originally written by Brian W. Kernighan, Bell Laboratories.

Preface

This document is an introduction to the most basic use of troff. It presents enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of troff commands.

This document also serves as a tutorial on nroff, the ZEUS formatter for text printed on a character or line printer. Nroff commands are a subset of troff commands, and nroff ignores commands specific to troff. Only those parts of Sections 2, 3, and 6 that cover functions specific to phototypesetting are irrelevant to nroff.

Table of Contents

SECTION 1
GENERAL INFORMATION

Nroff and troff are text-formatting programs for producing high quality printed output. Nroff produces output for devices such as line printers and hardcopy terminals, and troff produces output for a Graphic Systems C/A/T phototypesetter with other typesetter interfaces being developed. Presently, the C/A/T is not supported under ZEUS, although troff output files can be transferred to other UNIX systems with C/A/T support.

The phototypesetter itself normally runs with four fonts, containing Roman, Italic, and Bold letters, a full Greek alphabet, and a number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and can be placed anywhere on the page.

Troff allows full control over fonts, sizes, and character positions, as well as the usual features of a formatter, such as right-margin justification, automatic hyphenation, page titling, and numbering. It also provides macros, arithmetic variables and operations, and conditional testing for complicated formatting tasks.

The most important rule is to use troff through some intermediary. In many ways, troff resembles a remarkably powerful and flexible assembly language that requires many operations to be specified at a level of detail and in a form that is hard for most people to use effectively.

For two special applications there are programs that provide an interface to troff. The eqn program provides an easy-to-learn language for typesetting mathematics; the eqn user does not need to know troff to typeset mathematics. The tbl program provides the same convenience for producing complex tables.

For producing text that contains mathematics or tables, there are a number of macro packages that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with troff. In particular, the -ms and PWB/MM packages provide most facilities needed for a wide range of document preparation. There are also packages for viewgraphs, for simulating the older roff formatters on ZEUS, and for other special applications. These packages are easier to use than direct use of troff and should be considered first. In the cases where existing

packages do not do what is needed, small changes can be made to adapt packages that already exist.

To use troff, prepare the actual text to be printed and some information that tells how it is to be printed. Most commands to troff start with a period and appear one command per line, separate from the text itself. For example,

```
Some text.
.ps 14
Some more text.
```

Many troff commands use the backslash character (\) to introduce the commands and separate special characters within a line of text.

SECTION 2
POINT SIZES AND LINE SPACING

The command .ps sets the point size. One point is 1/72 inch, so 6-point characters are 1/12 inch high, and 36-point characters are 1/2 inch. There are 15 available point sizes, shown in Figure 2-1.

If the number after .ps is not one of these sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, troff reverts to the previous size given. Troff begins with point size 10.

The point size can also be changed in the middle of a line, or even a word, with the in-line command \s followed by a legal point size. For example, \s10 stands for point size 10. An exception to legal point size is \s0, which causes the size to revert to its previous value.

Relative size changes are also useful. For example,

```
\s-2ZEUS\s+2
```

temporarily decreases the size, whatever it is, 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 is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines. This 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, the vertical spacing should be about 20% bigger than the character size. By default, troff uses 10 as a vertical space on 12-point type, referred to as "10 on 12."

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. When used without arguments, .ps and .vs revert to the previously defined size and vertical spacing.

The command .sp is used to get extra vertical space. It generates one extra blank line (one .vs, whatever that has been set to). To alter that space, use the .sp command. So,

```
.sp 2i
```

means "two inches of vertical space,"

```
.sp 2p
```

means "two points of vertical space," and

```
.sp 2
```

means "two vertical spaces," that is, two of whatever .vs is set to. Troff also uses decimal fractions. For example,

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and after most commands that deal with physical dimensions.

SECTION 3
FONTS AND SPECIAL CHARACTERS

With troff and the typesetter, four different fonts can be on-line at one time. Normally, three fonts (Times Roman, Italic, and Bold) and one collection of special characters are permanently mounted.

The Greek mathematical symbols and miscellany of the special font are listed in the Appendix.

Troff prints in Roman unless told otherwise. To switch into bold, use the .ft command

```
.ft B
```

and for italics,

```
.ft I
```

To return to Roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The underline command

```
.ul
```

causes the next input line to be printed in italics. If .ul is followed by a number, it indicates how many lines are to be italicized.

Fonts can also be changed within a line or word with the in-line command \f. To keep the previous font undisturbed, insert extra .ft commands. Only the immediately previous font remains in memory, so it is necessary to restore the previous font after each change. The same is true of .ps and .vs when used without an argument.

The command .fp tells troff what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

indicates that the Helvetica font is mounted on position 3. (For a complete list of fonts, see the troff manual.) Appropriate .fp commands appear at the beginning of the document if the standard fonts are not being used.

To make a document independent of the actual fonts, use font numbers instead of names. For example, \f3 and .ft 3 mean "whatever font is mounted at position 3," and thus work for any setting. Normal settings are Roman font on 1, Italic on 2, Bold on 3, and Special on 4.

Special characters have four-character names beginning with \(. In particular, Greek letters are all of the form \(*-, where - is an upper or lowercase Roman letter similar to the Greek. A complete list of these special names appears in the Appendix.

Some characters are automatically translated into others. For example, the grave ` and acute ' accents (apostrophes) become open and close single quotes. Similarly, a typed minus sign becomes a hyphen (-). To print an explicit minus sign, use backslash minus (\-).

SECTION 4
INDENTS AND LINE LENGTHS

Troff starts with a line length of 6.5 inches. To reset the line length, use the .ll command, as in

```
.ll 6i
```

As with .sp, the actual length is specified in several ways.

The maximum line length provided by the typesetter is 7.5 inches. To use the full width, reset the default physical left margin (page offset), which is normally slightly less than one inch from the left edge of the paper. This is done with the .po command. For example,

```
.po Ø
```

sets the offset to the leftmost point.

The indent command (.in) idents the left margin by some specified amount from the page offset. Using .in to move the left margin in, and .ll to move the right margin to the left, makes offset blocks of text. For example,

```
.in Ø.3i
.ll -Ø.3i
text to be set into a block
.ll +Ø.3i
.in -Ø.3i
```

creates a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

The "+" and "-" change the previous setting by the specified amount rather than overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch long.

With .in, .ll, and .po, the previous value is used if no argument is specified.

To indent a single line, use the temporary indent command .ti. The default unit for .ti, as for most horizontally

oriented commands (.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.) Although inches are usually clearer than ems, ems measure size that is proportional to the current point size. For text that keeps its proportions regardless of point size, use ems for all dimensions. Ems can be specified as scale factors, as in .ti 2.5m.

Lines can also be indented negatively if the indent is currently positive. The command

```
.ti -Ø.3i
```

moves the next line back three-tenths of an inch.

SECTION 5
TABS

Tabs (the ASCII "horizontal tab" character) produce output in columns or 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 with the .ta command. To set stops every inch, for example, use

```
.ta 1i 2i 3i 4i 5i 6i
```

For numeric columns, precede every number by enough blanks to make it line up when typed. For example,

```
.sp
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but has the same width as a digit. When printed, this produces

```
  1       2       3
 40      50      60
700     800     900
```

To fill up tabbed-over space with some character other than blanks, set the "tab replacement character" with the .tc command. For example,

```
.ta 1.5i 2.5i
.tc \(ru        (\(ru is "_")
Name tab Age tab
```

produces

Name_____ Age _____

To reset the tab replacement character to a blank, use .tc with no argument.

SECTION 6
LOCAL MOTIONS:
DRAWING LINES AND CHARACTERS

Troff has a host of commands for placing characters of any size at any place, making it possible to draw special characters or arrange output for a particular appearance.

If eqn is not used, subscripts and superscripts are most easily done with the half-line local motions \u and \d. To go back up the page half a point-size, insert a \u at the desired place; to go down, insert a \d. \u and \d must always be used in pairs.

The \v command requests an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in (amount).

There are many other ways to specify the amount of motion. The commands

```
\v'Ø.1i'
\v'3p'
\v'-Ø.5m'
```

are all legal. The scale specifier i, p, or m goes inside the quotes.

Since troff 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, must always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions can be generated using \h. The default scale factor, however, is ems instead of line spaces. As an example,

```
\h'-Ø.1i'
```

causes a backward motion of a tenth of an inch.

Frequently, \h is used with the width function \w to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of "thing" in machine units (1/432 inch or 1/6 point). All troff computations are done in these units. To move horizontally the width of x, enter

```
\h'\w'x'u'
```

There are also several special-purpose troff commands for local motion. The \Ø is an unpaddable white space of the same width as a digit. "Unpaddable" means that it is never widened or split across a line-by-line justification and filling. There is also the slash-blank (\ ), an unpaddable character the width of a space. The \| is half that width, \^ is one-quarter of the width of a space, and \& has zero width. This last command is useful in entering a text line that otherwise begins with a period (.).

The command \o , used in

```
\o'set of characters'
```

causes up to nine characters to be overstruck, centered on the widest. (See Accents, Section 15.)

Overstrikes can be generated with another special convention, \z, the zero-motion command. \zx suppresses the normal horizontal motion after printing the single character x, so another character can be printed on top of it. Although sizes change within \o, it centers the characters on the widest. There is no horizontal or vertical motion, so \z is the only way to achieve this.

Troff also has a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. The \l'1i' draws a line one inch long, (_____). The length can be followed by any character. For example, \l'Ø.5i.' draws a half-inch line of dots (.....). The construction \L is entirely analogous, but draws a vertical line instead of a horizontal one.

SECTION 7
STRINGS

With troff, it is possible to store an arbitrary collection of text in a "string," and use the string name as shorthand for its contents. A reference to a string is replaced by whatever text defined the string. Strings are defined with the command .ds. The line

```
.ds e \o"e\'"
```

defines the string e to have the value \o"e\'"

String names can be either one or two characters long. They are referred to by * for one-character names or *(xy for two-character names.

If a string must begin with blanks, define it as

```
.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 can be several lines long; if troff encounters a \ at the end of any line, it is deleted and the next line is added to the current one. So, to define a long string, end each line except the last with a backslash:

```
.ds xx this \
is a very \
long string
```

SECTION 8
INTRODUCTION TO MACROS

In its simplest form, a macro is a shorthand notation similar to a string. To start every paragraph with a space and a temporary indent of two ems, use

```
.sp
.ti +2m
```

To save typing, collapse these into one shorthand line. A troff "command"

```
.lp
```

is treated by troff exactly as

```
.sp
.ti +2m
```

The .lp is called a macro. Tell troff what .lp means by defining it with the .de command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (.lp for paragraph). The last line (..) marks the end of the definition. In between is the text that is inserted whenever troff sees the "command" or macro call .lp.

A macro can contain any combination of text and formatting commands.

The definition of .lp has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is important. Not only does it save typing, but it makes later changes much easier. Suppose that the paragraph indent is too small, the vertical space is too big, and Roman font should be forced. Instead of changing the whole document, change the definition of .lp to

```
.de PP  \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere .lp appears.

The \" is a troff command that causes the rest of the line to be ignored. It is used here to add comments to the macro definition.

Another example of macros starts and ends a block of offset, unfilled text. The following commands define the amount of indentation:

```
.de BS  \" start indented block
.sp
.nf
.in +Ø.3i
..
.de BE  \" end indented block
.sp
.fi
.in -Ø.3i
..
```

Using .BS and .BE to surround the text

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

produces an indented and aligned block. The text is indented by .in +Ø.3i instead of .in Ø.3i. Thus it is possible to nest uses of .BS and .BE to get blocks within blocks.

To change the indent to Ø.5i, change only the definitions of .BS and .BE.

SECTION 9
TITLES, PAGES, AND NUMBERING

Suppose that an application calls for a title with the following format at the top of each page:

left top center top right top

It is necessary to define what the actual title is and how and when to print it. Taking these in reverse order, first define a macro .NP (for new page) to process titles at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp Ø.5i
.tl 'left top'center top'right top'
'sp Ø.3i
..
```

To make sure this starts at the top of a page, issue a "begin page" command ('bp) to skip to top-of-page. Space down half an inch, print the title, and space another Ø.3 inches.

To ask for .NP at the bottom of each page, use the "when" command (.wh):

```
.wh  -1i  NP
```

No "." is used before NP because this is simply the name of a macro, 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 command appears in the input outside the definition of .NP; typically the input is

```
.de NP
...
..
.wh -1i NP
```

As text is actually being output, troff 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 activated. .NP causes a skip to the top of the next page (activated by the 'bp), then prints the title with the defined margins.

The commands .sp and .bp cause a break. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. Using ' for a command tells troff that the output line currently being filled should not be forced out before the space or new page.

Here is a list of commands that cause a break.

```
.bp   .br   .ce   .fi   .nf   .sp   .in   .ti
```

Be careful when inserting breaks near changes of fonts or point sizes to prevent the special font or size from being carried over to the new page (Section 13). 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 the length is changed with the .lt command.

There are several ways to solve the problems of point sizes and fonts in titles. For the simplest applications, change .NP to set the proper size and font for the title, then restore the previous values, as follows:

```
.de NP
'bp
'sp Ø.5i
.ft R          \" set title font to roman
.ps 1Ø  \" and size to 1Ø point
.lt 6i  \" and length to 6 inches
.tl 'left'center'right'
.ps     \" revert to previous size
.ft P   \" and to previous font
'sp Ø.3i
..
```

This version of .NP does not work if the fields in the .tl command contain size or font changes. To solve that problem requires troff's environment mechanism (Section 13).

To get a footer at the bottom of a page, modify .NP to do some processing before the 'bp command. Alternatively, split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page.

Output page numbers start at one and are computed automatically as each page is produced. No numbers are printed unless they are explicitly requested. To get page numbers printed, include the character % in the .tl line at the position where the number is to appear, for example,

```
.tl "- % -"
```

Set the page number at any time with either .bp n, which immediately starts a new page numbered n, or with .pn n, which sets the page number for the next page but does not skip to the new page. Again, .bp +n sets the page number to n more than its current value; .bp means .bp +1.

SECTION 10
NUMBER REGISTERS AND ARITHMETIC

Troff has a facility called number registers for doing arithmetic and for defining and using variables with numeric values. Number registers, like strings and macros, can be useful in setting up a document in addition to serving for arithmetic computation.

Like strings, number registers have one or two-character names. They are set by the .nr command and are referenced anywhere by \nx (one-character name) or \n(xy (two-character name).

Among the predefined number registers maintained by troff are

% 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

.s and .f
for the current size and font. The font is a number from 1 to 4. Any of these can be used in computations, and all except .s and .f can be changed with .nr.

As an example of how to use the number registers, the -ms macro package defines most significant parameters in terms of the values of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, enter:

```
.nr PS 9
.nr VS 11
```

The paragraph macro .lp is 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 sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Two backslashes are required to quote a quote. When troff originally reads the macro definition, it uses one backslash. To ensure that another is left in the definition when the macro is used, put two backslashes in the definition. 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.

An extra layer of backslashes is required for \n, *, \$ (Section 12), and \ itself; \s, \f, \h, \v do not need an extra backslash because they are immediately converted by troff to an internal code.

Arithmetic expressions can appear anywhere that a number is expected. For example,

```
.nr PS \\n(PS-2
```

decrements PS by 2. Expressions can use the arithmetic operators +, -, *, /, % (mod), the relational operators >, >=, <, <=, =, and |= (not equal), and parentheses.

The arithmetic examples given thus far have been straightforward. More complicated expressions have inherent complications. First, number registers hold only integers, and troff arithmetic uses truncating integer division. Second, 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 and scale indicators such as p, i, and m can occur anywhere in an expression. Although integer division causes truncation, each number and its scale indicator is converted to machine units before any arithmetic is done. This means that 1i/2u evaluates to Ø.5i correctly.

The scale indicator u often has to appear when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
^11 7/2i
```

appears to mean 3 1/2 inches. However, when translated into machine units, it becomes zero because the default units for

horizontal parameters are ems. The correct entry for 3 1/2 inches is:

```
.ll 7i/2u
```

Attach a scale indicator to every number, even constants, and avoid problems such as this.

For arithmetic done within an .nr command, there is no implication of horizontal or vertical dimension, so the default units are "units," and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

is interpreted as expected, so long as the u is put on the .ll command.

SECTION 11
MACROS WITH ARGUMENTS

Macros can change from one use to the next according to parameters supplied as arguments. To do this, two things are required. First, when the macro is defined, indicate that some parts of it will be provided as arguments. Second, when the macro is called, provide actual arguments to be plugged into the definition.

As an illustration, define a macro .SM that prints its argument two points smaller than the surrounding text. The definition of .SM is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol \\$n refers to the nth argument that the macro was called with. Thus \\$1 is the string to be placed in a smaller point size when .SM is called.

The following definition of .SM 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. It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading punctuation.

The number register .$ contains the number of arguments that a macro was called with.

Two backslashes are needed with the \\$n commands. One protects the other when the macro is being defined. For example, a macro called .SH produces section headings with the sections numbered automatically, and the title in bold in a smaller size. The format is

```
.SH "Section title ..."
```

If the argument to a macro is to contain blanks, it must be surrounded by double quotes, unlike a string, where only one

leading quote is permitted.

The following is the definition of the .SH macro:

```
.nr SH Ø            \" initialize section number
.de SH
.sp Ø.3i
.ft B
.nr SH \\n(SH+1     \" increment number
.ps \\n(PS-1        \" decrease PS
\\n(SH.  \\$1       \" number. title
.ps \\n(PS          \" restore PS
.sp Ø.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time, just before it is used. (A number register can have the same name as a macro without conflict, but a string cannot.)

In this example, the command \\n(SH is used instead of \n(SH, and \\n(PS is used instead of \n(PS. If \n(SH had been used, the value of the register would be determined at the time the macro is defined, not at the time it is used. Similarly, by using \\n(PS, the point size is determined at the time the macro is called.

As an example that does not involve numbers, use the .NP macro from Section 9:

```
.tl 'left'center'right'
```

Make these into parameters by using

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

The title comes from three strings called LT, CT, and RT. If these strings are empty, the title is a blank line. Normally, CT is set with

```
.ds  CT  - \% -
```

to give the page number between hyphens. The strings can be defined according to the requirements of the application.

