DOCUMENTS

PDP–11 Version 2.9

July, 1983

Second Berkeley Software Distribution
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

DOCUMENTS

PDP–11 Version 2.9

Second Berkeley Software Distribution

Installing and Operating 2.9BSD
March 26, 1984

Michael J. Karels
Carl F. Smith

University of California
Berkeley, California 94720

ABSTRACT

This document contains instructions for installation and operation of the Second Berkeley Software Distribution's 2.9BSD release of the PDP-11† UNIX‡ system. It is adapted from the paper *Installing and Operating 4.1bsd* by Bill Joy.

This document explains the procedures for installation of Berkeley UNIX on a PDP-11 or to upgrade an existing Berkeley PDP-11 UNIX system to the new release. It then explains how to configure the kernel for the available devices and user load, lay out file systems on the available disks, set up terminal lines and user accounts, and do system specific tailoring. It also explains system operations procedures: shutdown and startup, hardware error reporting and diagnosis, file system backup procedures, resource control, performance monitoring, and procedures for recompiling and reinstalling system software. Technical details on the kernel changes are presented in the accompanying paper, "Changes in the Kernel in 2.9BSD."

The 2.9BSD release, unlike previous versions of the Second Berkeley Software Distribution, is a complete Version 7 UNIX system with all of the standard UNIX tools and utilities, with or without Berkeley modifications. Therefore, it does not need to be layered onto an existing Version 7 system; because of the many changes and additions throughout the system, it would require a substantial effort to merge into most earlier systems.

†DEC, MASSBUS, PDP, and UNIBUS are trademarks of Digital Equipment Corporation.
‡UNIX is a trademark of Bell Laboratories.

1. INTRODUCTION

This document explains how to install the 2.9BSD release of the Berkeley version of UNIX for the PDP-11 on your system. If you are running the July 1981 release of the system, which was called 2.8BSD, you can avoid a full bootstrap from the new tape by extracting only the software that has changed. Be warned, however, that there are a large number of changes. Unless you have many local modifications it will probably be easier to bring up an intact 2.9BSD system and merge your local changes into it. If you are running any other version of UNIX on your PDP-11, you will have to do a full bootstrap. This means dumping all file systems which are to be retained onto tape in a format that can be read in again later (*tar* format is best, or V7 *dump* if the file system configuration will be the same). A new root file system can be made and read in using standalone utilites on the tape. The system sources and the rest of the /usr file system can then be extracted. Finally, old file systems can be reloaded from tape.

To get an overview of the process and an idea of some of the alternative strategies that are available, it is wise to look through all of these instructions before beginning.

1.1. Hardware supported

This distribution can be booted on a PDP-11/23, 24, 34, 34A, 40, 44, 45, 55, 60, or 70 CPU with at least 192 Kbytes of memory and any of the following disks†:

DEC MASSBUS:	RM03, RM05, RP04, RP05, RP06
DEC UNIBUS:	RK05, RK06, RK07, RL01, RL02, RM02, RP03, RP04, RP05, RP06
AED 8000 UNIBUS:	AMPEX DM980 (emulating RP03)
AED STORM-II	AMPEX DM980 (emulating RM02)
DIVA COMP V MASSBUS:	AMPEX 9300
EMULEX SC-21 UNIBUS:	AMPEX 9300, CDC 9766 (emulating RM05)
EMULEX SC-11 or SC-21 UNIBUS:	CDC 9762, AMPEX DM980

The tape drives† supported by this distribution are:

DEC MASSBUS:	TE16, TU45, TU77
DEC UNIBUS:	TE10, TE16, TS11, TU45, TU77
DATUM 15X20 UNIBUS:	KENNEDY 9100 (emulating TE10)
EMULEX TC-11 UNIBUS:	KENNEDY 9100, 9300 (emulating TE10)

1.2. Distribution format

The distribution format is two 9-track 800bpi or one 1600bpi 2400' magnetic tape(s). If you are able to do so, it is a good idea to immediately copy the tape(s) in the distribution kit to guard against disaster. The first tape contains some 512-byte records, some 1024-byte records, followed by many 10240-byte records. There are interspersed tape marks; end-of-tape is signaled by a double end-of-file. The second tape contains only 10240-byte records with no interspersed tape marks.