SECTION 12
CONDITIONALS

To leave two extra inches of space before Section 1 but nowhere else, use a conditional (.if) to test the .SH macro. The following command causes space to be added only if the section number is one.

```
.if \\n(SH=1 .sp 2i        \" first section only
```

The condition after the .if 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. If the condition is false, zero, or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before Section 1. One possibility is to define a macro S1 and invoke it before Section 1.

```
.de S1
---  processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate method is to use the extended form of .if, as follows:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown.

A condition can be negated by preceding it with !; thus

```
.if !\\n(SH>1 .S1
```

produces the same output as the previous commands.

There are several other conditions that can be tested with .if. For example, .if can determine whether the current page is even or odd. The command

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are t and n, which indicate whether the formatter is troff or nroff.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons can be made with .if. The command

```
.if 'string1'string2' stuff
```

does "stuff" if string1 is the same as string2. The character separating the strings can be anything that is not contained in either string. The strings themselves can reference strings with * and arguments with \$.

SECTION 13
ENVIRONMENTS

There is a potential problem when page boundaries are crossed. Parameters like size and font for a page title can be different from those in effect in the text when the page boundary occurs. Troff provides a very general method of dealing with this and similar situations. Three "environments" allow parameters to be set independently. Thus, the titling problem can be solved by processing the main text in one environment and titles in a separate one with its own parameters.

The command .ev n shifts to environment n; n must be Ø, 1, or 2. The command .ev with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments can be nested and unwound.

In the titling problem, process the main text in environment Ø, where troff begins by default. Modify the new page macro .NP to process titles in environment one, as follows:

```
.de NP
.ev 1   \" shift to new environment
.lt 6i  \" set parameters here
.ft R
.ps 1Ø
... any other processing ...
.ev          \" return to previous environment
..
```

The following parameters are part of an environment:

.ps	point size
.ss	space size
.ft	current font
.fi	output line filling enabled
.nf	output line filling disabled
.ad	output line adjustment enabled
.na	output line adjustment disabled
.ce	output line centering
.vs	vertical spacing
.ls	line space
.ll	line length
.in	indentation
.ti	temporary indent
.it	input line trap
.ta	tab settings
.tc	tab character
.lc	leader character
.ul	underline
.cu	continuous underline
.cc	control character
.c2	no break control character
.nh	no hypenation mode
.hy	hypenation mode
.hc	hypenation indicator char.
.lt	length of title
.nm	number mode on/off
.nn	number mode no number
.mc	margin character

SECTION 14
DIVERSIONS

When doing page layout, it is often necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example; the text of the footnote usually appears in the input before its place on the page is reached.

A troff mechanism called a diversion determines the size of a footnote and allows room for it on the page. Any part of the output can be diverted into a macro instead of being printed, and the macro can be put back into the input.

The command .di xy begins a diversion. All subsequent output is collected into the macro xy until the command .di with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last complete diversion is contained in the built-in number register dn.

The example that follows implements a "keep-release" operation so that text between the commands .KS and .KE is not split across a page boundary (as for a figure or table). When a .KS is encountered, the output is diverted and its size is determined. When a .KE is encountered, the diverted text is evaluated and is printed on the current page if it fits, or at the top of the next page if it does not.

```
.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 doesn't fit
.nf                         \" bring it back in no-fill
.XX                         \" text
.ev                         \" return to normal environment
..
```

Number register nl is the current position on the output page. Since output is being diverted, this remains at the value it had when the diversion started. dn is the amount of text in the diversion; .t is another built-in register to show the distance to the next bottom margin of the page (trap). If the diversion is large enough to go past the trap, the .if is satisfied, and a .bp is issued. In either case, the diverted output is then brought back with .XX. It is essential to bring it back in no-fill mode so troff does no further processing on it.

UUCP INSTALLATION*

* This information is based on an article originally written by D. A. Nowitz, Bell Laboratories.

10/14/83

Preface

This document gives the system administrator/installer a detailed description of **uucp.** The operation of each program in the **uucp** system, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system are discussed in this document.

Table of Contents

SECTION 1
INTRODUCTION

1.1. General

Uucp is a group of programs that permit communication between ZEUS systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution.

Each system participating in the **uucp** network has a spool directory (/usr/spool/uucp) that stores work to be done. There are three types of files used for the execution of work: data files, copy command files, and execution files. Data files contain data to be transferred to remote systems. Copy command files contain the directions for file transfers between systems. Execution files contain the directions for ZEUS command executions that involve the resources of one or more systems.

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

uucp(1) Creates copy command files and gathers data files in the spool directory for the transmission of files.

uux(1) Creates copy command files, executes files, and gathers data files for the remote execution of ZEUS commands.

uucico Executes the copy command files for data transmission.

uuxqt Executes ZEUS execution files.

The secondary programs are:

uulog(1) Updates the log file with new entries, reports the status of **uucp** requests.

uuclean(1)
Removes old files from the spool directory.

uusub(M) Defines a **uucp** subnetwork, monitors traffic and connection statistics.

uustat(1) **Uucp** job status inquiry and control.

uuname(1) Lists the names of known **uucp** systems.

uuto(1) Public file copy to specific user.

uupick(1) Accepts or reviews files sent to user by **uuto.**

1.2. Security

The **uucp** system, if left unrestricted, lets anyone execute any command and copy in or out any file that is readable/writable by the **uucp** login user. Necessary precautions should be taken as required by the local implementation.

There are security features available other than the normal file-mode protections that must be set up by the installer of the **uucp** system.

- The login for **uucp** does not spawn a standard shell; instead, the **uucico** program is started. The work can be done only through **uucico.**

- A path check is performed on file names that are to be sent or received. The USERFILE supplies the information for these checks. The USERFILE can also be set up to require call-back for certain login IDs. (See Section 3.3.1 for file description.)

- A conversation sequence count can be set up so that the called system can verify the caller's identity (See Section 4.2 for file description).

- The **uuxqt** program comes with a file of commands (L-cmds) that it allows remote systems to execute. A path shell statement (PATH=/bin:/usr/bin;) is prepended to the command line by **uuxqt.**

- The L.sys file must be owned by **uucp** and have mode Ø4ØØ to protect the phone numbers and login information for remote sites. Programs **uucp, uucico, uux,** and **uuxqt** must also be owned by **uucp** and have the setuid bit set.

- L-cmds specifies those commands that may be executed by **uux** from remote systems. L-cmds should be owned by **uucp** and have mode 0444 to protect against unauthorized modification (see Section 3.2.1 of this manual entry for file description).

SECTION 2
THE UUCP PROGRAMS

2.1. Uucp

The **uucp** command is the primary interface with the system. It sets up file copying and is similar to the ZEUS copy command, **cp**(1). **Uucp** is invoked by the command line

uucp [options] ... source ... destination

where source and destination contain the prefix system name specifying the system on which files reside or the system on which the files will be copied.

2.1.1. Options: The following options are valid for the **uucp** command:

-d When necessary, make directories for copying the file (default).

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

-c Use the specified source for the transfer. Do not copy source files to the spool directory (default).

-C Copy the source file to the spool directory.

-esystem
Send this job to system to execute. (Note that this will only work when the system allows **uuxqt** to execute a **uucp** command (See Section 3.2.1).

-gletter
Insert letter as the grade in the name of the copy command file. Lower letters are higher priority. This can be used to change the order of work for a specified system (default value is 'n').

-m Send mail on completion of the work.

-nuser
Notify user on the remote system that a file has been sent.

The following options are used primarily for debugging:

created. The file is transmitted from the indicated source. The entry fields are as follows:

- S
- The full path name of the source file.
- The full path name of the destination or ~user/filename.
- The user's login name.
- A minus sign (-) followed by an option list.
- The name of the data file in the spool directory.
- The file mode bits of the source file in octal print format (mode **Ø666**).
- The user on the remote system to be notified if -n option was used.

Types 4 and 5
: **Uucp** generates a **uucp** command and sends it to the remote machine; the remote **uucico** executes the **uucp** command.

Type 6
: This occurs when the -e option is used. The **uux** facility is used to create and send the request.

2.2. Uux

The **uux** command sets up the execution of a command if the execution system and some of the files are remote. The syntax of the **uux** command is

uux [-] [option **] ...** command-string

where command-string is composed of one or more arguments. All special shell characters such as `<`, `>`, `|`, and `^` must be quoted, either by quoting the entire command string or by quoting the character as a separate argument. Within command-string, the command and file names can contain a system-name! prefix. All arguments must contain an exclamation point (!) if they are to be treated as files and copied to the execution system. The minus sign (-) indicates that the standard input for command-string must come from the standard input of the **uux** command. The options, which are for debugging, are the following:

-r Do not start **uucico** or **uuxqt** after queuing the job.

-xnum Num is the level of debugging output desired.

The command

pr abc | uux - usg!lpr -c

sets up the output of **pr abc** as standard input to a line printer (**lpr**(1)) command to be executed on system usg. Note the use of the -c option to **lpr**.

Uux generates an execute file containing the names of the files required for execution, the user's login name, the destination of the standard output, and the command to be executed. The execute file is placed in the spool directory for local execution or is sent to the remote system using a generated send command (Type 3 in Section 2.1.3).

Uux generates receive command-files (Type 2) for files that are not on the execution system. These command files are placed on the execution machine and executed by the **uucico** program if the local system has permission to place files in the remote spool directory.

The execute file is processed by the **uuxqt** program on the execution system. It is composed of several lines, each containing an identification character and one or more arguments. There is no set order for the lines and not all must be present. The following sections describe each line.

2.2.1. User Line: The user line is as follows

U user system

where user and system are the requester's login name and system.

2.2.2. Required File Line: The required file line is

F filename realname

where filename is the generated name of an execution system file and realname is the last part of the file name, which contains no path information. Zero or more of these lines are present in the execute file. The **uuxqt** program checks for the existence of all required files before the command is executed.

2.2.3. Standard Input Line: The standard input line is

I filename

The standard input is either specified by a < in the command-string or obtained from the standard input of the **uux** command if the - option is used. If the standard input is not specified, /dev/null is used.

2.2.4. Standard Output Line: The standard output line is

O filename system-name

The standard output is specified by a > within the command string. If the standard output is not specified, /dev/null is used. The use of >> is not implemented.

2.2.5. Command Line: The command line is

C command arguments ...

The arguments are specified in the command string. The standard input and standard output do not appear on this line. All required files are moved to the execution directory (a subdirectory of the spool directory) and the ZEUS command is executed using **sh**(1) (/bin/sh). An execution system checks the command against its L-cmds file to see if it is allowable. In addition, a shell path statement (PATH=/bin:/usr/bin;) is prepended to the command line as specified in the **uuxqt** program.

To determine what commands are allowed by a remote system, use **uucp** to copy the L-cmds file from the remote system's program directory to the local system. This copy will fail if the remote system's USERFILE restricts access to its program directory.

After execution, the standard output is copied or set up to be sent to the designated place.

2.3. Uucico

The copy in, copy out (**uucico**) program performs the following communications functions between two systems:

- Scans the spool directory for work.

- Places a call to a remote system.
- Negotiates a line protocol to be used.
- Executes all requests from both systems.
- Logs work requests and work completions.

Uucico can be started by a system daemon, by one of the **uucp**, **uux**, **uuxqt**, or **uucico** programs, directly by the user, or by a remote system. The **uucico** program must be specified as the shell field in the /etc/**passwd** file for the **uucp** logins.

When started by a remote system, the local program is in "SLAVE" mode. When started by any other method, the local program is in "MASTER" mode, and a connection is made to a remote system.

The MASTER mode operates in one of two ways. If a system name is specified, that system is called and work is done only for that system. If a system name is not specified, the program scans the spool directory for systems to call.

The **uucico** program is generally started by another program. There are several options used for execution:

-**r**1 Start the program in MASTER mode. This is used when **uucico** is started by a program or **cron**(M) shell.

-**s**sys Do work only for system sys. If -**s** is specified, a call to the specified system is made even if there is no work for system sys in the spool directory. This program is useful for polling systems that do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

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

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

The following subsections describe the major steps within the **uucico** program.

2.3.1. Scan for Work: The names of the work-related files in the spool directory have the format

type . *system-name grade number*

where *type* is an uppercase `C` (copy command file), `D` (data file), or `X` (execute file); *system-name* is the remote system, *grade* is a character, and *number* is a padded four-digit sequence number obtained from /usr/lib/uucp/SEQF. For example, the file

C.res45nØØ31

is a copy command file for a file transfer between the local machine and the res45 machine.

The scan for work is done by looking through the spool directory for copy command files (files with prefix C.). A list is created for all systems to be called; **uucico** then calls each system and processes all copy command files.

2.3.2. Call Remote System: The call is made using information from several files that reside in the **uucp** program directory (/usr/lib/uucp). At the beginning of the call process, a lock is set on the system being called to prevent multiple conversations between the two systems in /usr/spool/uucp (see section 4.6).

The system name is found in the L.sys file. The information contained for each system is the system name, the time to call the system (days-of-week and times-of-day), the device or device type to be used for the call, the line speed, the phone number if the device or device type is an automatic call unit (ACU) or the device name if the device or device type is not ACU, and the login information.

The time field is checked against the present time to see if the call should be made. The time field can alternately contain the string "passive" to denote that the remote system must initiate the conversation and cannot be called. In

SECTION 2
THE UUCP PROGRAMS

2.1. Uucp

The **uucp** command is the primary interface with the system. It sets up file copying and is similar to the ZEUS copy command, **cp**(1). **Uucp** is invoked by the command line

uucp [options] ... source ... destination

where source and destination contain the prefix system name specifying the system on which files reside or the system on which the files will be copied.

2.1.1. Options: The following options are valid for the **uucp** command:

-d When necessary, make directories for copying the file (default).

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

-c Use the specified source for the transfer. Do not copy source files to the spool directory (default).

-C Copy the source file to the spool directory.

-esystem
Send this job to system to execute. (Note that this will only work when the system allows **uuxqt** to execute a **uucp** command (See Section 3.2.1).

-gletter
Insert letter as the grade in the name of the copy command file. Lower letters are higher priority. This can be used to change the order of work for a specified system (default value is 'n').

-m Send mail on completion of the work.

-nuser
Notify user on the remote system that a file has been sent.

The following options are used primarily for debugging:

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

-sdir Use directory dir for the spool directory.

-xnum Num is the desired level of debugging output. Num ranges from one through nine; higher numbers give more information.

2.1.2. Sources and Destinations: If the destination is a directory name, the file name is taken from the last part of the source name. If the directory exists, it must be writable by everybody. If the destination is a directory and the -g option is used, then the directory name must be followed by a '/'. The source name can contain special shell characters such as `?`, `*`, `[`, and `]`. If a source argument has a system-name! prefix indicating a remote system, the file name expansion is performed on the remote system.

The command

uucp *.c usg!/usr/dan

copies all files with names ending in .c to the /usr/dan directory on system usg. The exclamation point must be escaped when using the cshell, e.g.:

usg\!/usr/dan

The source and destination names can also contain a `~user` prefix to refer to the login directory on the specified system. Filenames beginning with `~` refer to the public directory (/usr/spool/uucppublic) on the remote system. The current directory is prepended to the file name for names with partial path names. File names with `../` are not permitted.

The command

uucp usg!~dan/*.h ~dan

copies to dan's local login directory files in dan's login directory on system usg whose names end with '.h'.

2.1.3. Types of Work: For each source file, the **uucp** program checks the source and destination file names and the system-part of each to classify the work into one of five types:

1. Copy source to destination on local system.

2. Receive files from other systems.

3. Send files to remote systems.

4. Send files from a remote system to another remote system.

5. Receive files from remote systems when the source contains special shell characters, such as `?`, `*`, `[`, and `]`.

After the work has been set up in the spool directory, the **uucico** program contacts the other system to execute the work unless the **-r** option is specified or the system is designated "passive" (see Section 3.3.2).

Type 1 Copy source to destination on the local system. The **-d** and the **-m** options are not valid in type-1 operations.

Type 2 A one-line copy command file is created for each file requested and is placed in the spool directory with the following fields, each separated by a blank. All copy command files and execution files use a blank as the field separator.

- R
- The full path name of the source or a ~user/pathname; the `~user` part is expanded on the remote system.
- The full path name of the destination file; if the `~user` notation is used, it is immediately expanded to the user's login directory.
- The user's login name.
- A minus sign (-) followed by an option list; only the **-m** and **-d** options appear in this list.

Type 3 For each source file, a copy command file is created. The source file is copied into a data file in the spool directory. A **-c** option on the **uucp** command prevents the data file from being

created. The file is transmitted from the indicated source. The entry fields are as follows:

- S
- The full path name of the source file.
- The full path name of the destination or ~user/filename.
- The user's login name.
- A minus sign (-) followed by an option list.
- The name of the data file in the spool directory.
- The file mode bits of the source file in octal print format (mode **0666**).
- The user on the remote system to be notified if -n option was used.

Types 4 and 5
: **Uucp** generates a **uucp** command and sends it to the remote machine; the remote **uucico** executes the **uucp** command.

Type 6
: This occurs when the -e option is used. The **uux** facility is used to create and send the request.

2.2. Uux

The **uux** command sets up the execution of a command if the execution system and some of the files are remote. The syntax of the **uux** command is

uux [-] [option] ... command-string

where command-string is composed of one or more arguments. All special shell characters such as `<`, `>`, `|`, and `^` must be quoted, either by quoting the entire command string or by quoting the character as a separate argument. Within command-string, the command and file names can contain a system-name! prefix. All arguments must contain an exclamation point (!) if they are to be treated as files and copied to the execution system. The minus sign (-) indicates that the standard input for command-string must come from the standard input of the **uux** command. The options, which are for debugging, are the following:

-r Do not start **uucico** or **uuxqt** after queuing the job.

-x<u>num</u> <u>Num</u> is the level of debugging output desired.

The command

pr abc | uux - usg!lpr -c

sets up the output of **pr abc** as standard input to a line printer (**lpr**(1)) command to be executed on system usg. Note the use of the -c option to **lpr**.

Uux generates an execute file containing the names of the files required for execution, the user's login name, the destination of the standard output, and the command to be executed. The execute file is placed in the spool directory for local execution or is sent to the remote system using a generated send command (Type 3 in Section 2.1.3).

Uux generates receive command-files (Type 2) for files that are not on the execution system. These command files are placed on the execution machine and executed by the **uucico** program if the local system has permission to place files in the remote spool directory.

The execute file is processed by the **uuxqt** program on the execution system. It is composed of several lines, each containing an identification character and one or more arguments. There is no set order for the lines and not all must be present. The following sections describe each line.

2.2.1. User Line: The user line is as follows

U <u>user</u> <u>system</u>

where <u>user</u> and <u>system</u> are the requester's login name and system.

2.2.2. Required File Line: The required file line is

F <u>filename</u> <u>realname</u>

where <u>filename</u> is the generated name of an execution system file and <u>realname</u> is the last part of the file name, which contains no path information. Zero or more of these lines are present in the execute file. The **uuxqt** program checks for the existence of all required files before the command is executed.

2.2.3. Standard Input Line: The standard input line is

I filename

The standard input is either specified by a < in the command-string or obtained from the standard input of the **uux** command if the - option is used. If the standard input is not specified, /dev/null is used.

2.2.4. Standard Output Line: The standard output line is

O filename system-name

The standard output is specified by a > within the command string. If the standard output is not specified, /dev/null is used. The use of >> is not implemented.

2.2.5. Command Line: The command line is

C command arguments ...

The arguments are specified in the command string. The standard input and standard output do not appear on this line. All required files are moved to the execution directory (a subdirectory of the spool directory) and the ZEUS command is executed using **sh**(1) (/bin/sh). An execution system checks the command against its L-cmds file to see if it is allowable. In addition, a shell path statement (PATH=/bin:/usr/bin;) is prepended to the command line as specified in the **uuxqt** program.

To determine what commands are allowed by a remote system, use **uucp** to copy the L-cmds file from the remote system's program directory to the local system. This copy will fail if the remote system's USERFILE restricts access to its program directory.

After execution, the standard output is copied or set up to be sent to the designated place.

2.3. Uucico

The copy in, copy out (**uucico**) program performs the following communications functions between two systems:

- Scans the spool directory for work.

- Places a call to a remote system.
- Negotiates a line protocol to be used.
- Executes all requests from both systems.
- Logs work requests and work completions.

Uucico can be started by a system daemon, by one of the **uucp**, **uux**, **uuxqt**, or **uucico** programs, directly by the user, or by a remote system. The **uucico** program must be specified as the shell field in the /etc/**passwd** file for the **uucp** logins.

When started by a remote system, the local program is in "SLAVE" mode. When started by any other method, the local program is in "MASTER" mode, and a connection is made to a remote system.

The MASTER mode operates in one of two ways. If a system name is specified, that system is called and work is done only for that system. If a system name is not specified, the program scans the spool directory for systems to call.

The **uucico** program is generally started by another program. There are several options used for execution:

-rl Start the program in MASTER mode. This is used when **uucico** is started by a program or **cron**(M) shell.

-s*sys* Do work only for system *sys*. If **-s** is specified, a call to the specified system is made even if there is no work for system *sys* in the spool directory. This program is useful for polling systems that do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

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

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

The following subsections describe the major steps within the **uucico** program.

2.3.1. Scan for Work: The names of the work-related files in the spool directory have the format

type . *system-name* *grade* *number*

where *type* is an uppercase `C` (copy command file), `D` (data file), or `X` (execute file); *system-name* is the remote system, *grade* is a character, and *number* is a padded four-digit sequence number obtained from /usr/lib/uucp/SEQF. For example, the file

C.res45nØØ31

is a copy command file for a file transfer between the local machine and the res45 machine.

The scan for work is done by looking through the spool directory for copy command files (files with prefix C.). A list is created for all systems to be called; **uucico** then calls each system and processes all copy command files.

2.3.2. Call Remote System: The call is made using information from several files that reside in the **uucp** program directory (/usr/lib/uucp). At the beginning of the call process, a lock is set on the system being called to prevent multiple conversations between the two systems in /usr/spool/uucp (see section 4.6).

The system name is found in the L.sys file. The information contained for each system is the system name, the time to call the system (days-of-week and times-of-day), the device or device type to be used for the call, the line speed, the phone number if the device or device type is an automatic call unit (ACU) or the device name if the device or device type is not ACU, and the login information.

The time field is checked against the present time to see if the call should be made. The time field can alternately contain the string "passive" to denote that the remote system must initiate the conversation and cannot be called. In

this case, the remaining fields are ignored.

The phone number field can contain abbreviations (for example, `mh`, `py`, or `boston`), that get translated into dial sequences using the L-dialcodes file. The same phone number can then be stored at every site, despite local variations in telephone services and dialing conventions.

The L-devices file is scanned using the device and line speed from the L.sys file to find an available device for the call. The program tries all devices that satisfy the device types and line speed until the call is made, or until no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of **uucico** does not attempt to use it. If the call is complete, the login information from the L.sys file is used to log in to the remote system. A command is then sent to the remote system to start the **uucico** program.

The conversation between the two **uucico** programs begins with a handshake started by the called (SLAVE) system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from the MASTER is verified by the SLAVE and, if acceptable, protocol selection begins. The SLAVE can also reply with a `call-back required` message and the current conversation is terminated.

2.3.3. Line Protocol Selection: The remote system sends the message

Pproto-list

where proto-list is a string of characters, each representing a line protocol.

The calling program checks proto-list for a letter corresponding to an available line protocol and returns a use-protocol message. The use-protocol message is

Ucode

where code is either a one-character protocol letter or N, which means there is no common protocol.

The initial role (MASTER or SLAVE) for the work processing is the mode in which each program starts. The MASTER is specified by the **-rl uucico** option.

There are five messages used during the work processing, each specified by the first character of the message. They are

`S` Send a file

`R` Receive a file

`C` Copy complete

`X` Execute a **uucp** command

`H` Hangup

The MASTER sends `R`, `S`, and `X` messages until all work from the spool directory is complete. It then sends an `H` message. The SLAVE replies with `SY`, `SN`, `RY`, `RN`, `HY`, `HN`, `XY`, or `XN`, corresponding to "yes" or "no" for each request.

The basis for the send and receive replies is the access permission for the requested file/directory obtained by using the userfile and read/write permissions of the file/directory. A copy-complete message is sent by the receiver of the file after each file is copied into the spool directory of the receiving system. The message `CY` is sent if the file has been successfully copied from the temporary spool file to the actual destination. Otherwise, a `CN` message is sent. In the case of `CN`, the transferred file is in the spool directory with a name beginning with `TM`. The requests and results are logged on both systems.

The SLAVE program determines the hangup response by a work scan of the spool directory. If work for the remote system exists in the SLAVE's spool directory, an `HN` message is sent, and the programs switch roles. If no work exists, an `HY` response is sent.

2.3.4. Conversation Termination: When an `HY` message is received by the MASTER, it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final `OO` message to the other. The original SLAVE program cleans up and terminates. The MASTER calls other systems and processes work, or terminates if a **-s** option is specified.

2.4. Uuxqt

The **uucp** command execution (**uuxqt**) program executes execute files generated by **uux**. The **uuxqt** program is started by either the **uucico** or **uux** programs. The program scans the spool directory for execute files (prefix `X`.). Each execute file is checked to see if all the required files are available. If so, the command line or send line is executed.

Uuxqt is initiated by executing the shell with the -c option after the appropriate standard input and standard output have been opened. If the standard output is specified, the program creates a send command or copies the output file as designated.

2.5. Uulog

The **uucp** programs create individual log files for each program invocation. Periodically, **uulog** can be executed to append these files to the system log file. This method of logging minimizes file locking of the log file during program execution.

The **uucp** log inquiry (**uulog**) program merges the individual log files and outputs specified log entries. The output request is specified by the following options:

-s*sys* Print entries where *sys* is the remote system name.

-u*user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users.

2.6. Uuclean

The **uucp** spool directory cleanup (**uuclean**) program is typically started by the **cron** process. It removes files that are more than three days old from the spool directory. These are usually files for work that could not be completed.

The **uuclean** program should be owned by **uucp** with the setuid bit set (mode 04700).

The options available for **uuclean** are:

-ddir	The directory to be scanned is dir (default is /usr/spool/uucp).
-nhours	Change the aging time from 72 hours to hours hours.
-ppre	Examine files with prefix pre for deletion. Up to ten file prefixes can be specified.
-xnum	Num is the desired level of debugging output.
-m	Send mail to the owner of each file being removed.

SECTION 3
UUCP INSTALLATION

3.1. General

Installing **uucp** under ZEUS requires little effort. The **uucp** files and directories are described here to facilitate tailoring **uucp** to a specific environment.

The following three directories are required for execution (default values appear within parentheses):

program (/usr/lib/uucp) This directory contains the executable system programs and the system files.

spool (/usr/spool/uucp) This spool directory is used during **uucp** execution.

xqtdir (/usr/spool/uucp/.XQTDIR) This directory is used during execution of execute files.

The names program, spool, and xqtdir are used in this section as a shorthand form to represent their corresponding directory path names.

The modes of spool and xqtdir should be mode Ø777; that is, readable, writable, and executable by everyone. The mode of program should be Ø775.

3.2. Files Required for Execution

The five files required for execution must reside in the program directory. The field separator for all files is a space unless otherwise specified.

3.2.1. L-cmds: This file contains the list of commands that the local system will allow a remote system to use via **uuxqt**. If the string `Any` appears first in the file, then no restrictions are imposed. This file should be owned by **uucp** with mode Ø444 to prevent unauthorized changes.

3.2.2. L-devices: This file contains call-unit device and hardwired connection information. The special device files are assumed to be in the /dev directory. The format for each entry is

type line call-unit speed

where

type is a device type such as ACU or DIR. ACU refers to a connection that needs to be dialed. DIR implies a direct connection, but can also be used for automatic dialing units.

line is the device for the line (e.g., tty3).

call-unit is the automatic call unit associated with line (e.g., cuaØ). Hardwired lines have the number "Ø" in this field.

speed is the line speed.

The line

```
ACU culØ cuaØ 3ØØ
```

is set up for a system that has device culØ connected to a call unit cuaØ for use at 3ØØ baud.

The line

```
DIR tty3 Ø 12ØØ
```

is set up for a system that has a hardwired connection to terminal line 3 for use at 12ØØ baud or one that has an autodialer for use at 12ØØ baud connected to terminal line 3.

3.2.3. L-dialcodes: This file contains entries with location abbreviations used in the L.sys file (for example, `py`, `mh`, or `boston`). The entry format is

abb dial-seq

where abb is the abbreviation and dial-seq is the dial sequence to call that location.

The line

```
py 165-
```

is set up in L.sys so that entry `py7777` in L-dialcodes sends 165-7777 to the dial-unit.

3.2.4. SEQF: This file contains the four digit sequence number used to create unique names for **uucp** work files. This file should have mode 666 and reside in the program directory. It can be initialized with the command

```
echo -n 0000 > SEQF
```

3.3. Login/System Names

The login name used by a remote computer to call a local computer must not be the same as the login name of a local user. However, several remote computers can employ the same login name.

Each computer has a unique system name that is transmitted at the start of each call. This name identifies the calling machine to the called machine. The system name transmitted is the name specified during **sysgen** in response to the "network node" prompt. This name is revealed with either the `uuname -l` or `uname -n` commands. See The ZEUS Administrator Manual, Section 5.11 and ZEUS Reference Manual entries **uuname**(1) and **uname**(1).

3.3.1. USERFILE: This file contains user accessibility information. It specifies four types of constraints:

1. Which files can be accessed by a normal user of the local machine.
2. Which files can be accessed from a remote computer.
3. Which login name is used by a particular remote computer.
4. Whether a remote computer should be called back to confirm its identity.

Each line in the file has the following format

user,sys [c] pathname [pathname] ...

where user is the login name of a user on a remote computer, sys is the system name for a remote computer, **c** is the optional call-back flag, and pathname is a path name prefix that is acceptable for user. Pathname must occur once; it can occur more than once (This is indicated by the notation `pathname [pathname] ...`).

The constraints are implemented as follows:

1. When the program is obeying a command stored on the local machine (MASTER mode) the path names allowed are those given for the first line in the user file that has a login name matching the login name of the user who entered the command. If no such line is found, the first line with a null login name is used.

2. When the program is responding to a command from a remote machine (SLAVE mode) the path names allowed are those given for the first line in the file that has a system name matching the system name of the remote machine. If no such line is found, the first one with a null system name is used.

3. When a remote computer logs in, the login name that it uses must appear in the user file. There can be several lines with the same login name, but one of them must either have the name of the remote system or must contain a null system name.

4. If the line matched contains a **c**, the remote machine is called back before any transactions take place.

The line

u,m **/usr/xyz**

allows machine m to log in with name u and request the transfer of files whose names start with **/usr/xyz.**

The line

dan, /usr/dan

allows the ordinary user, dan, to issue commands for files whose names start with /usr/dan.

The lines

u,m **/usr/xyz /usr/spool**
u, **/usr/spool**

allow any remote machine to log in with name u,. If its system name is not m, it can only ask to transfer files whose names start with /usr/spool.

The lines

zeus, /
, /usr

allow any user to transfer files beginning with /usr. The user with login **zeus** can transfer any file.

3.3.2. L.sys: Each entry in this file represents one system that can be called by the local **uucp** programs. The fields are described below.

SYSTEM NAME

The name of the remote system.

TIME

This field indicates the days-of-week and times-of-day when the system is called (for example, MoTuTh0800-1730). Alternatively, the field can contain the string "passive", indicating that only the remote system can initiate a conversation. If the field contains "passive", the remaining fields are ignored.

The day portion can be a list containing

`Su` `Mo` `Tu` `We` `Th` `Fr` `Sa`

or it can be `Wk` for any week-day or `Any` for any day.

The time must be a range of times (for example, 0800-1230). If no time portion is specified, any time of day can be used for the call. Note that a time range that spans 0000 is permitted. For example, 2100-0800 allows calls between 11:00 p.m. and 8:00 a.m.

An optional subfield is available to indicate the minimum time in minutes before a retry following a failed attempt. The subfield separator is a comma. (e.g: `Any,5` means call any time but wait at least 5 minutes after a failure.)

DEVICE

This is either ACU or the hardwired device to be used for the call. For hardwired devices, the last part of the special file name is used (for example, tty0).

SPEED

This is the line speed for the call (for example, 300).

PHONE The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one that appears in the L-dialcodes file (for example, `mh5900`, `boston995-9980`).

For hardwired devices (ie: direct-connect systems), this field contains the same string as the device field.

LOGIN

The login information is given as a series of fields and subfields in the format

expect send [expect send] ...

where expect is the string expected to be read and send is the string to be sent when the expect string is received. Note that the null string, "", is a valid expect string.

The expect field is made up of subfields of the form

expect[-send-expect]...

where the send is sent if the prior expect is not successfully read and the expect following the send is the next expected string.

There are two special names available to be sent during the login sequence. The string `EOT` sends 2 EOT characters and the string `BREAK` tries to send a BREAK character. The BREAK character is simulated using line speed changes and null characters and may not work on all devices and systems.

Typical entries in the L.sys file are

```
sysA Any ACU 300  mh7654 login: uucp ssword: word
sysB Wk,10 tty2 1200 tty2 login: uucp ssword: word
sysC Wk 2300-0800 tty3 1200 tty3 login: uucp sword: word
sysD passive
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

Some example expect-send strings for autodialers are:

```
"" ^M^M $ K DIAL: mh7654 LINE! login: uucp sword: word
```

```
"" ATTD mh7654 CONNECT login: uucp ssword: word
```

The first example is for a Ventel MD212+ and the second is for a Hayes Smartmodem 1200.

Note that the autodialer must be connected to one of the serial ports with a null modem cable. The serial I/O port must be disabled in the /etc/inittab file (see **inittab**(5)); it must be configured as a modem (see **ttyconfig**(M)).

SECTION 4
UUCP ADMINISTRATION

4.1. General

This section describes some events and files that must be administered for the **uucp** system. Some administration can be accomplished by shell files initiated by **crontab** entries. Others require manual intervention. Some sample shell files are given toward the end of this section.

4.2. Sequence Check File

The Sequence Check File (SQFILE) in the program directory contains an entry for each remote system with which conversation sequence checks are to be performed. The initial entry is the system name of the remote system. The first conversation adds two items to the line: the conversation count, and the date/time of the most recent conversation. These items are updated with each conversation. If a sequence check fails, the entry must be adjusted and the corresponding system status file must be removed (see Section 4.5).

4.3. Temporary Data Files

Temporary Data Files (TM) are created in the spool directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where pid is a process-id and ddd is a sequential three-digit number starting at zero for each invocation of **uucico** and incremented for each file received.

After the entire remote file is received, the TM file is moved or copied to the requested destination. If processing is abnormally terminated or if the move or copy fails, the file remains in the spool directory. These unused files must be removed periodically with the **uuclean** program. The command

```
uuclean -pTM
```

removes all TM files more than three days old.

4.4. Log Entry Files

During execution of programs, individual Log Entry Files (LOG files) are created in the spool directory with information about queued requests, calls to remote systems, execution of **uux** commands, and file copy results. These files must be combined into the LOGFILE by using the **uulog** program. The command

uulog

puts the new LOG files at the end of the existing LOGFILE. Options are available to print some or all the log entries after the files are merged. The LOGFILE must be removed periodically since it is copied each time new log entries are put into the file.

The log files are created with mode Ø222. If the program that creates the file terminates normally, it changes the mode to Ø666. Aborted runs can leave the files with mode Ø222 and the **uulog** program does not read or remove them. To remove them, use either **rm**(1) or **uuclean**, or change the mode to Ø666 and let **uulog** merge them with the logfile.

4.5. System Status Files

System Status Files (STST) are created in the spool directory by the **uucico** program. They contain information of failures such as login, dialup, or sequence check. They contain a TALKING status when two machines are conversing. The form of the file name is

STST.sys

where sys is the remote system name.

For ordinary failures, such as dialup and login, the file prevents repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it contains a talking status. In this case, the file must be removed before a conversation is attempted.

4.6. Lock Files

Lock files (LCK) are created in the spool directory for each device in use; e.g., the automatic calling unit and each

system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..str

where str is either a device or system name. The files can be left in the spool directory if runs abort. They are ignored (reused) after 24 hours. When runs abort and calls are desired before the time limit, the lock files must be removed.

4.7. Error Log

The ERRLOG file is created in the spool directory to record **uucp** system errors. Entries in this file should be rare. Wrong modes on files or directores, missing files, and read/write system call failures on the transmission channel may cause entries in ERRLOG.

4.8. Audit File

The AUDIT file is created in the remote system's spool directory whenever **uucico** is run with the -x option. This file contains the debug information from the remote **uucico** process.

4.9. Shell Files

The **uucp** program spools work and attempts to start the **uucico** program, but the starting of **uucico** sometimes fails due to communication lines being busy or the presence of lock files or status files. Therefore, the **uucico** program must occasionally be started. The command to start **uucico** can be put in a shell file with a command to merge log files and started by a **crontab** entry on an hourly basis. The file contains commands such as

```
program /uulog
program /uucico   -r1
```

The -r1 option is required to start the **uucico** program in MASTER mode.

Another shell file can be set up on a regular basis to remove TM ST, and LCK files, and C., X., or D. files for work that cannot be accomplished. Use a shell file containing commands such as

```
program /uuclean   -pTM -pC. -pD. -pX.
program /uuclean   -pST -pLCK -n12
```

The **-n12** option causes the ST and LCK files older than 12 hours to be deleted. If there is no **-n** option, a three-day limit is used.

A daily or weekly shell must also be created to remove or save old logfiles. Use a shell such as

```
cp spool /LOGFILE    spool /o.LOGFILE
rm spool /LOGFILE
```

4.10. Login Entry

One or more logins must be set up for **uucp**. Each of the /etc/passwd entries must have program/uucico as the shell to be executed. The login directory is not used, but if the system has a special directory for use as a sending or receiving file, it must be the login entry. The various logins are used in conjunction with the user file to restrict file access. Specifying the shell argument limits the login to the use of **uucp** (**uucico**) only.