The boot tape contains several standalone utility programs, a *dump* image of a root file system, and a *tar* image of part of the /usr file system. The files on this tape are:

† Other controllers and drives may be easily usable with the system, but might require minor modifications to the system to allow bootstrapping. The controllers and the drives shown here are known to work as bootstrap devices.

File	Contents	Record Size
0	boot block	512
	(EOR)	
	boot block	512
	(EOR)	
	Standalone **Boot**	512
	(EOF)	
1	Standalone **cat**	1024
	(EOF)	
2	This index	1024
	(use **cat** to read)	
	(EOF)	
3	Standalone **mkfs**	1024
	(see *mkfs*(8)†)	
	(EOF)	
4	Standalone **restor**	1024
	(see *restor*(8))	
	(EOF)	
5	Standalone **icheck**	1024
	(see *icheck*(8))	
	(EOF)	
6	Dump of small "root" file system	10240
	(217 10K-byte blocks; see *dump*(8))	
	(EOF)	
7	Tar archive of /usr files	10240
	(most of the tape; see *tar*(1))	
	(EOF)	
	(EOF)	

The last file on the 800bpi tape is a *tar* image of most of the /usr file system except for sources (relative to /usr; see *tar*(1)). It contains most of the binaries and other material which is normally kept on-line, with the following directories: **70 adm bin contrib dict doc games include lib local man msgs preserve public spool sys tmp ucb**. It contains 1785 10K byte blocks. The second 800bpi tape contains one file in *tar* format, again relative to /usr, consisting of 1903 10K byte blocks and containing the source tree with all command and kernel sources. It contains the directories **ingres**, **net**, and **src**, and includes the Berkeley additions (which are in /usr/src/ucb and /usr/ingres). The **net** directory contains sources for the TCP/IP system. On the 1600bpi tape, the two *tar* images are combined into one tape file of 3687 10K byte blocks.

1.3. UNIX device naming

UNIX has a set of names for devices that are different from the DEC names for the devices. The disk and tape names used by the bootstrap and the system are:

†References of the form *X*(*Y*) mean the subsection named *X* in section *Y* of the Berkeley PDP-11 UNIX Programmer's manual.

RK05 disks	**rk**
RK06, RK07 disks	**hk**
RL01, RL02 disks	**rl**
RP02, RP03 disks	**rp**
TE16, TU45, TU77/TM02, 3 tapes	**ht**
TE10/TM11 tapes	**tm**
TS11 tapes	**ts**

There is also a generic disk driver, **xp**, that will handle most types of SMD disks on one or more controllers (even different types on the same controller). The **xp** driver handles RM02, RM03, RM05, RP04, RP05 and RP06 disks on DEC, Emulex, and Diva UNIBUS or MASSBUS controllers.

The standalone system used to bootstrap the full UNIX system uses device names of the form:

xx(*y*,*z*)

where *xx* is one of **hk**, **ht**, **rk**, **rl**, **rp**, **tm**, **ts**, or **xp**. The value *y* specifies the device or drive unit to use. The *z* value is interpreted differently for tapes and disks: for disks it is a block offset for a file system and for tapes it is a file number on the tape.

Large UNIX physical disks (**hk**, **rp**, **xp**) are divided into 8 logical disk partitions, each of which may occupy any consecutive cylinder range on the physical device. The cylinders occupied by the 8 partitions for each drive type are specified in section 4 of the Berkeley PDP-11 UNIX Programmer's manual.† Each partition may be used for either a raw data area such as a swapping area or to store a UNIX file system. It is conventional for the first partition on a disk to be used to store a root file system, from which UNIX may be bootstrapped. The second partition is traditionally used as a swapping area, and the rest of the disk is divided into spaces for additional "mounted file systems" by use of one or more additional partitions.

The disk partitions have names in the standalone system of the form "*xx*(*y*,*z*)" as described above. Thus partition 1 of an RK07 at drive 0 would be "hk(0,5940)". When not running standalone, this partition would normally be available as "/dev/hk0b". Here the prefix "/dev" is the name of the directory where all "special files" normally live, the "hk" serves an obvious purpose, the "0" identifies this as a partition of hk drive number "0" and the "b" identifies this as partition 1 (where we number from 0, the 0th partition being "hk0a"). Finally, "5940" is the sector offset to partition 1, as determined from the manual page *hk*(4).