4.11. File Modes

The owner and file modes of various programs and files are to be set as follows.

The programs **uucp**, **uux**, **uucico**, and **uuxqt** must be owned by **uucp** with the setuid bit set and execute only permissions (mode Ø4111). This prevents outsiders from modifying the programs to get at a standard shell from the **uucp** login.

The L.sys, SQFILE, and the USERFILE that are in the program directory must be owned by **uucp** and set with mode Ø4ØØ.

The L-cmds file in the program directory must be owned by **uucp** and set with mode Ø444.

Introduction to Display Editing with vi*

* This information is based on an article written by William Joy and revised by Mark Horton.

Table of Contents

List of Tables

SECTION 1
INTRODUCTION

1.1. General

Vi (Visual) is a display oriented interactive text editor in which the display acts as a window into the file being edited. Changes are reflected in the display, and this simplifies modifications. The regularity and the mnemonic assignment of commands makes the editor command set easy to remember and use. The full command set of the more traditional, line-oriented editor ex is available with vi, and it is easy to switch between the two editing modes.

Vi can be used on a wide variety of display terminals. New terminals are easily driven after editing a terminal description file. While it is advantageous to have an "intelligent" terminal that can insert and delete lines and characters from the local display, the editor functions well on "dumb" terminals with low-bandwidth telephone lines. The editor optimizes response time by using a smaller window and a different display updating algorithm. The command set of vi can be used as a one-line-window editor on hard-copy terminals and storage tubes.

This document was written on the assumption that the system being used is a Zilog System 8000, that the system console is a Lear Siegler ADM-31, and that the system software is Zilog ZEUS, a super-set of UNIX.

1.2. Command Notation

In this document, the following notation is used in command descriptions.

< > Angle brackets enclose descriptive names for the data or item to be entered. For example, <filename>.

[ ] Square brackets enclose optional data.

| Bar denotes an OR function.

ESC denotes the escape key (ALT on some keyboards)

RUB denotes the delete key (DEL on some keyboards)

CTRL denotes the CONTROL key. On certain terminals, CTRL is echoed as the circumflex symbol (^). Do not confuse the echo with the symbol used in this document for the up arrow (shown below).

↑ denotes an up arrow

1.3. Special Characters

ESC key: this key cancels partially entered commands. It also terminates text mode operations. If the terminal is already quiescent this key may also trigger a bell or audio annunciator.

RETURN key: this key initiates execution of most commands. It also initiates the csh (C Shell) commands.

RUB key: this key interrupts and stops the editor.

Interrupting the editor while it is redrawing or otherwise updating large portions of the display, might cause a confused display. If this occurs, it is still possible to continue editing by:

a. Entering the command

    CTRL-z

    redraws the display.

b. Ignoring the state of the display and either moving or searching again.

For the purposes of this document, the use of RUB is equivalent to an interrupt.

Slash (/): This symbol specifies a string for a search. When this key is pressed, the cursor moves to the bottom line of the display, where it acts as a prompt. To return the cursor to the current position, press RUB. Backspacing over the slash will also cancel the search.

The line kill and erase characters are user programmable. They can be changed with the program stty (refer to STTY(1) in the System 8000 ZEUS Reference Manual, 03-3255).

Line kill: the line kill character is usually the character

@

Character erase: the erase character is usually

CTRL-h

1.4. Invoking vi

When the system is up and running, set the terminal type, as shown:

%setenv TERM <code> (RETURN)

where: % is the system prompt
setenv is the command for setting the environment
TERM ie a required keyword
<code> is the terminal type code to be entered
For the Lear Siegler terminal, <code> is adm31
(RETURN) is the RETURN key

For other terminals, and additional information relevant to setting the terminal type, refer to Appendix A.

For the VTZ-2/10 terminal, <code> is vtz. The command format is

% vi [-t <tag>] [-r[<filename>]] [+[<command>]]
[+<n>] [+/<string>] [-l] [<filename>] (RETURN)

where: % is the system prompt.

vi is the command to invoke the visual display editor.

-t <tag> is the option to edit the file containing the tag <tag> at <tag> (see Section 5.2).

-r [<filename>] is the option used to recover a file after an editor or system crash (see Section 7.2).

+[<command>] executes the ex command <command> prior to entering visual mode. Without <command> the visual editor starts at the end of file (see also Section 5.2).

+<n> starts the visual editor at line number <n>.

+/<string> causes vi to search for and start at the string <string>.

-l is the option to set editing options for LISP (see Section 6).

<filename> is the name of the file to be edited.

NOTE

Do not include the square brackets ([]) and the angle brackets (<>) in the command.

Examples:

1. The simplest vi command is to invoke vi for editing a single file:

   %vi <filename> <RETURN>

where: <filename> is the name of the file to be edited.

NOTE

All entries must be terminated by a RETURN. For the remainder of this document, neither RETURN nor the system prompt is shown in the system commands; however, it is assumed that each command is terminated by RETURN.

2. To start vi at line number n, use the form

   vi [+<n>] <filename>

3. To start vi at some string <string>, use the form

   vi [+/<string>] <filename>

   vi searches for <string> and, if found, starts at <string>. For additional details, see Section 5.2. After the command is entered, the file name is echoed on the screen. The editor does not directly modify the file being edited. Rather, the editor copies the file in a buffer, and then remembers the file name. The contents of the file are not affected until the changes are written back to the original file. After a file has been copied, vi edits that file. The display clears, and the text of the file appears on the display. If it does not

1. Check for the correct terminal type code. An incorrect type code entry produces an unusable display. To check exit vi, enter

   :q

   Following the RETURN, control returns to the shell (command interpreter). To verify the correct terminal code, enter:

   printenv TERM

   and reenter it as described above.

2. Check for the correct filename. An incorrect filename can result in the display of an error diagnostic. If this occurs, return to the shell (as shown in 1. above) and restart.

3. If the editor does not respond, interrupt it with the delete key DEL (or RUB). Then, return to the shell (as shown in 1. above).

1.5. Operating Modes

Vi has four operating modes:

1. Command mode. This is the initial state and the normal operating mode. The other modes return to this mode. The escape key (ESC) cancels any partially entered command.

2. Text mode. This mode is entered by one of the following operators:

   a A i I o O c C s S R

   Any desired text can be entered in this mode. Text entry is normally terminated with the ESC. Text entry can also be terminated (abnormally) with RUB.

3. Last line mode. This mode is initiated by the following operators:

   : / ? !

   Commands or string searches are executed after a RETURN or ESC. Commands are canceled with DEL (or RUB). When the editor is in this mode, commands are echoed on the last line. If the cursor is in the first position of

the last line, the editor is performing a computation such as computing a new position in the file after a search, or running a command to reformat part of the buffer. While this is happening, it is possible to stop (interrupt) the editor with RUB. On some systems, when the cursor is on the bottom line, and the editor has been interrupted, the operator cannot type ahead.

4. Open mode. This is described in Section 1.9.

1.6. Escape to the Shell

To execute a shell command, while in vi, use a command of the form

:!<command>

where <command> is the shell command. The system runs the <command> and returns to vi when the command is completed. The operator is prompted

Hit RETURN to continue

After RETURN is entered, the editor clears and redraws the display; vi resumes control, and editing can continue. However, if another : command is entered prior to the RETURN, the display is not redrawn.

To execute more than one command in the shell, enter the command

:sh

When all necessary shell commands are completed, return to vi by entering

CTRL-d

Vi clears the display and editing can continue. Or, to execute more than one command in the C shell, enter the command

:csh

1.7. Leaving vi

To leave vi and return to the shell, use the command

ZZ

If changes have been made to the text, the contents of the vi buffer are written back into the original file, and the editor exits. If no changes have been made, the editor exits.

It is also possible to write the changes to the file without leaving vi by using the command

```
:w
```

To exit vi (quit) without writing the changes, use the command

```
:q!
```

This discards all text changes. This command is convenient when changes have been made to the contents of the buffer and the original file must remain unchanged. Do not use this command for changes that must be saved.

1.8. vi and ex

Vi is one mode of editing within the line-oriented editor ex. Some operations are easier in ex than in vi, such as systematic changes in line-oriented material. Experienced users often mix vi and ex commands to facilitate their work.

When vi is running, it is possible to escape to ex with the command

```
Q
```

Ex prompts with a colon (:). The vi commands prefaced with a colon (:) that are described in this document are available in ex. Similarly, most of the ex commands are available in vi when prefaced with a colon.

In rare instances, an internal error may occur in vi. In this case, a diagnostic is displayed, vi exits, and control returns to the command mode of ex. It is then possible to either:

1. Save the work in progress and quit by entering the command

   ```
   x
   ```

   or,

2. Re-enter vi with the command

    vi

1.9. Using vi on Hardcopy Terminals and Glass TTYs

It is possible to use vi on a hardcopy terminal, or a terminal with a cursor that cannot move from the bottom line. On these terminals, vi runs in "open" mode. In this mode, when a vi command is entered, the editor states that it is in open mode. This name comes from the open command in ex which invokes the open mode. With a "dumb" terminal, vi automatically enters open mode.

To invoke open mode manually, enter ex, and then, from ex, enter the command

    open

to return to ex from open mode, enter the command

    Q

To return to vi from ex, enter

    vi

The differences between visual and open mode are:

1. The way the text is displayed. In open mode, the editor uses a single-line window into the file. Moving backward and forward in the file displays new lines, which are always below the current line.

2. The command

    z

    takes no parameters, but draws a window of context around the current line and returns to the current line.

3. On a hardcopy terminal, the command

    CTRL-R

    retypes the current line. On these terminals, the editor usually uses two lines to represent the current line. The first line is a copy of the original line, and the second line is the work line; that is, it shows

any editorial changes. When characters are deleted, the editor displays a number of backslashes (\) to show what characters were deleted. The editor also reprints the current line soon after such changes so that they are visible.

1.10. Uppercase Terminals

The editor also reprints the current line soon after such changes so that they are visible. All characters are converted to lowercase characters. However, each upper case character must be preceded with a backslash. The combination "\character" does not echo until the backslash is followed by the second character.

The following characters are not available on uppercase-only-terminals:

{ } | ~ `

These characters can be entered as shown below:

For { use \(
For } use \)
For ~ use \^
For | use \!
For ` use \'

1.11. Slow Terminals

The vi editor minimizes the delay time required for display updates by limiting the output to the display. For slow and for "dumb" terminals, vi optimizes screen updates during text mode, and it replaces deleted lines with the symbol "@".

On slow terminals that can support vi in the full screen mode, it is useful to use "open" mode.

Vi has an operating option (slowopen) that is convenient when a slow terminal is being used. For additional information, see Section 6.2.

1.12. Abbreviations

Vi has a number of short commands that abbreviate longer commands that have been introduced above. These commands

are listed on the quick reference card.

1.13. Line Numbers

The vi editor, if desired, can number each line. Use the editor option, "number" (line number option) which is described in Section 6.8.

1.14. Line Representation in the Display

The vi editor folds long logical lines into shorter physical lines on the display. Commands that advance lines also advance logical lines. Hence they skip over all segments of a line in one motion. The command

```
|
```

moves the cursor to a specific column, and it can be useful for getting near the middle of a long line to split it. (This command is a vertical bar, not a numeral one or a lowercase l). For example, the command

```
8Ø|
```

places the cursor on the 8Øth column in a long sentence.

On a "dumb" terminal, the editor puts only full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty and places an @ on the line as a place indicator. When lines are deleted, the editor often just clears each text line and displays an "@" to save time, rather than rewriting the entire display. To maximize the information on the display enter:

```
CTRL-R
```

1.15. End of File Indicators

When the end of the file is displayed, and the last line is not at the bottom of the display, the vi editor displays the tilde (~) at the left end of each remaining line. This indicates that the 1st line of the file is shown in the display, and that those lines with the tilde are past the end of the file.

1.16. Counts

A count is an argument that affects the number of times the command is executed, or the number of lines affected. Several vi commands use a preceding count that affects the operation of the command. Some of the most common are the following:

1. For the following commands, a preceding count affects the amount of scroll:

   CTRL-d CTRL-u

2. For the following commands, the count affects the line or column number:

   z G |(vertical bar)

3. For most vi commands, a preceding count affects the number of times the command is repeated. For example, the command

   5RETURN

   advances 5 words. The command

   5dw

   deletes 5 words

   3.

   deletes 3 more words.

SECTION 2
DISPLAY CONTROL

2.1. Scroll Control

Use the following commands to scroll the display:

[<n>]CTRL-u to scroll up n lines.
[<n>]CTRL-d to scroll down n lines.

If n is omitted the default is half the window size.

NOTE

Certain "dumb" terminals cannot scroll up. In this case, CTRL-U clears the display and refreshes it with a line that is farther back in the file (towards the top).

2.2. Page Control

The functions CTRL-F and CTRL-B move the viewing window forward and backward one page, respectively. Both commands retain a few lines of text from the previous page for continuity. It is possible to read through a file using the page commands rather than the scroll commands. The primary difference is that the scroll commands move the text smoothly and leave more of the previous text, whereas the page commands change a page at a time, leaving only a few lines of text for continuity.

2.3. String Searches

The search function also positions the display within a file. This function searches the text file for a particular string of characters and positions the cursor at the next occurrence of the specified string. The search command is:

/<string>

To search backwards from the location of the cursor, use the command

?<string>

To repeat the forward or backward string search to the next occurrence of <string>, use the command

n

To repeat the string search in the reverse direction enter

N

If <string> is not present in the text file, vi prints the message "Pattern not found" on the last line of the screen, and returns the cursor to its original position. String searches normally wrap around the end of the file, and to find the string even if it is not in the direction originally specified in the command (provided the string is indeed in the file). The wraparound function can be disabled by the editor option "nowrapscan" (or nows). The nowrapscan option is one of the options described briefly in Section 6. Refer to the "Ex Reference Manual."

If the search is to match a string at the beginning of a line, then precede the search string with an up arrow (↑). To match only at the end of a line, end the search string with $.

Examples:

/↑ search

searches for the word "search" at the beginning of a line, and

/last$

searches for the word "last" at the end of a line.

If the search string contains a slash (/), it must be preceded by a backslash (\). This is also true if the editor option, "magic," is set (see Section 6).

At the end of the string search, vi places the cursor at the next or the previous occurrence of the string, as appropriate.

Whole lines of text can be affected up to the line prior to the line containing the string. To do so, use a search command with the form

/<string>/-<n>

where: <string> is part of the search command, and <n> is the number of lines preceding the line containing the string.

A "+" can be substituted for the "-". The result is that the search locates the string <n> lines after the line containing <string>. If no line offset is included, the editor affects characters up to the point of the string match, rather than whole lines. Thus, use "+Ø" to affect the line that matches.

The editor, if commanded, ignores the case of words in the string search. This is briefly described in the ignore case option in Section 6.

String searches can also be used in conjunction with the operators "d" and "c" (see Section 3.4), and "y" (see Section 4.3).

2.4. Cursor Position Control

To position the cursor at any particular line, where the lines are identified by number, use the command

[<n>]G

where n is a line number. Thus, 1G moves the cursor to the first line in the file. If <n> is omitted, the default is the last line of the file.

The cursor can be moved up, down, forward and back by the following keys:

up: k, CTRL-p, or CTRL-k
down: j, CTRL-n, or CTRL-j

back: h, CTRL-h, or backspace
forward: space bar, or l

Some terminals have arrow keys (four or five keys with arrows going in various directions) that have the same functions. (On the HP 2621 the function keys must be shifted.)

To advance the cursor to the first non-white position of the next line in the file, strike RETURN or "+" key. Similarly, strike "-" to move the cursor back to the first non-white position on the preceding line. These keys can also be used to scroll when the cursor is at the top or bottom of the display, as appropriate.

Vi also has commands to position the cursor at the top, middle, or the bottom of the display. For the top, strike the H key. Striking

<n>H

moves the cursor n lines down from the top of the display. The <n> is optional; the default position is the top of the display. Similarly, the command "M" positions the cursor in the middle of the display. The command

<n>L

positions the cursor either on the last line of the display, or the nth line from the bottom. If the <n> is omitted, the default is the bottom of the display.

The cursor can also be moved within a line with any of the following commands. To position the cursor on some word other than the first word, use the command

[<n>]w

which moves the cursor right to the beginning of the nth word on the line. The default is one word. The command

[<n>]b

moves the cursor back n words. The default is one word. The command

[<n>]e

advances the cursor right to the end of the nth word, rather than the beginning of the word. The default is one word.

The commands "b", "w" and "e" stop at punctuation marks. To move the cursor forward or backward without stopping at punctuation, use the characters "W", "B" or "E", respectively. The word keys wrap around the end of the line, and continue to the next line.

After the cursor has been moved for any reason, it can be returned to its previous position with the command `` (two back single quotation marks). The command '' (two forward single quotation marks) moves the cursor to the first non-white character of the line containing the previous position mark ('').

This is often more convenient than the command G because it requires no line count or other preparation.

To move the cursor to the first non-white position on the current line of text, use either "Ø" or the up arrow (↑). To move the cursor to the end of the current line, use "$."

The command

[<n>]f<c>

moves the cursor to the nth subsequent occurrence of the character <c>. The default is the next occurrence. Repeat by using the semicolon (;). The inverse command is

[<n>]F<c>

This performs the same function, but moves the cursor backward (into the preceding text). Repeat with a semicolon.

To move the cursor to the character preceding the nth occurrence of the character <c>, enter:

[<n>]t<c>

To move the cursor backwards to the character following the nth occurrence of the character <c>, enter:

[<n>]T<c>

The commands (f, F, t, and T) can be repeated with the semicolon, or the direction can be reversed with the comma.

To move the cursor to the matching parenthesis in a pair, place the cursor at either an opening or closing parenthesis and strike the percent (%) key. This feature also works for braces ({}) and square brackets ([]).

To advance the cursor to the beginning of the nth sentence following, use the command:

[<n>])

where the default for n is one. Similarly, to move the cursor back to the beginning of a sentence, use the command

[<n>]

where the default for n is one. A sentence is defined as ending with a period, a question mark or an exclamation point, followed either by two spaces or by an end of line. Sentences also begin at paragraph and section boundaries. For example, the command

2)

advances the cursor one sentence beyond the end of the current sentence.

To move the cursor forward to the beginning of the next paragraph, use the closing brace (}); similarly, to move the cursor back to the beginning of the preceding paragraph, use the opening brace ({). To move the cursor additional paragraphs, precede the brace with a count, n. For example, the command

3}

advances three paragraphs. A paragraph begins after an empty line or at a section boundary.

Finally, to move the cursor to the beginning of the next section, use a double closing square bracket:

]]

Use a double opening square bracket:

[[

to move the cursor back to the previous section boundary.

2.5. Tags

It is possible to mark a position in the editor file with a single letter tag, and then to return to any particular tag. To tag a position in text, use the command

m<tag>

where the tag is any letter of the alphabet.

To return to the tag, use the command

`<tag>

When using operators (such as the delete operator) with a tagged line, it may be convenient to operate on entire lines (for example, to delete entire lines), rather than to the exact position of the tag. In this case, use the form

'<tag>

rather than the form

```
`<tag>
```

For example, the command

```
d`<tag>
```

deletes entire lines from the position of the cursor to the line with the tag.

2.6. File Status

To find out the file status, enter the command

```
CTRL-g
```

The editor displays the name of the file being edited, the number of the current line, the number of lines in the buffer, and the relative position in the buffer as a percentage.

2.7. Clearing the Display

If, for any reason, the terminal display is garbled, it is often possible to obtain a correct display by using the command:

```
CTRL-l
```

or

```
CTRL-z
```

depending on the terminal. On a "dumb" terminal, when one or more lines have been deleted, it is possible to eliminate the "@" symbols with the command

```
CTRL-R
```

or

```
CTRL-r
```

This redraws the display and closes the deleted line(s).

2.8. Window Size

The window size is the number of lines written on the display. Vi maintains the current or default window size. On terminals that run at speeds greater than 1200 baud, the editor uses the full terminal display. On slower terminals (most dialup lines are in this group) the editor uses eight lines as the default window size. On terminals that run at 1200 baud, the default window size is 16 lines.

The appropriate window size is used when the editor clears and refills the display after a search or other motion that moves beyond the edge of the current window. Commands that take a new window size as count (see Section 1.16) often cause the display to be redrawn. With some of these commands, a smaller window size may be equally convenient, and it may be expedient to specify a smaller window size with the appropriate command. In any case, the number of lines displayed increases when:

1. Commands such as "-" are used; these move the window up.

2. Commands such as "+", RETURN or CTRL-d are used; these move the window down.

The scroll commands CTRL-d and CTRL-u "remember" the amount of scroll last specified. The default is half the window size.

The editor makes editing easier at low speeds by starting with a small window and expanding as the editing progresses. The editor can expand the window easily when inserts are placed in the middle of the display on intelligent terminals.

The window can be enlarged or reduced, and the current line, or any desired line, can be placed anywhere in the window with the command

<m>z<n><suffix>

where: <m> is the line number. The default is the current line z is the command operator
<n> is the number of lines in the window <suffix> controls the position of the desired line within the window, and is any of the following:

<RETURN> places the line at the top
. places the line at the center
- places the line at the bottom

For example, the command

```
z5.
```

redraws the display with the current line in the center of a five line window, while the command

```
5z5.
```

places line five in the center of a five line window.

SECTION 3
EDIT COMMANDS

3.1. General

In general, the edit commands use text mode. Text mode is initiated by entering of one of the various insert commands. Following the entry of the insert command, all subsequent keystrokes become text insertions. The text insert mode is always terminated by striking the (ESC) key.

Many related editor commands are invoked by the same alpha key and differ only in that one is given by a lowercase key, and the other is given by an uppercase key. The uppercase key usually differs from the lowercase key only in the sense of direction: the uppercase key operates backward and/or up and the lowercase key operates forward and/or down.

Using any of the text mode commands, it is possible to insert one letter, or many lines of text. To insert more than one line of text, strike the RETURN key in the middle of the input. A new line is then created for text and the insertion can continue. For slow or "dumb" terminals, the editor may wait to redraw the tail of the screen. In this case, the new text overwrites existing lines on the display. This avoids delays that occur if the editor attempts to keep the tail of the display up to date. The display is updated correctly when text mode is terminated.

Those characters normally used at the system command level for character or line deletion can also be used in text mode (e.g., CTRL-h or #; and @, CTRL-x or CTRL-u, as appropriate). CTRL-H always erases the last input character, regardless of the erase character.

Backspacing (while in text mode) does not erase characters. The cursor moves backwards, but the characters remain on the display. This is useful for entering similar text. The display is updated after the escape. To correct the display immediately, use the ESC, and reenter text mode.

It is not possible to backspace around the end of a line. To back up for a correction on a previous line, use ESC and then move the cursor back to the previous line. Make the correction, return and then reenter the appropriate text command.

NOTE

The character CTRL-W erases a whole word and leaves a space after the previous word. This is useful for backing up quickly for an insert.

It is not possible to erase characters with CTRL-W unless these characters were entered in text mode.

3.2. Insert Text

The general form of the text mode command is

<n><command><string>ESC

where: <n> is a preceding count; the default is one
<command> is one of the insert mode commands listed below
<string> is the inserted text string
ESC is the escape key

The effect of the preceding count is to repeat the inserted string n times. All one of the following command operators can be used to enter insert mode:

a A i I o O c C s S R

These commands and their variations are described below.

To insert text in the file, use one of the insert mode commands. For example,

i

Following the "i" (or other insert mode operator), all subsequent string of characters or text entered on the terminal are inserted in the file, until insert mode is terminated. To terminate insert mode, strike ESC (escape). On certain "dumb" terminals, when text is inserted, the display appears to overwrite the original text. When insert mode is terminated, all inserted and previous text is displayed properly.

A variation of the "i" command is

^i

which inserts text at the beginning of a line. The command

I

is equivalent.

In general, most of the insert commands can have a preceding count. For example, the command

5iapple

repeats the word "apple" five times:

appleappleappleappleapple

In the following description the preceding count is not always shown.

The command

a

also enters the vi text mode. The difference between the two commands is that with the command "i," text is inserted before the cursor (to the left), whereas with "a," text is inserted after the cursor (to the right). The command "a" is sometimes convenient for appending one or more letters to a word. The append operation is also terminated with the ESC key.

A variation of the command "a" uses the dollar sign,

$a

to move the cursor to the end of the current line and append text. An equivalent command is

A

Another way to add one or more lines of text to the file is to use the command

o

This opens the existing text and adds new text below the current line. Similarly, the command

O

opens and adds new text above the current line. Both commands are terminated with ESC. A preceding count opens n

lines.

It is also possible to insert non-printing characters in the text. Refer to Section 8.2.

3.3. Delete And Insert Characters

To delete a character or characters, place the cursor on the character to be deleted. Use the following command

[<n>]x

where: <n> is the number of characters and spaces to be deleted; the default is one.
x is the character delete command.

To delete a character or characters preceding the cursor, use the command

[<n>]X

where: <n> is the number of characters and spaces to be deleted; the default is one.
X is the character delete command.

To replace (change) one or more characters, use the command

[<n>]r<c>

where: <n> is the number of characters to be changed,
r is the replace command,
<c> is limited to one character which is repeated n times in place of n deleted characters.

To replace (change) one or more characters with a string, use the command

[<n>]R<string>

where: <n> is the number of times the replacement is performed,
R is the replace command,
<string> is the string used for replacement. The string can be any length.

To replace a number of characters with more than one character, use the command:

[<n>]s<string>

where: <n> is the number of characters to be replaced,
s is the substitute command,
<string> is the string that is substituted for the deleted characters. The string can be any length.

Use ESC to terminate string input.

3.4. Delete Operator

The command

d

acts as the delete operator.

To delete n words, position the cursor, and then enter

[<n>]dw

or

d[<n>]w

The default is one word.

To delete a word backwards (to the left of the cursor), enter

[<n>]db

or

d[<n>]b

The default is one word.

To delete n single characters, position the cursor on the appropriate starting character, and enter the command

[<n>]d<space>

This is equivalent to the x command. The default is one space.

A variation of the "d" command is

```
d$
```

which deletes the rest of the text on the current line. An equivalent command is

```
D
```

The operator "c" changes entire words. To change n words, enter the command

```
[<n>]cw
```

When the commmand is entered, the end of the text to be changed is marked with the symbol "$". Enter the replacement text, and terminate text entry with ESC. The default is one word.

A variation of the "c" command is

```
c$
```

which changes the rest of the text on the current line. An equivalent command is

```
C
```

When operating on a line of text, it is often desirable to delete the characters up to the first instance of a character. To do so, use the command

```
[<n>]df<x>
```

where f<x> locates the nth occurrence of the character <x> following the cursor. The default is the first occurrence of <x>. This command deletes the text up to--and including--the character <x>. A variant is the command

```
[<n>]dt<x>
```

where the operator f is replaced by the t. In this instance, the text is deleted up to--but not including--the character <x>. The command

```
T
```

is similar, but it operates in the reverse of the t

operator--that is, it operates in the preceding text.

To delete n entire lines, use the delete operator twice:

```
[<n>]dd
```

The default is one line.

On a "dumb" terminal, the editor may sometimes erase the entire line on the screen and replace it the symbol "@" at the far left. This does not correspond to any line in the file, but is a place indicator; it helps avoid a lengthy redraw of the display, which would be required in order to close up the deleted lines.

The operator

```
[<n>]cc
```

is similar to the command "dd", but it leaves vi in text mode, whereas dd does not. The command "cc" is convenient for changing an entire line. Position the cursor as appropriate, enter the command, and then enter the replacement text. Terminate text mode operation with ESC. The command

```
[<n>]S
```

is synonymous to the command "cc", and it is analogous to the command "s". Think of the "s" as a character substitute and the "S" as a line substitute.

There are several other variations on the line delete commands. The command

```
d<n>L
```

deletes all of the lines from the cursor down to the nth line from the botttom of the display. The default is all lines to the bottom of the display.

It is also possible to use a string search with the delete operator:

```
d/<string>
```

This command deletes characters from the cursor position to the point of the string match. Similarly, the command

```
d/<string>/-n
```

deletes characters from the cursor position to the nth line preceding the string match. The command

```
d/<string>/+n
```

deletes characters from the cursor to the nth line following the string match. Similar commands can be used to change entire lines in relation to a string:

```
c/<string>/-n
```

and

```
c/<string>/+n
```

In editing a document, it is usually easiest to edit in terms of sentences, paragraphs and sections. The operators "(" and ")" can be used with the delete operator. For example, the command

```
[<n>]d)
```

deletes the rest of n sentences. The default is from the cursor position to the end of the current sentence. Similarly,

```
[<n>]d
```

performs one of two deletions:

With the cursor at the beginning of a sentence, the command deletes the previous n sentences, or

When the cursor is not at the beginning of a sentence, the command deletes the text from the cursor back to the beginning of n sentences. The default is the beginning of the current sentence. The editor displays the extent of the change; it also indicates when a change will affect text that is not shown on the display.

To repeat the command more than once, use the period (.) key.

3.5. The Undo Operator

Vi has an undo operator

```
u
```

that reverses the last change made. The undo command can undo the preceding undo command--that is, the first undo command can return the text to its original state, and the second command can reinsert the change, but it can involve several lines. The undo command reverses only a single change. However, after having made more than one change to a line, the line can be restored to its original state with the command U.

Deleted text can be recovered even when the undo operator does not recover it. Recovering lost text is discussed in a separate section.

3.6. Program Editing Features

The editor has a number of commands for program editing. One of the most convenient is the autoindent option, which helps generate correctly indented programs. Another is the shiftwidth option, which is used to reset the backtab value. Both are discussed in Section 6.5.

The operators "<" and ">" are used to shift individual lines left or right, respectively, by one shiftwidth. To shift a line, use the double operators, as shown: [<n>]<< shifts the line to the left one shiftwidth, and [<n>]>> shifts the line to the right one shiftwidth.

Where n specifies a number of lines; the default is one line.

It is also possible to shift all lines from the cursor to the bottom of the display, either to the left or to the right, respectively. Use the command

```
<L
```

or

```
>L
```

respectively.

Another feature is useful for matching the opening and closing parenthesis in complicated expressions. To see the matching parenthesis, place the cursor at either an opening or a closing parenthesis and strike the percent key (%). This feature also works for braces ({}) and brackets ([]).

For editing programs in C, the double brackets ([[and]]) advance and retreat, respectively, to a line starting with a brace ({)--that is, one function declaration at a time. When the closing double brackets ( ]]) are used with an operator, it stops after a after a line that starts with a brace ({). This is sometimes useful with the command "y", as shown:

```
y]]
```

where the y operator yanks a line, and stores it in a buffer.

3.7. Erase and Line Kill Characters

The most common way to correct input text is to strike CTRL-H to delete an incorrect character, or to strike CTRL-W to delete incorrect words. If the normal system uses the crosshatch as the character erase (#), it works like CTRL-H in vi.

The line kill character is normally one of the following:

```
@
CTRL-X
CTRL-U
```

which erases all input on the current line. In general, the kill character does not erase back around an end of line, nor will it erase characters that were not inserted with the current text mode command. To make corrections on the previous line--after a new line has been started--use the following procedure:

1. Strike ESC to terminate input mode.

2. Move the cursor as appropriate to make the correction.

3. Return and continue in input mode. When continuing, the operator "A" is often convenient for appending the current line.

SECTION 4
REARRANGING AND DUPLICATING TEXT

4.1. General

By definition, a sentence ends with a period (.), an exclamation point (!), or a question mark (?); and is followed by either the end of a line, or two spaces. Any number of closing parens, brackets, or quotation marks may appear after the closing punctuation marks, but before the spaces or new line.

The operators (and) move the cursor to the beginning and the end of the previous and next sentences, respectively. Similarly, the operators { and }, and the operators [[and]] move over paragraphs and sections, respectively. The square bracket operators require a double operator entry because they can move the cursor an appreciable distance. While it is easy to return with the back quotation marks '' these commands could still be frustrating if they were easy to execute accidentally.

By definition, a paragraph begins after each empty line, and also at each of a set of paragraph macros. (Refer to the NROFF and TROFF documentation in the System 8000 ZEUS Reference Manual.) The paragraph macros can be changed or extended by assigning a different string to the the paragraphs option in EXINIT. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs. Sections in the editor begin after each macro in the sections option. Section boundaries are always line and paragraph boundaries.

It is possible to look through a large document by using the section commands. It is also possible to use a preceding count with each of the section and paragraph commands. The section commands interpret a preceding count as a different window size in which to redraw the screen display at the new location. This window size is the base size for newly drawn windows until another size is specified. This is useful when looking for a particular section on a slow terminal. It is possible to give the first section command a small count, and then see each successive section heading in a small window.

4.2. Buffers

Vi has the following buffers:

1. A single, unnamed buffer, where the last delete or changed text is saved.

2. A set of named buffers--a through z--that can be used to save or move text, either within a file, or between files.

The buffers are used by the "yank" and "put" operators described in section 4.3.

4.3. Text Manipulation

The operator (for "yank") is used to place text into the unnamed buffer, or any of the named buffers. The command syntax is

"[<buffer>][<n>]yw

where: " indicates that the following character is a buffer, and not a command
<buffer> is a buffername a through z; default is the unnamed buffer
<n> is the number of words to yank; default is one word
y is the yank operator
w is the word operator

This command does not delete the yanked text. Punctuation marks are counted as words. To yank a complete word, the cursor must be on the first letter of the word. If the cursor is not on the beginning of the word then all characters from the cursor position to next white space (at the end of the word) are yanked.

The operator "yy" is eqivalent to "Y"; the command

"[<buffer>][<n>]Y

yanks the entire line on which the cursor rests, and places it in a buffer, as described above. The count <n> preceding the Y operator yanks n lines of text. The default is one line.

Examples:

The command

```
yw
```

yanks the word on which the cursor is located. The command

```
4yw
```

yanks the word on which the cursor is located, and the following three words into the unnamed buffer. The command

```
"a12yw
```

yanks 12 words into buffer a.

An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put command (p or P, described below) can move it elsewhere. However, the unnamed buffer contents are lost when files are changed; therefore, to change text from one file to another, be sure to use a named buffer.

Text that has been yanked can be reinserted (put) in the text with the operators p or P, where the command syntax is

```
"[<buffer>]p
```

where quotation marks and <buffer> indicate the buffername, where the yanked text was stored. The operator "p" reinserts the yanked text after or below the cursor, and the operator "P" reinserts the text before or above the cursor. Command syntax is identical for both P and p operators. If a buffer is not specified, the default is the unnamed buffer.

The text being yanked can be part of a line, or an object such as a sentence that spans more than one line. In this case, when the text is replaced, it is replaced after (or before) the cursor, depending on the command. If the text forms whole lines, then it is returned in whole lines, without changing the current line.

The command

```
[<n>]YP
```

yanks a copy of n lines, and then reinserts the same text immediately prior to the current line. The result is that there are two identical text lines and the cursor moves to the top line. The command

[<n>]Yp

is similar, but it copies n lines and places them after (below) the current line, so that there are two identical lines. For example, the command 3YP repeats the line of text three times. The default is one line of text.

The yank command, like the delete and change commands, can be used with a string search. The command

y/<string>/-<n>

yanks the characters from the cursor position to the nth line preceding the string match. Similarly, the command

y/<string>/+<n>

yanks characters from the cursor to the nth line following the string.

The same buffers can be used with the delete operators to move blocks of text within the file or to another file. Moving a block of text requires three operations:

1. Delete and store n lines.

2. Move cursor to the new location.

3. "Put" the text.

Example:

Delete five lines of text and temporarily store them in buffer a:

"a5dd

The quotation marks indicate a buffername, not the "a" command. Next, move the cursor to the new text location, and enter the command

"ap

or the command

"aP

to insert the text in the new location.

To switch to another file for editing before restoring the yanked text use a command of the form

:e <filename>

where <filename> is the other file to be edited. (These commands are described in a later section.)

NOTE

If the contents of the current editor buffer have been changed, they must be either written back or discarded prior to switching to the other file.

SECTION 5
FILE MANIPULATION

5.1. Writing, Quitting and Editing New Files

The basic write and quit commands are described in Section 1.7.

If the text has been changed, but the changes are not to be written to the file, the quit command (:q!) discards the changes. To re-edit the same file (starting over) enter the command:

```
:e!
```

This command is seldom used, because the changes cannot be made after they have been discarded.

To edit a different file without leaving the vi editor enter

```
:e <filename>
```

If the changes have not been written to the file (prior to this command), vi displays the message

```
No write since last change (:edit! overrides)
```

and delays editing the other file. Respond by entering the command

```
:w
```

to save the changes in the first file. After the changes are written, repeat the command ":e <filename>" or use the command

```
:e!
```

to discard the changes in the first file and call the second file. To save changes automatically set the autowrite option. When autowrite is set, use the command

```
:n
```

rather than

```
:e
```

5.2. File Manipulation Commands

Table 5-1 contains the vi file manipulation commands. These commands are followed by a carriage return (RETURN) or an escape (ESC). Most of the commands are self explanatory; however, the following describes how to use these commands.