Returning to the discussion of the standalone system, we recall that tapes also took two integer parameters. In the case of a TE16/TU tape formatter on drive 0, the files on the tape have names "ht(0,0)", "ht(0,1)", etc. Here "file" means a tape file containing a single data stream separated by a single tape mark. The distribution tapes have data structures in the tape files and though the first tape contains only 8 tape files, it contains several thousand UNIX files.

1.4. UNIX devices: block and raw

UNIX makes a distinction between "block" and "character" (raw) devices. Each disk has a block device interface where the system makes the device byte addressable and you can write a single byte in the middle of the disk. The system will read out the data from the disk sector, insert the byte you gave it and put the modified data back. The disks with the names "/dev/*xx*0a", etc. are block devices and thus use the system's normal buffering mechanism. There are also raw devices available, which do physical I/O operations directly from the program's data area. These have names like "/dev/r*xx*0a", the "r" here standing for "raw." In the bootstrap procedures we will often suggest using the raw devices, because these tend to work

†It is possible to change the partitions by changing the values in the disk's sizes array in ioconf.c.

faster. In general, however, the block devices are used. They are where file systems are "mounted." The UNIX name space is increased by logically associating (*mount*ing) a UNIX file system residing on a given block device with a directory in the current name space. See *mount*(2) and *mount*(8). This association is severed by *umount.*

You should be aware that it is sometimes important to use the character device (for efficiency) or not (because it wouldn't work, e.g. to write a single byte in the middle of a sector). Don't change the instructions by using the wrong type of device indiscriminately.

1.5. Reporting problems or questions

Problems with the software of this distribution, or errors or omissions in the documentation, should be reported to the 2BSD group. For bug reports and fixes, the address is:

2bsd-bugs@berkeley (by ARPAnet)
or
ucbvax!2bsd-bugs (by UUCP)

These reports or fixes are expected to be in the format generated by the *sendbug*(1) program. For administrative concerns, use:

2bsd@berkeley (by ARPAnet)
or
ucbvax!2bsd (by UUCP)

A redistribution list of users who have indicated that they would like to receive bug reports is also maintained:

2bsd-people@berkeley (by ARPAnet)
or
ucbvax!2bsd-people (by UUCP)

This list may also be used as a general forum for help requests, sharing common experiences, etc. Requests to be added to or deleted from this list should be made to the 2bsd address above. If it is not possible to use electronic mail, then call or write the 2BSD office. Although there is seldom someone there to take your call, there is an answering machine, and your request will be forwarded to the appropriate person. The phone number and mailing address are:

Berkeley PDP-11 Software Distribution – 2BSD
Computer Science Division, Department of EECS
573 Evans Hall
University of California, Berkeley
Berkeley, California 94720
(415) 642-6258

2. BOOTSTRAP PROCEDURES

This section explains the bootstrap procedures that can be used to get one of the kernels supplied with this tape running on your machine. If you are not yet running UNIX or are running a version of UNIX other than 2.8BSD, you will have to do a full bootstrap.

If you are running 2.8BSD you can use the update procedure described in section 4.2 instead of a full bootstrap. This will affect modifications to the local system less than a full bootstrap. Note, however, that a full bootstrap will probably require less effort unless you have made major local modifications which you must carry over to the new system.

If you are already running UNIX and need to do a full bootstrap you should first save your existing files on magnetic tape. The 2.9BSD file system uses 1K-byte blocks by clustering disk blocks (as did the 2.8BSD system); file systems in other formats cannot be mounted. **Those upgrading from 2.8 should note that 2.9BSD uses generally different file system partition sizes than 2.8BSD, and that a few of the major device numbers have changed (in particular, that for the hk).** The easiest way to save the current files on tape is by doing a full dump and then restoring in the new system. This works also in converting V7, System-III, or System-V 512-byte file systems. Although the dump format is different on V7, System-III, and System-V, *512restor*(8) can restore old format V7 *dump* image tapes into the file system format used by 2.9BSD. *Tar*(1) can also be used to exchange files from different file system formats, and has the additional advantage that directory trees can be placed on different file systems than on the old configuration. Note that 2.9BSD does not support *cpio* tape format.

The tape bootstrap procedure involves three steps: loading the tape bootstrap monitor, creating and initializing a UNIX "root" file system system on the disk, and booting the system.

2.1. Booting from tape

To load the tape bootstrap monitor, first mount the magnetic tape on drive 0 at load point, making sure that the write ring is not inserted. Then use the normal bootstrap ROM, console monitor or other bootstrap to boot from the tape. If no other means are available, the following code can be keyed in and executed at (say) 0100000 to boot from a TM tape drive (the magic number 172526 is the address of the TM-11 current memory address register; an adjustment may be necessary if your controller is at a nonstandard address):

```
012700   (mov $172526, r0)
172526
010040   (mov r0, -(r0))
012740   (mov $60003, -(r0))
060003
000777   (br .)
```

When this is executed, the first block of the tape will be read into memory. Halt the CPU and restart at location 0.

The console should type

*nn***Boot**
:

where *nn* is the CPU class on which it believes it is running. The value will be one of 24, 40, 45 or 70, depending on whether separate instruction and data (separate I/D) and/or a UNIBUS map are detected. The CPUs in each class are:

Class	PDP11s	Separate I/D	UNIBUS map
24	24	-	+
40	23, 34, 34A, 40, 60	-	-
45	45, 55, 73, 83	+	-
70	44, 70, 84	+	+

The bootstrap can be forced to set up the machine as for a different class of PDP11 by placing an appropriate value in the console switch register (if there is one) while booting it. The value to use is the PDP11 class, interpreted as an *octal* number (use, for example, 070 for an 11/70). **Warning:** some old DEC bootstraps use the switch register to indicate where to boot from. On such machines, if the value in the switch register indicates an incorrect CPU, be sure to reset the switches immediately after initiating the tape bootstrap.

You are now talking to the tape bootstrap monitor. At any point in the following procedure you can return to this section, reload the tape bootstrap, and restart.

To first check that everything is working properly, you can use the *cat* program on the tape to print the list of utilities on the tape. Through the rest of this section, substitute the correct disk type for *dk* and the tape type for *tp*. In response to the prompt of the bootstrap which is now running, type

tp(0,1) (load file 1 from tape 0)

Cat will respond

Cat
File?

The table of contents is in file 2 on the tape, therefore answer

tp(0,2)

The tape will move, then a short list of files will print on the console, followed by:

exit called
*nn***Boot**
:

After *cat* is finished, it returns to the bootstrap for the next operation.

2.2. Creating an empty UNIX file system

Now create the root file system using the following procedures. First determine the size of your root file system from the following table:

Disk	Root File System Size (1K-byte blocks)
hk	2970
rk†	2000
rl01†	4000
rl02†	8500
rp	5200
xp	4807 (RP04/RP05/RP06)
	2400 (RM02/RM03)
	5168 (RM05)
	4702 (DIVA)

†These sizes are for full disks less some space used for swapping.

If the disk on which you are creating a root file system is an **xp** disk, you should check the drive type register at this time to make sure it holds a value that will be recognized correctly by the driver. There are numbering conflicts; the following numbers are used internally:

Drive Type Register Low Byte (nominal address: 0776726)	Drive Assumed
022	RP04/05/06
025	RM02/RM03
027	RM05
076	Emulex SC-21/300 Mb 815 cylinder RM05 emulation
077	Diva Comp-V/300 Mb SMD

Check the drive type number in your controller manual, or halt the CPU and examine this register. If the value does not correspond to the actual drive type, you must place the correct value in the switch register after the tape bootstrap is running and before any attempt is made to access the drive. This will override the drive type register. This value must be present at the time each program (including the bootstrap itself) first tries to access the disk. On machines without a switch register, the *xptype* variable can be patched in memory. After starting each utility but before accessing the disk, halt the CPU, place the new drive type number at the proper memory location with the console switches or monitor, and then continue. The location of *xptype* in each utility is mkfs: 032724, restor: 031614, icheck: 030174 and boot: 0430000 (the location for boot is higher because it relocates itself). Once UNIX itself is booted (see below) you must patch it also.