Table 5-1 File Manipulation Commands

COMMAND	FUNCTION
:w	Write changes back to file
:wq	Write changes back and quit
:x	Write, if necessary, and quit
:e<name>	Edit file <name>
:e!	Discard changes and re-edit
:e+<name>	Edit file <name>, starting at end
:e+<n><name>	Edit file <name> starting at line n or with command n
:e#	Edit alternate file, which is designated by the last filename typed before the current filename
:e%	Edit current file
:w <name>	Write file <name>
:w! <name>	Overwrite file <name>
:<x>,<y>w <name>	Write lines <x> through <y> to <name>
:r <name>	Read file <name> into buffer
:r!<cmd>	Read output of <cmd> into buffer
:n	Edit next file in argument list
:n!	Discard changes to current file, and edit next file
:n <arglist>	Specify new list of arguments <arglist>
:ta <tag>	Edit the file containing the tag <tag>, at <tag

The basic write command is

```
:w
```

which writes changes to the file. When editing is completed

for a single file, write the changes back and terminate vi with the command

ZZ

For editing long text, it is convenient to write back the changes more frequently with the command ":w" and terminate with the command "ZZ".

When editing more than one file, write back the changes with the command ":w" and start editing a new file with an ":e" command. Another way is to set the autowrite option (see Section 6) and use the command

:n <file>

to fetch the next file for editing. This command is inoperative unless the changes to the current file have been written back.

Whenever changes have been made to the editor's copy of a file, but they are not to be written back, then the exclamation point (!) is added to the command being used. The result is that the editor discards any changes that have been made. For best results, use this command carefully.

The various ":e" commands can be given arguments. The argument "+" starts editing at the end of the file, and the argument

+<n>

starts the editor at line n. Moreover, n can also be any editor command not containing a space, such as a scan like

+/<string>

or

+?<string>

where the editor searches for <string>.

Other arguments for ":e" include the character "%", which, when used in the command, is interpreted as the current file name. Another argument is "#", which is interpreted as an alternate filename, where the alternate filename is the last filename typed other than the current filename. For example, suppose the command

:e

has been entered, and a diagnostic is returned indicating that the file has not been written. One possibility is to enter the command

:w

which writes the file, and then the command

:e#

to redo the previous ":e". The command

CTRL-↑

performs the same function.

To write a part of a buffer to a file, first determine the line numbers that bound the portion to be written. Use the command

CTRL-g

to display the line number where the cursor is located or set the option number. Then enter the command

:<x>,<y>w <name>

where: <x>,<y> specify the top and bottom line numbers
<name> is the file name of the destination file.

If the destination file does not exist, it will be created; otherwise vi prints the diagnostic message

"<name>" File exists - use "w! <name>" to overwrite

command. Then, instead of line numbers, use the address marks in the command. For example, the command

ma

marks the first line in register a, and

mb

marks the last line in register b. The command

w!

writes these lines to the file <name>.

It is possible to read another file into the buffer after the current line. Use the command

:r <name>

To edit a set of files in succession, first enter all of the filenames as arguments in the command

:n <name1> <name2> .... <namex>

then edit each one, in turn, using the command

:n

It is also possible to use the command ":n" and specify a pattern to be expanded, such as with an asterisk (*) or a set of characters to match. This can also be done with the initial vi command.

The command

:ta

is very useful for editing large programs. It uses a data base of function names and their locations (which can be created by the program ctags(1). See the System 8000 ZEUS Reference Manual) for finding a function with a name. If the ":ta" command requires the editor to switch files, any current work must be written to a file or abandoned prior to switching files. To relocate a tag, repeat this command without any arguements.

To read in the output from a shell command, use an exclamation point with a shell command <cmd>, as shown:

:!<cmd>

SECTION 6
OPTIONS

6.1. General

As noted previously, the options in the editor ex are also available and easy to use with Vi. The most useful ones are listed in Table 6-1 below.

Table 6-1. Frequently Used Options

Option	Default	Function
autoindent	noai	Automatic indentation
autowrite	noaw	Automatic write before :n, :ta, CTRL-↑, and !
ignorecase	noic	Ignore case in searching
lisp	nolisp	Commands deal with S-expressions
list	nolist	Tabs print as CTRL-I; end of lines are marked with $
magic	magic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para= IPLPPPQPP LIbp	Macro names that start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
scroll	1/2	Number of lines scrolled
sections	sect=NHSHH HU	Macro names that start new sections
shiftwidth	sw=8	Shift distance for <, > and input CTRL-d and CTRL-t
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	noslow	Postpone display updates during inserts
term	vtz	The type of terminal being used
terse	noterse	Shorter error diagnostics
window	speed dependent	Number of lines in display window
wrapmargin	wm=Ø	Bring right margin in from the right
wrapscan	ws	Wrapping around end-of-file

In general, there are three kinds of options: numeric options, string options, and toggle options. Numeric and string options are set by commands of the form:

set <opname>=<val>

where: <opname> is the name of the option
<val> is the appropriate string or numeric value for the option. Toggle options can be set or reset, respectively, with the following commands:

set <opname>
set no<opname>

These options can be entered while in vi by preceding the set command with a colon, and the command can be abbreviated as shown:

:se <opname>=<value>

or

:se <opname>

To display a list of those options that have been set, enter the set command without any option name, as shown:

:set

To display the value of a single option enter the command:

:set <opname>?

Similarly, to display a list of all possible options and their current values, enter the command

:set all

Note that the above commands can also be abbreviated, and that multiple options can be placed set using only one option command:

:se ai as nu

The options that are set during an editing session last only until the editor is exited. However, it may be convenient to have a list of options that are set whenever the editor is used. This can be accomplished by creating a list of ex commands--that is, commands used by the text editor ex--that are to be run every time the programs ex, edit, or vi are invoked. (Note that all commands that start with a

colon are ex commands.) It is good practice to list these commands on a single line.

It is possible to put any number of the option commands in the environment variable EXINIT. When options are set in the environment, then they are automatically set at each entry to vi. For example, to set autoindent, autowrite and terse, the command would be (using csh):

```
setenv EXINIT 'set ai aw terse'
```

6.2. Editing on Slow Terminals

The slow terminal text mode is controlled by the slowopen option. This option is set by the command

```
:se slow
```

On slow systems this option limits the output to the terminal. It is also possible to force the editor to use this option even on faster terminal by using this option. To disable the slowopen option, use the command

```
:se noslow
```

It is also possible to simulate an intelligent terminal with the redraw option. This simulation generates a great deal of output, and is generally tolerable only on lightly loaded systems and fast terminals. This option is set with the command

```
:se redraw
```

and it is cancelled with the command

```
:se noredraw
```

6.3. Ignore Case

The editor will, if commanded, ignore the case of words in the string search. The appropriate command is:

```
:se ic
```

To turn off the ignore case option, use the command

```
:se noic
```

6.4. Magic Characters

Strings used in a string search can contain characters that have "magic" meanings to vi. If this capability is not desired, then reset the magic option with the command

```
:se nomagic
```

With nomagic, only the characters "^" and "$" are special in patterns. The character "\" is also special (as it is almost everywhere in the system), and may be used for an extended pattern matching capability.

With either magic or nomagic, it is necessary to use a "\" (backslash) before a "/" in a forward string search or a "?" in a backward string search. That is, if the string search is for either a "/" (forward) or a "?" (backward), then the character must be preceded by a backslash. Table 6-2 lists the extended forms that are used when the magic option is set.

Table 6-2. Magic Option Extended Operators

Operator	Function
↑	At the beginning of a pattern, matches the beginning of a line
$	At the end of a pattern, matches the end of a line
.	Matches any character
<	Matches the beginning of a word
>	Matches the end of a word
[str]	Matches any single character in the string str
[^str]	Matches any single character not in the string str
[x-y]	Matches any character between x and y, where x and y are alphanumeric characters
*	Matches any number of the preceding pattern

Note that in the nomagic mode the primitives

```
.   [   and *
```

are used with a preceding "\".

6.5. Autoindent and Shiftwidth

The autoindent option is convenient for generating correctly indented programs. To set the autoindent option, use the command

```
:se ai
```

To demonstrate the operation of the option, open a new line with the letter "o", enter a few tabs, type some characters, and then start another line. The editor supplies white space at the start of the new line, so that it is lined up with the previous line of text. Note that it is not possible to backspace over the automatic indentation.

When the autoindent option is being used, it is sometimes convenient to return to the margin--for example, to place a label at the margin. To defeat the autoindent, use the command

```
CTRL-d
```

which then backspaces over the automatic indent. Each time this command is entered, the cursor backs up one shiftwidth. If the shiftwidth is set to eight, the cursor backs up eight columns. Note that this only works immediately after the supplied autoindent.

To stop all indent, including the next line, strike:

```
ØCTRL-d
```

An easy way to place a label at the left margin is to strike the up-arrow (^) and then CTRL-D. The editor moves the cursor to the left margin for one line, and then restores the indent on the next line.

There is normally an eight column left boundary. To reset this boundary, use the shiftwidth option, which is entered by the command

```
:se sw=<n>
```

where <n> is the number of columns that sets the width of the boundary.

6.6. Continuous Text Input

When large amounts of text are being entered, it is often convenient to have lines broken near the right margin

automatically. To have the text broken n columns from the right margin, use the command

```
:se wm=<n>
```

If the editor breaks an input line, it can be rejoined with the command

```
[<n>]J
```

where n is the number of lines to be joined. The default is to move the following line to the end of the current line. The editor supplies white space, as appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. To delete the white space use the command "x".

6.7. LISP Editing Options and Commands

The vi editor has some convenient options for editing programs in LISP. The first is the lisp option which is set with the command

```
:se lisp
```

This option changes the parenthesis commands "(" and ")" so that they move backward and forward over s-expressions. The braces-- "{" and "}"--are like the parenthesis commands, but they do not stop at atoms. These commands can be used to skip quickly through a comment, or to the next list.

The autoindent option works differently for LISP. It supplies indent to align at the first argument to the last open list. If there is no such argument, then the indent is two spaces more than the last level.

The showmatch option is convenient for typing in LISP. Providing that the opening parenthesis is showing on the display, if a closing parenthesis is typed, the cursor then briefly moves to the position of the opening parenthesis. To set this option, use the command

```
:se sm
```

The vi editor also uses the operator

```
=
```

which realigns existing lines as though they had been typed with the lisp and the autoindent options set. For example,

the command

```
=%
```

at the beginning of a function realigns all the lines of the function declaration.

Finally, when editing LISP, the double brackets "[[" and "]]" cause the cursor to advance or retreat, respectively, to lines beginning with an opening parenthesis. This is useful for dealing with entire function definitions.

6.8. Line Numbers

If desired, the editor can place line numbers before each line of text on the display. Use the command

```
:se nu
```

To disable the line number option, use the command

```
:se nonu
```

6.9. Tabs and End of Line Indicators

It is possible to have the display represent tabs as CTRL-I and represent the ends of lines with the symbol "$" by using the list option. Give the command

```
:se list
```

This option can be disabled with the command

```
:se nolist
```

6.10. Automatic Writing of Files

When a file has not been written out prior to changing to a new file, vi prints the diagnostic

```
"No write since last change (edit! overrides)".
```

To have the editor automatically save changes, set the "autowrite" option

```
:se aw
```

To change files, use the command

```
:n
```

instead of

```
:e
```

To disable this option use the command

```
:se noaw
```

6.11. Defining Paragraphs and Sections

There are editor options available to define a paragraph and/or section for NROFF macros (see Section 7 of the System 8000 ZEUS Reference's Manual). A paragraph normally begins after each empty line; these paragraph boundaries are used by the operators "{" and "}" (see Section 2.4). By setting the "paragraph" option

```
set para=<macro name>
```

where <macro name> is an nroff macros(s) that defines the start of a paragraph. Similarly, sections can be redefined by using

```
set sections=<macro name>
```

By definition, a section begins after each line with a formfeed CTRL-L in the first column; section boundaries are also line and paragraph boundaries. These boundaries are used by the operators "[[" and "]]" (see Section 2.4).

6.12. Terminal Type

The terminal type is determined from the environment when

```
% setenv TERM <type>
```

was executed (see Section 1.4). This option

```
:se term
```

simply outputs the terminal type.

6.13. Scroll

The amount of scroll when using the CTRL-d, CTRL-u and "z" commands can be altered by issuing

```
:se scroll=<val>
```

where: <val> is the amount of scroll (number of lines).

6.14. Terse

The error diagnostics can be shortened with the command

```
:se terse
```

and lengthened again with

```
:se noterse
```

This is desirable for the more experienced user.

6.15. Window

The number of lines in a text window can be altered with this command

```
:se window=<val>
```

For slow terminals (600 baud or less), the window size is 8; for medium terminals (1200 baud), the size is 16; and for high speed terminals, the full screen size minus 1 is assigned.

6.16. Wrapping Around the End of Files

String searches normally proceed through a file and then continue to search at the beginning. This capacity can be disabled with

```
:se nows
```

APPENDIX A
SPECIFYING TERMINAL TYPE

Before calling vi, the correct terminal type must be entered. The following is an incomplete list of terminals and terminal type numbers that can be entered in vi, as appropriate. Unless indicated by an asterisk (*), the terminals listed here are all intelligent.

Terminal	Code
VTZ-2/10	vtz
C. Itoh 101	cit *
Hewlett-Packard 2621A/P	2621
Hewlett-Packard 264x	2645
Microterm ACT-IV	act4 *
Microterm ACT-V	act5 *
Lear Siegler ADM-3a	adm3a *
Lear Siegler ADM-31	adm31
Human Design Concept 100	c100
Datamedia 1520	dm1520 *
Datamedia 2500	dm2500
Datamedia 3025	dm3025
Perkin-Elmer Fox	fox *
Hazeltine 1500	h1500
Heathkit h19	h19
Infoton 100	i100
Teleray 1061	t1061
Dec VT-52	vt52 *

To enter the type of terminal, use the command

```
setenv TERM <code>
```

where <code> is the terminal type code listed above.

APPENDIX C
vi SYMBOL DICTIONARY

This appendix gives the uses the editor makes of each character. The characters are presented in their order in the ASCII character set: control characters come first, then most special characters, then the digits, upper and then lowercase characters.

The information for each character includes the meaning it has as a command, and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed.

CTRL-@ Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation.

CTRL-A Unused.

CTRL-B Backward window. A count specifies repetition. Two lines of continuity are kept if possible.

CTRL-C Unused.

CTRL-D As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future CTRL-D and CTRL-U commands. During an insert, backtabs over autoindent white space at the beginning of a line; this white space cannot be backspaced over.

CTRL-E Unused.

CTRL-F Forward window. A count specifies repetition. Two lines of continuity are kept if possible.

CTRL-G Equivalent to :fCR, printing the current file name, whether it has been modified, the current line number and the number of lines

in the file, and the percentage of the way through the file that you are.

CTRL-H (BS) Same as **left arrow.** (See **h.**) During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different.

CTRL-I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the tabstop option.

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

CTRL-K Same as **up arrow** (see **k**).

CTRL-L Same as right arrow. The ASCII formfeed character, this causes the screen to be cleared and redrawn on dumb terminals. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.

CTRL-M (RETURN)
A carriage RETURN advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines. During an insert, a RETURN causes the insert to continue onto another line.

CTRL-N Same as **down arrow** (see **j**).

CTRL-O Unused.

CTRL-P Same as up arrow (see k).

CTRL-Q Not a command character. In input mode, CTRL-Q quotes the next character, the same as ^V, except that some teletype drivers eat the CTRL-Q so that the editor never sees it.

CTRL-R Same as replacement operator (see r). On hardcopy terminals in <u>open</u> mode, retypes the current line.

CTRL-S — Unused. Some teletype drivers use CTRL-S to suspend output until CTRL-Q is typed.

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

CTRL-U — Scrolls the screen up, inverting CTRL-D, which scrolls down. Counts work as they do for CTRL-D, and the previous scroll amount is common to both. On a dumb terminal, CTRL-U will often necessitate clearing and redrawing the screen further back in the file.

CTRL-V — Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.

CTRL-W — Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see CTRL-h).

CTRL-X — Unused.

CTRL-Y — Unused.

CTRL-Z — Redraws the screen.

CTRL-[(ESC) — Cancels a partially formed command, such as a z when no following character has yet been given; terminates inputs on the last line (read by commands such as : / and ?); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. Thus, ESC can be used to stop any function and reenter command mode. The flash or ring indicates that all functions have been stopped, and vi has returned to command mode. Prior to entering insert mode, if there is any doubt about what mode is currently in effect, then press ESC, followed by an insert mode command, such as a. the result is that vi enters insert mode, regardless of the previous mode.

CTRL-\ — Goes to ex.

CTRL-] Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a RETURN. Mnemonically, this command is "go right to"

CTRL-↑ Equivalent to :e #. Display returns to the previous position in the last edited file. To edit a file that was specified by this command, and the system response was the diagnostic "No write since last change", enter the command :w. This allows CTRL-↑ to operate. To change files without writing the current underscore file, use the command :e! # instead.

CTRL-_ Unused. Reserved as the command character for the Tektronix 4Ø25 and 4Ø27 terminal.

SPACE Same as right arrow (see 1).

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

In input mode, if this is the erase character, it deletes the last character typed in input mode. It must be preceded with a \ to insert it, since it normally backs over the last preceding input character.

$ Moves the cursor to the end of the current line. With a count <n>, the cursor advances to the nth end of line following. For example, 2$ advances the cursor to the end of the following line. With the list option, the end of each line is indicated by a $.

% Moves to the parentheses or brace {} which balances the parentheses or brace at the current cursor position.

& Same as :& RETURN; repeats a previous substitution.

" Precedes a named buffer specification. There are named buffers 1-9 that save deleted text, and named buffers a-z that store "yanked" text.

The ` can be used the following ways: (a) When followed by another ', the cursor returns to its previous position, but at the beginning of the line. The previous position is set whenever the cursor is moved from the current line. (b) When the ' is followed by a letter a-z, the cursor returns to the line that was marked with this letter (by the m command), at the first non-white character in the line. (c) When ' is used with a second ' and an operator such as d, the operation takes place over complete lines. Example: d'' deletes the lines between the appropriate marks. Similarly, when used with a `, the operation takes place from the exact marked place to the current cursor position within the line. Retreats to the beginning of a previous sentence, or to the beginning of a LISP s-expression if the lisp option is set. Any number of closing)] " and ' characters may appear after the . ! or ?, and before the spaces or end of line. A count <n> advances n sentences.

) Advances to the beginning of the next sentence. A count repeats the effect. See (above for the definition of a sentence.

* Unused.

+ Same as RETURN when used as a command.

, Reverse of the last f F t or T command, looking the other way in the current line. Especially useful after hitting too many ; characters. A count repeats the search.

- Retreats to the previous line at the first non-white character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be

required the screen is also cleared and redrawn, with the current line at the center.

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; use "." to delete more and more words or lines. A count is passed on to the command being repeated. Thus, after 2dw, 3. deletes three words.

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input of the bottom line; an (ESC) returns to command state without searching. The search begins with the RETURN which terminates the pattern. The cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line. The cursor returns to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, whole lines are affected. To do this, give a pattern with a closing / and then an offset +n or -n.

To include the character / in the search string, escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless **nomagic** is set in the .exrc file, the characters ., [, *, and ~ in the search pattern must be preceded with a \ to get them to work as expected.

Ø Moves to the first character of the current line. Also used to form numbers after an initial 1-9.

1-9 Used to form numeric arguments to commands

: A prefix for the commands for file and option manipulation, and for escapes to the system. Input is given on the bottom line and terminated with a RETURN; and the command then executed. If the colon (:) is hit accidentally, return by hitting DEL (or RUB).

< Shifts lines left one shiftwidth (normally 8 spaces). Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus 3<< shifts three lines.

= Reindents line for LISP, as though they were typed in with lisp and autoindent set.

> Shifts lines right one shiftwidth (normally 8 spaces). Affects lines when repeated as in >>. Counts repeat the basic object.

? Scans backwards; the opposite of /. For details see the / description above.

@ If this is the kill character, escape it with a \ to type it in during input mode, as it normally backs over input on the current line.

A Appends at the end of line, a synonym for $a

B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.

C Changes the rest of the text on the current line; a synonym for c$.

D Deletes the rest of the text on the current line; a synonym for d$.

E Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.

F Finds a single following character, backwards in the current line. A count repeats this search that many times.

G Goes to the line number given as preceding argument, or the end of the file if no

preceding count is given. The screen is redrawn with the new current line in the center if necessary.

H **Home arrow.** Homes the cursor to the top line of the screen. If a count <n> is given, then the cursor moves to the nth line of the screen. In any case, the cursor moves to the first non-white character on the line. If used as the target of an operator, full lines are affected.

I Inserts at the beginning of a line; a synonym for ↑ i.

J Joins together lines, supplying appropriate white space; one space between words, two spaces after a ., and no spaces at all if the first character of the joined line is). A count causes that many lines to be joined rather than the default two.

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a count <n> to the first non-white character on nth line from the bottom. Operators affect whole lines when used with L.

M Moves the cursor to the middle line on the screen, at the first non-white character on the line.

N Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.

O Opens a new line above the current line and inputs text there. Terminate with (ESC). A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the slowopen option works better.

P Puts the last deleted text back before/after the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text is inserted between the characters before and at the cursor. May be preceded by a named buffer

specification "x to retrieve the contents of the buffer; buffers 1-9 contain deleted material, buffers a-z are available for general use.

Q — Quits from vi to ex command mode. In this mode, whole lines form commands, ending with a RETURN. For all commands; the editor ex prompts with the colon.

R — Replaces characters on the screen with characters you type (overlay fashion). Terminate with (ESC).

S — Changes whole lines, a synonym for cc. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.

T — Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count <n> repeats the effect n times. Most useful with operators such as **d**.

U — Restores the current line to its state before you started changing it.

V — Unused.

W — Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count <n> repeats the effect n times.

X — Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y — Yanks a copy of the current line into the unnamed buffer, to be put back by a later p or P; a very useful synonym for yy. Count <n> yanks n lines. May be preceded by a buffer name to put lines in that buffer.

ZZ — Exits the editor (Same as :xRETURN). If any changes have been made, the buffer is written out to the current file. Then the editor quits.

[[Backs up to the previous section boundary. A section begins at each macro in the sections option, normally a ".NH" or ".SH" and also at lines which start with a formfeed ^L. Lines beginning with { also stop [[; this makes it useful for looking backwards, a function at a time, in C programs. If the option lisp is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects.

\ Unused

]] Forward to a section boundary; see [[for a definition.

↑ Moves to the first non-white position on the current line.

_ Unused.

` When the ` is followed by another ` the cursor returns to the previous context. The previous context is set when the cursor is moved from the line. When followed by a letter a-z, returns to the position which was marked with this letter by the **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line. When ' is used, the operation takes place over complete lines. See forward quote (').

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with (ESC).

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count <n> repeats the effect n times.

c An operator that changes the following object and replaces it with following input text. The c command must take an object, such as

the operator w. Terminated by (ESC). If more than one line is affected, then the previous text is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed is marked by $. A count <n> affects n objects. For example, both 3c) and c3) change the following three sentences.

d An operator d deletes the following object; an object is an operator like w. If more than part of a line is affected, the text is saved in the numeric buffers. A count <n> affects n objects. Thus **3dw** is the sames as **d3w.**

e Advances to the end of the next word, defined as for **b** and **w.** A count <n> repeats the effect n times.

f Finds the first instance of the next character following the cursor on the current line. A count <n> repeats n times.

g Unused.

Arrow keys **h, j, k, l,** and **H.**

h **Left arrow.** Moves the cursor one character to the left. **h** and CTRL-H have the same effect. On terminals that send escape sequences (such as **vt52, cl00,** or **hp**), the arrow keys cannot be used. A count repeats the effect.

i Inserts text before the cursor; otherwise like **a.**

j **Down arrow.** Moves the cursor one line down in the same column. If the position does not exist, vi comes as close as possible to the same column. synonyms include **^J** (linefeed) and **^N.**

k **Up arrow.** Moves the cursor one line up. **^P** is a synonym.

l Right arrow. Moves the cursor one character to the right. SPACE is a synonym.

m Marks the current position of the cursor in the mark register, which is specified by the next character **a-z**. Return to this position or use with an operator using ` or '.

n Repeats the last string search command.

o Opens new lines below the current line; otherwise like **O**.

p Puts texts after/below the cursor; otherwise like **P**.

q Unused.

r Replaces the single character at the cursor with another single character. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R**, which is usually more useful.

s Changes the single character under the cursor to the text that is inserted. Terminate with (ESC). With a count, the count characters on the current line are changed. The last character to be changed is marked with $ (as in **c**).

t Advances the cursor up to the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. Use . to delete more.

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers.

v Unused.

w Advances to the beginning of the next word, as defined by **b**.

x Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but

only on the current line.

y An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "x, the text is placed in that buffer also. Text can be recovered by a later **p** or **P**.

z Redraws the screen with the current line placed as specified by the following character; RETURN specifies the top of the screen, . the center of the screen, and - at the bottom of the screen. A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line.

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the paragraphs option, normally, `.ip', `.lp,' `.pp,' `.QP,' and `.bp.' A paragraph also begins after a completely empty line, and each section boundary (see [[above).

| Places the cursor on the character in the column specified by the count.

} Advances to the beginning of the next paragraph.

~ Unused.

CTRL-? (DEL) Interrupts the editor, returning it to command accepting state.

APPENDIX D
vi QUICK REFERENCE

Interrupting and Canceling

File manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit; discard changes
:e name	edit file name
:e!	reedit, discard changes
:e #	edit alternate file (also CTRL-^)
:w name	write file name
:w!	overwrite file name
:!cmd	run cmd, then return
:n	edit next file in arglist
:f	show current file and line number (also CTRL-g)
:sh	escape to shell (CTRL-d for return)

Cursor Positioning within File

CTRL-f	forward screenful
CTRL-b	backward screenful
CTRL-d	scroll down half screen
CTRL-u	scroll up half screen
G	goto line (end default)
/<string>	next line matching <string>
?<string>	previous line matching <string>
n	repeat last / or ?
N	reverse last / or ?
/<string>/+n	n`th line after <string>
?<string>?-n	n`th line before <string>
]]	next section/function
[[	previous section/function
%	find matching parenthesis or brace

Marking and Returning

``	return to previous position in text
''	cursor moves to first non-white character on the line at the previous position
mx	mark position with letter x
`x	to mark x at position within line
'x	to mark x at first non-white character in line

Line Positioning

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
-	previous line, at first non-white
RETURN	same as carriage return; moves cursor to beginning of next line
j	next line, same column
k	previous line, same column

Cursor Positioning within Line

↑	first non white
Ø	beginning of line
$	end of line
h or ->	forward
l or <-	backwards
CTRL-H	same as <-
space	same as ->
fx	find x forward
Fx	f backward
tx	upto x forward
Tx	back upto x
;	repeat last f F t or T
,	inverse of ;
|	to specified column

Words, Sentences, Paragraphs

w	word forward
b	back word
e	end of word

)	beginning of next sentence
}	beginning of next paragraph
(	beginning of previous sentence
{	beginning of previous paragraph
W	blank delimited word
B	back W
E	to end of **W**

Corrections During Insert

CTRL-H	erase last character
CTRL-W	erases last word
erase	your erase; same as CTRL-h
kill	your kill; erase input this line
****	escapes CTRL-h; your erase and kill
ESC	ends insertions, back to command
CTRL-?	interrupt, terminates insert
CTRL-D	backtab over autoindent
^CTRL-D	kill autoindent, for one line only
ØCTRL-D	kills all autoindent
CTRL-V	quote non-printing character

Insert and Replace

a	append after cursor
i	insert before cursor
A	append at end of line
I	insert before first non-blank
o	open line below
O	Open above
rx	replace single character with x
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous Operations

C	change rest of line
D	delete rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete character at cursor
X	delete character before cursor
Y	yank lines

Yank and Put

p	put back line(s) after current line
P	put back line(s) before current line
xp	put from buffer x
"xd	delete into buffer x
"xy	yank to buffer x

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"np	retrieve nth last delete

Entering/Leaving vi

%vi name	edit name at top
ZZ	exit from vi, saving changes

The display

Last line	error messages, echoing input to :, /, ?, and !, feed back about i/o and large changes
@lines	on screen only, not in file (on dumb terminals)
~lines	lines past end of file
CTRL-x	control characters, CTRL-? is delete
tabs	expand to spaces, cursor at last

Simple Commands

dw	delete a word
de	delete a word, leave punctuation
dd	delete a line
3dd	delete 3 lines
itextESC	insert text text
cwnewESC	change word to new
easESC	pluralize word
xp	transpose characters

ZEUS FOR BEGINNERS*

* This information is based on an article originally authored by Brian W. Kernighan, Bell Laboratories.

Preface

This manual introduces the ZEUS operating system. It includes the basic procedures and commands needed for day-to-day use of the system. The major formatting programs and macro packages used for document preparation and hints on preparing documents are discussed. Descriptions of supporting software and ZEUS programming are also included.

This manual is divided into four sections. Section 1 describes how to log in, how to enter data, what to do about typing errors, and how to log out. Some of this information is dependent on the system and terminal that are being used, so this section must be supplemented by local information. Information required for day-to-day use of the system (such as commonly used commands) is found in Section 2. Section 3 describes some of the formatting tools used in preparing manuscripts. Some of the tools used for developing programs are described in Section 4.

For further information, refer to the System 8000 ZEUS Reference Manual and the System 8000 ZEUS Utilities Manual.

Table of Contents

SECTION 1
GETTING STARTED

1.1. Logging In

Terminals are connected to the system by a high-speed asynchronous line. Log in when the message login: appears on the terminal. If this message is not on the screen, press the RETURN key. If the message still does not appear, contact the system administrator for assistance.

When login: is displayed, enter the login name in lowercase, followed by a RETURN. For terminals that have only uppercase, it is possible to type commands in uppercase. If the login name is typed in uppercase, the entire terminal session must be performed in uppercase. The system does not respond until a RETURN is entered. If a password is required, the message Password: appears. Enter the password, followed by a RETURN. The password, which protects files from unauthorized access, is not echoed on the screen.

When a prompt character appears on the screen, the system is ready to accept commands. The prompt character is usually a dollar sign ($) or a percent sign (%). (Messages of the day or notifications that mail is being held can appear on the screen before the prompt character.)

1.2. Typing Commands

Once the prompt appears, commands (requests that the system do something) can be entered. Type the command

```
date
```

followed by a RETURN. A response similar to

```
Mon Jan 16 14:17:10 EST 1978
```

is displayed.

Always press RETURN after every command line; the system does not respond unless RETURN is pressed.

The command who specifies everyone who is currently logged in to the system. Entering

```
who
```

causes a response similar to the following:

```
ski      tty05   Jan 16    09:33
gam      tty11   Jan 16    13:07
```

The time specifies when the user logged in; ttyxx indicates the terminal being used.

If a typing mistake is made when a command is entered, thereby referencing a nonexistent command, the system responds with an error message. For example, typing

```
whom
```

results in the response

```
whom: not found
```

If the name of some other command is inadvertently typed, that command is run.

If the terminal does not have tabs, type the command

```
stty -tabs
```

The system then converts each tab into the correct number of spaces when printing. If the terminal does have computer-settable tabs, the command tabs sets the stops. Refer to stty(1) in the System 8000 ZEUS Reference Manual. (The notation stty(1) refers to the command stty in Section 1 of the System 8000 ZEUS Reference Manual).

1.3. Unusual Terminal Behavior

Sometimes the terminal functions incorrectly. For example, each letter may be typed twice, or RETURN may not cause a line feed or a return to the left margin. Logging out and logging back in may correct this.

1.4. Typing Errors

A typing error that is discovered before RETURN is typed can be corrected in one of two ways. Control-h (hitting "h" while holding down the control key) erases the last character typed. Control-h can be repeated to erase characters back to the beginning of the line (but not beyond).

Control-x erases the current input line. If a line of text has several errors, type control-x and then retype the line.

The system always echoes a new line after the control-x character.

The stty(1) command can be used to change the erase and kill characters. Backspace can also be used as an erase character, and control-x can be used as a kill character.

1.5. Read-Ahead

Read-ahead capability allows typing to be done as fast as possible, even while the system is responding to a command. If typing is done while the system is outputting text, the input characters appear intermixed with the output characters; however, they are interpreted in the correct order. Several commands can be typed one after another without waiting for each one to execute.

1.6. Stopping a Program

Most programs can be stopped by typing the character RUB (usually the delete or rubout key on the terminal). On most terminals, the "interrupt" or "break" key can also be used. In a few programs, such as the text editor, RUB stops whatever the program is doing but does not stop the program itself. Hanging up the phone also stops most programs, but this is not a recommended method of exiting a program.

1.7. Logging Out

To log out, type a control-d or type

```
logout
```

It is not sufficient to turn off the terminal because ZEUS does not use a time-out mechanism. When using a phone, it is possible to log out by hanging up, but this is not recommended.

1.8. Mail

After logging in, the message

```
you have mail.
```

may appear. ZEUS provides a postal system, allowing for communication with other users on the system. To read the mail, type the command

```
mail
```

Mail appears, one message at a time, with the most recent message given first. After each message, mail waits for a user response. Typing a d deletes the message. Typing RETURN causes mail to continue, leaving the message on the system; it will appear again the next time mail is read. Other responses are described in mail(1) of the System 8000 ZEUS Reference Manual.

To send mail to "joe" (a user whose login name is joe), type

```
mail joe
```

Then enter the text of the letter, using as many lines as necessary. After the last line of text, type control-d.

There are other ways to send mail. Mail can be sent to oneself as a handy reminder mechanism. Previously prepared mail can be sent to a number of people simultaneously. For more details, see mail(1).

1.9. Writing to Other Users

A message like

```
message from joe tty07...
```

may appear on the terminal, accompanied by a beep. This indicates that Joe is on line and wants to send a message. To respond, type the command

```
write joe
```

This establishes a two-way communication path, and messages can be exchanged via the terminals. This path is slow compared to system response in general. It is necessary to terminate any program that is being run before messages can be received. (It is possible to temporarily escape from the editor. Refer to the editor tutorial in the System 8000 ZEUS Utilities Manual.)

To keep the messages from becoming intermixed, care should be taken to ensure that both users do not type messages at the same time. A common way of doing this is to type an o on a line by itself at the end of the message to indicate that the message is over. To terminate a conversation, each side must type a control-d or a delete character on a line by itself.

If an attempt is made to write to someone who is not logged in, the system responds with the message

person not logged in

If an attempt is made to write to someone who does not want to be disturbed, the system responds with the message

permission denied

If the target person is logged in but does not answer, type control-d to obtain a prompt.

1.10. On-Line Manual

The System 8000 ZEUS Reference Manual is usually kept on line, and sections of it can be displayed at the terminal. The manual also contains the most up-to-date information on commands. To print a manual section, type

man command-name.

For example, to read about the who command, type

man who

1.11. Computer-Aided Instruction

The ZEUS system has a program called learn that provides computer-aided instruction on the file system and basic commands, the editor, document preparation, and programming in C. Enter the command

learn

for further information.

SECTION 2
DAY-TO-DAY USE

2.1. The Editor

The ZEUS text editor, ed, is usually used to type papers, letters, programs, and to store information in the computer. Refer to ed(1) and ED in the System 8000 ZEUS Reference Manual for in-depth explanations on how to use the editor.

To create a file called junk containing some text, enter

```
ed junk      (invokes the text editor; the system
              responds by listing the number of
              characters in the file)
a            (command to ed, to add text)
text
.            (signals the end of adding text)
```

A period (.) typed by itself at the beginning of a line indicates the end of text addition. Until it is entered, everything typed is treated as text to be added, and no other ed commands are recognized.

To store the information that has been typed into a file, use the editor command w. The editor responds by listing the number of characters in the file junk. Until the w command is entered, nothing is stored permanently. Therefore, if the user hangs up or logs out, the information is lost. (There is, however, a special feature of ZEUS that saves the edited data in a file called ed.hup.k) After a w command is issued, the stored information can be accessed at any time by typing

```
ed junk
```

To exit from the editor, type a quit (q) command. If the q command is entered before the text has been stored, ed prints a ? as a reminder. Entering a second q followed by an exclamation point (!) causes the exit to take place.

Now create a second file called temp in the same manner. Two files, junk and temp, should now exist.

2.2. The List Commands

The list (ls) command lists the names (not contents) of all files in the directory. If

```
ls
```

is typed, the response is

```
junk
temp
```

These are the two files just created. Unless an optional argument is added to the ls command, the names are listed alphabetically. Other variations are possible. For example, the command

```
ls -t
```

lists the files in the order in which they were last changed, with the most recently changed file listed first. Typing

```
ls -l
```

produces a long listing similar to the following:

```
-rwxrwxrwx  1 bwk   41 Jul 22 2:56 junk
-rwxrwxrwx  1 bwk   78 Jul 22 2:57 temp
```

The date and time indicate when the last changes to the file were made. The 41 and 78 refer to the number of characters in the file. The initials bwk indicate the owner of the file, that is, the person who created it. The -rwxrwxrwx specifies who has permission to read, write, and execute the file. The first dash in each line indicates an ordinary file; a d instead of a dash indicates a directory. The left-most rwx indicates the read, write, and execute permissions for the owner of the file. The middle rwx pertains to the read, write, and execute permissions for the user group to which the owner belongs. The right-most rwx pertains to everyone else. In this example, everyone has read, write, and execute permission. For more information, refer to chmod(1) and chmod(2).

Listing options can be combined. For example, the command ls -lt gives a long listing (-l) in time order (-t). More information is found in ls(1).

The use of optional arguments that begin with a dash (like -t and -lt) is a common convention for ZEUS programs. In

general, if a program accepts such optional arguments, they precede any file name arguments. The various arguments must be separated with a blank space (ls-l is not the same as ls -l).