Finally, determine the proper interleaving factors *m* and *n* for your disk and CPU combination from the following table. These numbers determine the layout of the free list that will be constructed; the proper interleaving will help increase the speed of the file system. If you have a non-DEC disk that emulates one of the disks listed, you may be able to use these numbers as well, but check that the actual disk geometry is the same as the emulated disk (rather than the controller mapping onto a different physical disk). Also, the rotational speed must be the same as the DEC disk for these numbers to apply.

Disk Interleaving Factors for Disk/CPU Combinations (*m*/*n*)

CPU	RK05	RK06/7	RL01/2	RM02	RM03	RM05	RP03	RP04/5/6
11/23	X/12	X/33	X/10	X/80	-	-	X/100	X/209
11/24	X/12	7/33	X/10	10/80	-	-	X/100	10/209
11/34	X/12	6/33	X/10	8/80	-	-	3/100	8/209
11/40	2/12	6/33	X/10	8/80	-	-	3/100	8/209
11/44	X/12	4/33	X/10	6/80	-	-	2/100	6/209
11/45	2/12	5/33	X/10	7/80	-	-	3/100	7/209
11/55	X/12	5/33	X/10	7/80	-	-	3/100	7/209
11/60	X/12	5/33	X/10	7/80	-	-	3/100	7/209
11/70	X/12	3/33	X/10	5/80	7/80	7/304	X/100	5/209

For example, for an RP06 on an 11/70, *m* is 5 and *n* is 209. See *mkfs*(8) for more explanation of the values of *m* and *n*. An X entry means that we do not know the correct number for this combination of CPU and disk. If you do, please let us know. If *m* is unspecified or you have a disk which emulates a DEC disk, use the number for the most similar disk/CPU pair. **If *n* is unspecified, use the cylinder size divided by 2.**

Then run a standalone version of the *mkfs* (8) program. In the following procedure, substitute the correct types for *tp* and *dk* and the size determined above for *size*:

```
: tp(0,3)
Mkfs
file system: dk(0,0)                          (root is the first file system on drive 0)
file system size: size                        (count of 1024 byte blocks in root)
interleaving factor (m, 5 default): m         (interleaving, see above)
interleaving modulus (n, 10 default): n       (interleaving, see above)
isize = XX                                    (count of inodes in root file system)
m/n = m n                                     (interleave parameters)
Exit called
nnBoot
:                                             (back at tape boot level)
```

You now have an empty UNIX root file system.

2.3. Restoring the root file system

To restore a small root file system onto it, type

```
: tp(0,4)
Restor
Tape? tp(0,6)                                 (unit 0, seventh tape file)
Disk? dk(0,0)                                 (into root file system)
Last chance before scribbling on disk.        (just hit return)
    (30 second pause then tape should move)
    (tape moves for a few minutes)
end of tape
Exit called
nnBoot
:                                             (back at tape boot level)
```

If you wish, you may use the *icheck* program on the tape, *tp*(0,5), to check the consistency of the file system you have just installed.

2.4. Booting UNIX

You are now ready to boot from disk. It is best to read the rest of this section first, since some systems must be patched while booting. Then type:

```
:dk(0,0)dkunix            (bring in dkunix off root system)
```

The standalone boot program should then read *dk*unix from the root file system you just created, and the system should boot:

```
Berkeley UNIX (Rev. 2.9.1) Sun Nov 20 14:55:50 PST 1983
mem = xxx

CONFIGURE SYSTEM:
(Information about various devices will print;
most of them will probably not be found until
the addresses are set below.)
erase=^?, kill=^U, intr=^C
#
```

If you are booting from an *xp* with a drive type that is not recognized, it will be necessary to patch the system before it first accesses the root file system. Halt the processor after it has begun printing the version string but before it has finished printing the "mem = xxx" string. Place the

drive type number corresponding to your drive at location 061472; the addresses for drives 1, 2 and 3 are 061506, 061522 and 061536 respectively. If you plan to use any drives other than 0 before you recompile the system, you should patch these locations. Make the patches and continue the CPU. The value before patching must be zero. If it is not, you have halted too late and should try again.

UNIX begins by printing out a banner identifying the version of the system that is in use and the date it was compiled. Note that this version is different from the system release number, and applies only to the operating system kernel.

Next the *mem* message gives the amount of memory (in bytes) available to user programs. On an 11/23 with no clock control register, a message "No clock???" will print next; this is a reminder to turn on the clock switch if it is not already on, since UNIX cannot enable the clock itself. The information about different devices being attached or not being found is produced by the *autoconfig*(8) program. Most of this is not important for the moment, but later the device table can be edited to correspond to your hardware. However, the tape drive of the correct type should have been detected and attached.

The "erase=..." message is part of /.profile that was executed by the root shell when it started. The file /.profile contained commands to set the UNIX erase, line kill and interrupt characters to be what is standard on DEC systems so that it is consistent with the DEC console interface characters. This is not normal for UNIX, but is convenient when working on a hardcopy console; change it if you like.

UNIX is now running, and the Berkeley PDP-11 UNIX Programmer's manual applies. The '#' is the prompt from the Shell, and lets you know that you are the super-user, whose login name is "root."

There are a number of copies of *unix* on the root file system, one for each possible type of root file system device. All but one of them (*xpunix*) has had its symbol table removed (i.e. they have been "stripped"; see *strip*(1)). The unstripped copy is linked (see *ln*(1)) to */unix* to provide a system namelist for programs like *ps*(1) and *autoconfig*(8). All of the systems were created from */unix* by the C shell script */genallsys.sh*. If you had to patch the *xp* type as you booted, you may want to use *adb* (see *adb*(1)) to make the same patch in a copy of *xpunix*. If you are short of space, you can patch a copy of */unix* instead (setting the rootdev, etc.) and install it as */unix* after verifying that it works. See */genallsys.sh* for examples of using *adb* to patch the system. The system load images for other disk types can be removed. **Do not remove or replace the copy of */unix*, however, unless you have made a working copy of it that is patched for your file system configuration and still has a symbol table.** Many programs use the symbol table of */unix* in order to determine the locations of things in memory, therefore */unix* should always be an unstripped file corresponding to the current system. If at all possible, you should save the original UNIX binaries for your disk configuration (*dk*unix and unix) for use in an emergency.

There are a few minor details that should be attended to now. The system date is initially set from the root file system, and should be reset. The root password should also be set:

# date *yymmddhhmm*	(set date, see *date*(1))
# passwd root	(set password for super-user)
New password:	(password will not echo)
Retype new password:	

2.5. Installing the disk bootstrap

The disk with the new root file system on it will not be bootable directly until the block 0 bootstrap program for the disk has been installed. There are copies of the bootstraps in /mdec. This is not the usual location for the bootstraps (that is /usr/src/sys/mdec), but it is convenient to be able to install the boot block now. Use *dd*(1) to copy the right boot block onto the disk;

the first form of the command is for small disks (**rk**, **rl**) and the second form for disks with multiple partitions (**hk**, **rp**, **xp**), substituting as usual for *dk*:

dd if=*dk*uboot of=/dev/r*dk*0 count=1

or

dd if=*dk*uboot of=/dev/r*dk*0a count=1

will install the bootstrap in block 0. Once this is done, booting from this disk will load and execute the block 0 bootstrap, which will in turn load /boot (actually, the boot program on the first file system, which is root). The console will print

>boot	(printed by the block 0 boot)
*nn***Boot**	(printed by /boot)
:	

The '>' is the prompt from the first bootstrap. It automatically boots */boot* for you; if */boot* is not found, it will prompt again and allow another name to be tried. It is a very small and simple program, however, and can only boot the second-stage boot from the first file system. Once /boot is running and prints its ": " prompt, boot unix as above, using *dk*unix or unix as appropriate.