2.3. Displaying Files

Use the editor to display a file of text on the screen. Type

```
ed junk
1,$p
```

and ed lists the number of characters in junk and then displays the entire file on the screen.

It is not always feasible to use the editor for displaying files. There is a limit to the size of files that ed can handle, and only one file can be displayed at a time. There are alternate programs suitable to specific applications.

The cat command displays the contents of all the files named in a list. For example,

```
cat junk
```

displays the file junk, and

```
cat junk temp
```

displays the files junk and temp. The files are simply concatenated (hence the name cat) onto the screen.

The pr command produces formatted displays of files. As with cat, pr displays all the files named in a list, but pr displays text in formatted form, including headings with date, time, page number, and file name at the top of each page. The command

```
pr junk temp
```

displays junk, then skips to the top of a new page and displays temp.

The pr command can also produce multicolumn output. For example,

```
pr -3 junk
```

prints the file junk in three-column format. Any number of columns can be printed. See pr(1) for more information.

The command dog displays the contents of a specified file one page at a time. For example,

```
dog junk
```

displays the first page of the file junk on the terminal. Pressing the RETURN key causes the text to scroll forward, displaying the next page.

There are also programs that print ZEUS files on a high-speed printer. See lpr in the ZEUS Reference Manual. The nroff and troff programs are more complete text formatters. They are discussed in Section 3 and in the ZEUS Utilities Manual.

2.4. Rearranging Files

A file can be moved from one place to another (which amounts to changing the name) using the mv command. For example, typing

```
mv junk stuff
```

moves the contents of the file junk into the file stuff. If the ls command is entered, the response is now

```
stuff
temp
```

NOTE

If a file is moved to another file that already exists, the already existing contents are lost forever.

To make a copy of a file, use the cp command.

```
cp stuff templ
```

makes a duplicate copy of stuff in templ.

The rm command removes (deletes) files from a directory. For example,

```
rm temp templ
```

deletes the files temp and temp1.

NOTE

Be very careful when using the rm command. Once files are removed with the rm command, they no longer exist in the directory and can never be recovered.

A warning is displayed if one of the named files does not exist. Otherwise rm, like most ZEUS commands, does its work silently.

2.5. File Names

File names can be no longer than 14 characters. Although almost any character can be used in a file name, it is recommended that only letters, numbers, and the period be used. This is to avoid characters that might have other meanings. For example, if a file were created with the name -t, listing it by name would be difficult, if not impossible, because -t is an optional argument for requesting a time-order listing.

If a large manual is being typed, it must be divided into several smaller sections because the size of files that ed can handle is limited. The document should therefore be typed as a number of smaller files. Each chapter can be in a separate file named chap1, chap2, etc., or each chapter can be broken into several files named chap1.1, chap1.2, chap1.3, chap2.1, chap2.2, etc. This naming system makes the relationship between the files obvious.

One advantage to a systematic naming convention is that the entire book can be displayed with one command, such as

```
pr chap*
```

The asterisk (*) is a pattern matching character that means "anything at all," so this command prints in alphabetical order all files whose names begin with chap. This shorthand notation is used system-wide, not just with pr. For example, to list all the names of the files in the manual, enter

```
ls chap*
```

This lists

```
chap1.1
```

```
chap1.2
chap1.3
...
```

The * is not limited to the last position in a file name--it can be anywhere and can occur several times. For example,

```
rm *junk* *temp*
```

removes all files that contain junk or temp as any part of their name. As a special case, * by itself matches every filename, so

```
pr *
```

prints all the user's files in alphabetical order; and

```
rm *
```

removes all files in the current directory.

The * is not the only pattern-matching feature available. It is possible to match a group of characters by enclosing them in brackets ([ ]). For example, if only Chapters 1 through 4 and Chapter 9 are to be printed, type

```
pr chap[12349]*
```

A range of consecutive letters or digits can be abbreviated.

```
pr chap[1-49]*
```

A range of letters can also be specified with brackets. For example, [a-z] matches any character in the range a through z.

The question mark (?) pattern matches any single character. For example,

```
ls ?
```

lists all files that have single-character names, and

```
ls -l chap?.1
```

lists the first file of each chapter (chap1.1, chap2.1).

To cancel the special meaning of * or ?, enclose the argument in single quotes.

```
ls '?'
```

2.5.1. Directories and Path Names: Generally, each user has a private directory containing only the files that belong to that user. When logged in, the user is in his/her private directory, and unless special action is taken when a new file is created, it is created in the directory the user is currently in. This is most often the user's own directory, and therefore, the file is unrelated to any other file of the same name that exists in someone else's directory.

All files are organized in sets located in a tree, with the individual user's files located several branches outward from the root. Any file in the system can be found by starting at the root of the tree and moving along the proper set of branches. It is also possible to move inward toward the root.

The command pwd (print working directory) prints the path name of the directory the user is currently in.

The response to the pwd command is something similar to

```
/z/your-name
```

This indicates that the user is currently in the directory your-name, which is in the directory /z, which is, in turn, in the root directory, called /.

Typing

```
ls /z/your-name
```

lists the same file names obtained from the ls command alone. With no arguments, ls lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Typing

```
ls /z
```

prints a series of names, among which is your-name. In many installations, z is a directory that contains the directories of all users of the system.

Typing

```
ls /
```

gives a response something like:

```
bin
```

```
dev
etc
lib
tmp
usr
```

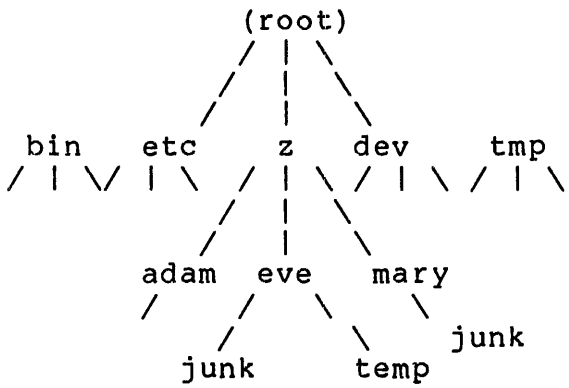
These are the basic directories of files--the root of the tree.

The full name of the path to be followed from the root through the tree of directories to get to a particular file is the path name. The path name of the file junk is

```
/z/your-name/junk
```

It is a universal rule in the ZEUS system that anywhere an ordinary file name can be used, a path name can be used.

Here is a picture of the tree used in this document:



Observe that mary's junk file is unrelated to eve's junk file.

To obtain a listing of files in another user's directory, type

```
ls /z/neighbor-name
```

To copy of one of these files, type

```
cp /z/your-neighbor/his-file yourfile
```

If users do not want other people examining these files, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which control file access. (See ls(1) and chmod(1) for details.) For an executable file, the owner generally has read, write, and execute permission; other

people in the owner's group might have read or execute permission; everyone else might have only execute permission.

As a final experiment with path names, try

```
ls /bin /z/bin
```

2.5.2. Current Directory: When the name of a file (command) is entered in response to the prompt character, the system looks for an executable file of that name in the current directory. If the file is not found in the current directory, the system searches /bin, and finally /usr/bin. The search path, which is normally the current directory, /bin, and /usr/bin can be changed. (See sh(1) and csh(1) in the ZEUS Reference Manual.)

If a user works regularly with someone else on common information in the other's directory, the user could simply log in under the other's login name each time the information is needed. It is also possible to change directories. Type

```
cd /z/your-friend
```

and a file name used with a command like cat or pr refers to the file in your-friend directory. Changing directories does not affect any permissions associated with a file. That is, if a file could not be accessed from the user's own directory, changing to another directory does not alter that fact.

Type

```
pwd
```

to find out which directory is the current directory.

2.5.3. Subdirectories: It is convenient to arrange files so that all files on a related subject are in a directory that is separate from other projects. For example, when writing a manual, it might be helpful to keep the text in a directory called book. To make the directory, use the command

```
mkdir book
```

This creates the directory called book. To go to that directory, type

```
cd book
```

Separate files can now be established in this directory. The path name of this directory is:

```
/z/your-name/book
```

To move back up to the login directory (one level up in the tree), type

```
cd ..
```

The double period (..) indicates the parent of the currently accessed directory. A single period (.) is an alternate name for the working directory.

To remove the directory book, type

```
rm book/*
rmdir book
```

The first command removes all files from the directory, and the second removes the empty directory.

2.6. Using Files Instead of Terminal Input and Output

Most of the commands discussed so far produce output on the terminal. Some, like the editor, also take their input from the terminal. In ZEUS systems, input, output, or both can go to or from files rather than the terminal. For example,

```
ls
```

lists all files on the terminal screen. However, entering

```
ls >filelist
```

places a list of files in the file filelist, which is created if it does not exist or is overwritten if it does. The symbol > means that the output should go to the following file rather than the terminal screen. Several files can be combined into one by capturing the output of cat in a file. For example,

```
cat f1 f2 f3 >temp
```

This concatenates f1, f2, and f3 into the file temp.

The symbol >> operates very much like > does. It means add the listed files to the end of the file that follows the symbol. That is,

```
cat f1 f2 f3 >>temp
```

means to add f1, f2, and f3 to the end of whatever is already in temp (instead of overwriting the existing contents of temp). As with >, if temp does not exist, it is created.

The symbol < means take the input for a program from the following file instead of from the terminal. For example, it is possible to create a file called script containing a group of editing commands that produces a specified set of changes. Typing

```
ed file <script
```

causes the set of editing commands to be executed throughout the file.

As another example, ed can be used to prepare a letter in the file let. Then, the letter can be sent to several people with

```
mail adam eve mary joe <let
```

2.7. Pipes

A pipe is a means of connecting the output of one program to the input of another program so that the two run as a sequence of processes. A command line that uses pipes is called a pipeline.

For example,

```
pr f g h
```

displays the files f, g, and h, beginning each on a new page. It is possible to display them together without page breaks by entering

```
cat f g h >temp
pr <temp
rm temp
```

A simpler way to do this is to take the output of cat and connect it to the input of pr by using a pipe.

```
cat f g h | pr
```

The vertical bar (|), which is the pipe command, means take the output from cat, which would normally have gone to the

terminal, and put it into pr to be formatted.

The pipeline

```
ls | pr -3
```

displays a list of files in three columns.

Any program that reads from the terminal can also read from a pipe. Any program that writes to the terminal can also drive a pipe. Any number of elements can be used in a pipeline.

Many ZEUS programs are written so that they can take their input from one or more files if file arguments are given. If no arguments are given, the programs read from the terminal and can be used in pipelines. One example is pr.

```
pr -3 a b c
```

prints files a, b, and c in order, in three-column format. The command

```
cat a b c | pr -3
```

produces the same output; pr prints the information coming down the pipeline in three-column format.

2.8. The Shell

The shell is the program that interprets the commands and arguments entered at the terminal. (See sh(1) and csh(1).) It also interprets characters that have special meaning in ZEUS. For example, two programs can be run with one command line by separating the commands with a semicolon (;). The shell recognizes the semicolon and breaks the line into two commands. In the command line

```
date; who
```

the shell executes the date and who commands before returning with a prompt character.

More than one program can be run simultaneously. For example, if something time consuming, like the editor script, is being run, type

```
ed file <script &
```

The ampersand at the end of a command line means start the

command running in the background and then take further commands from the terminal immediately. To prevent the output from interfering with what is being done on the terminal, type

```
ed file <script >script.out &
```

which saves the output lines in a file called script.out.

When initiating a command with &, the system replies with a number called the process number, which identifies the command so that it can be stopped later. To stop the command from executing, type

```
kill process-number
```

If the process number is forgotten, the command ps lists the process numbers of everything that ls is running. (It is possible to use the command kill Ø, which kills all the user processes that are running. This command should, of course, be used with caution.) The command ps -a lists all programs in the system that are currently running.

The command

```
(command-1; command-2; command-3) &
```

can be used to start three commands in the background. A background pipeline can be started with

```
command-1 | command-2 &
```

Just as the editor or some similar program can take its input from a file instead of from the terminal, the shell can read a file to get commands. For instance, suppose the tabs on the terminal are to be set, and the date and who is on the system are to be displayed every time the user logs in. The three necessary commands (tabs, date, who) can be put into a file called startup. To run this program, type

```
sh startup
```

The shell then runs with the file startup as input. This has the same effect as entering the contents of startup on the terminal.

To eliminate the need to type sh each time, use the command

```
chmod +x startup
```

The chmod command marks the file as executable; the shell

recognizes this and runs it as a sequence of commands. Thereafter, type only

```
startup
```

to run the sequence of commands.

If startup is to be run automatically after every login, place its contents in the current home directory in a file called .profile (if running in shell), or .cshrc (f the shell running is the C shell). When the shell gains control after the login, it looks for and executes the .profile or .cshrc file.

SECTION 3
DOCUMENT PREPARATION

3.1. Introduction

The ZEUS system has two major formatting programs for document preparation: nroff, which produces output on terminals and line printers, and troff, which drives a phototypesetter.

3.2. Formatting Programs

Formatting programs use commands that are entered along with the text that is to be formatted. The commands indicate in detail how the formatted text is to look. For example, there are commands that specify how long lines should be, whether to use single or double spacing, and what running titles are to be used on each page.

For nroff and troff, several packages of canned formatting requests called macro packages are available. These allow specification of formatting elements such as paragraphs, running titles, footnotes, and multicolumn output. It is not necessary to learn nroff and troff to use these macro packages. Formatting requests typically consist of a period and two uppercase letters; for example, .TL is used to introduce a title, and .PP is used to begin a new paragraph.

A document is typed so that it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
```

The precise meaning of .PP depends on whether the output device being used is a typesetter or terminal. For example, a paragraph is normally preceded by a space (one line in

nroff, one half line in troff), and the first word is indented. These rules can be changed as required.

To print a document in standard format using -ms, use the command

```
troff -ms files
```

for the typesetter and

```
nroff -ms files
```

for a terminal. The -ms argument tells troff and nroff to use the manuscript package of formatting requests. (Refer to ms(7) for more information.)

There are several similar packages; see the information on text formatting in the System 8000 ZEUS Utilities Manual.

3.3. Supporting Tools

In addition to the basic formatters, there are other supporting programs for document preparation.

Any spelling errors in a document can be detected by the programs spell and typo. The spell program compares the words in the document to a dictionary, then prints those that are not in the dictionary. The typo program searches for words that are unusual, then prints them.

The grep program examines a set of files for lines that contain a particular text pattern. For example,

```
grep 'ing$' chap*
```

finds all lines that end with the letters ing in the files chap*. (It is always good practice to put single quotes around the pattern being searched for, in case it contains characters like * or $ that have a special meaning to the shell.) The grep program is useful for discovering which set of files contains the misspelled words detected by spell.

A list of the differences between two files is printed by diff. Two versions of something can be compared automatically, eliminating the necessity of proofreading.

The words, lines, and characters in a set of files are counted by wc.

The tr program translates characters into other characters. For example, it converts uppercase to lowercase and vice versa. The following command translates uppercase into lowercase:

```
tr A-Z a-z <input >output
```

Files can be sorted in a variety of ways by sort.

The ptx program makes a permuted index (keyword-in-context listing).

The sed program provides many of the editing facilities of ed, but can apply them to arbitrarily long inputs.

For more information on these programs, see the System 8000 ZEUS Reference Manual.

3.4. Hints for Preparing Documents

Most documents go through several drafts before they are finished. The following hints make the process of revising drafts easier.

When the text is being typed, start each sentence on a new line, make lines short, and break lines at natural places, such as after commas and semicolons. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions will simplify any editing done to the document.

Keep the individual files of a document short (perhaps ten to fifteen thousand characters). Larger files edit more slowly, and of course, if an error is made, it is better to have destroyed a small file rather than a big one. Split documents into files at natural boundaries.

Refrain from deciding formatting details too early. One of the advantages of the formatting packages is that they permit decisions to be delayed until the last possible moment. As long as the text has been entered in some systematic way, it can always be cleaned up and reformatted by a judicious combination of editing commands and request definitions.

SECTION 4
PROGRAMMING

4.1. Introduction

The ZEUS system is a productive programming environment because it offers a rich set of programming tools. Facilities such as pipes, I/O redirection, and the capabilities of the shell make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The pipe mechanism allows fabrication of complicated operations out of spare parts that already exist. For example, an early version of the spell program was

```
cat ... | tr ... | tr ... | sort | uniq | comm
```

where cat collected the files, the first tr put each word on a new line, and the second tr deleted punctuation. The information was then sorted into dictionary order. The uniq command discarded duplicates, and comm printed words that were in the text but not found in the dictionary.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines for each file in a set of files, the following can be laboriously typed

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

An easier way is to type

```
ls chap* >temp
```

This lists the file names in the temp file. Then this file can be edited to incorporate the necessary series of editing commands (using the global commands of ed). When these commands have been written into script, the command

```
ed <script
```

produces the same output as the laboriously typed list of commands. Alternately, since the shell performs loops, it is possible to repeat a set of commands over and over again for a set of arguments. For example,

```
for i in chap*
do
        ed $i <script
done
```

sets the shell variable i to each file name in turn, then does the command. This command can be typed at the terminal or put in a file for later execution.

4.2. Programming the Shell

The shell itself is a programming language with variables, control flow (if-else, while, for, case), subroutines, and interrupt handling. Since there are many building-block programs, a new program can sometimes be created by piecing together some of the building blocks with shell command files.

Examples and rules for running the shell and the C shell can be found in SHELL, and CSHELL in the System 8000 ZEUS Utilities Manual.

4.3. Programming in C

ZEUS and most of the programs that run on it are written in C. C is an easy language to learn and use. It is introduced and fully described in The C Programming Language by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). See the System 8000 ZEUS Reference Manual for additional information.

Systems Publications

Reader's Comments

Your feedback about this document helps us ascertain your needs and fulfill them in the future. Plea take the time to fill out this questionaire and return it to us. this information will be helpful to us and, time, to future users of Zilog products.

Your Name: _____

Company Name: _____

Address: _____

Title of this document: _____

Briefly describe application: _____

_____

_____

Does this publication meet your needs? ☐ Yes ☐ No If not, why not? _____

_____

_____

How are you using this publication?

- ☐ As an introduction to the subject?
- ☐ As a reference manual?
- ☐ As an instructor or student?

How do you find the material?

	Excellent	Good	Poor
Technicality	☐	☐	☐
Organization	☐	☐	☐
Completeness	☐	☐	☐

What would have improved the material? _____

_____

_____

Other comments and suggestions: _____

_____

_____

If you found any mistakes in this document, please let us know what and where they are: _____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 35 CAMPBELL, CA.

POSTAGE WILL BE PAID BY ADDRESSEE

Zilog

Systems Publications
1315 Dell Avenue
Campbell, California 95008
Attn: Publications Manager