2.6. Checking the root file system

Before continuing, check the integrity of the root file system by giving the command

fsck /dev/r*dk*0a

(omit the **a** for an RK05 or RL). The output from *fsck* should look something like:

/dev/r*xx*0a
File System: /

**** Checking /dev/r*xx*0a**
**** Phase 1 - Check Blocks and Sizes**
**** Phase 2 - Check Pathnames**
**** Phase 3 - Check Connectivity**
**** Phase 4 - Check Reference Counts**
**** Phase 5 - Check Free List**
238 files 1916 blocks xxxxx free

If there are inconsistencies in the file system, you may be prompted to apply corrective action; see the document describing *fsck* for information. The number of free blocks will vary depending on the disk you are using for your root file system.

3. DEVICE AND FILE SYSTEM CONFIGURATION

This section will describe ways in which the file systems can be set up for the disks available. It will then describe the files and directories that will be set up for the local configuration. These are the */dev* directory, with special files for each peripheral device, and the tables in */etc* that contain configuration-dependent data. Some of these files should be edited after reading this section, and others can wait until later if you choose. The disk configuration should be chosen before the rest of the distribution tape is read onto disk to minimize the work of reconfiguration.

3.1. Disk configuration

This section describes how to lay out file systems to make use of the available space and to balance disk load for better system performance. The steps described in this section (3.1) are optional.

3.1.1. Disk naming and divisions

Each large physical disk drive can be divided into up to 8 partitions; UNIX typically uses only 3 to 5 partitions. For instance, on an RM03 the first partition, rm0a, is used for a root file system, a backup thereof, or a small file system like /tmp; the second partition, rm0b, is used for swapping or a small file system; and the third partition, rm0c, holds a user file system. Many disks can be divided in different ways; for example, the third section (**c**) of the RM03 could instead be divided into two file systems, using the rm0d and rm0e partitions instead, perhaps holding /usr and the user's files. The disk partition tables are specified in the *ioconf.c* file for each system, and may be changed if necessary. The last partition (**h**) always describes the entire disk, and can be used for disk-to-disk copies.

Warning: for disks on which DEC standard 144 bad sector forwarding is supported, the last track and up to 126 preceeding sectors contain replacement sectors and bad sector lists. Disk-to-disk copies should be careful to avoid overwriting this information. See *bad144*(8). Bad sector forwarding is optional in the **hk**, **hp**, **rm**, and **xp** drivers. It has been only lightly tested in the latter three cases.

3.1.2. Space available

The space available on a disk varies per device. The amount of space available on the common disk partitions for /usr is listed in the following table. Not shown in the table are the partitions of each drive devoted to the root file system and the swapping area.

Type	Name	Size
RK06	hk?d	9.2 Mb
RK07	hk?c	22.4 Mb
RM02, RM03	rm?c	60.2 Mb
RM02, RM03	rm?d	30.9 Mb
RP03	rp?c	33.3 Mb
RP04, RP05, RP06	hp?c	74.9 Mb
RP06	hp?d	158.9 Mb
RM05	xp?c	115.4 Mb
RM05	xp?e	80.9 Mb

Each disk also has a swapping area and a root file system. The distributed system binaries and sources occupy about 38 megabytes.

The sizes and offsets of all of the disk partitions are in the manual pages for the disks; see section 4 of the Berkeley PDP-11 UNIX Programmer's manual. Be aware that the disks have their sizes measured in "sectors" of 512 bytes each, while the UNIX file system blocks are 1024 bytes

each. Thus if a disk partition has 10000 sectors (disk blocks), it will have only 5000 UNIX file system blocks, and you **must** divide by 2 to use 5000 when specifying the size to the *mkfs* command. The sizes and offsets in the kernel (ioconf.c) and the manual pages are in 512-byte blocks. If bad sector forwarding is supported for your disk, be sure to leave sufficient room to contain the bad sector information when making new file systems.

3.1.3. Layout considerations

There are several considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. Swapping space is an important parameter. Since running out of swap space often causes the system to panic, it must be large enough that this does not happen.

Many common system programs (the C compiler, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks; if you have several disks, it makes sense to mount this in a "root" or "swap" (i.e. first or second partition) file system on another disk. On RK06 and RK07 systems, where there is little space in the hk?c or hk?d file systems to store the system source, it is normal to mount /tmp on /dev/hk1a.

The efficiency with which UNIX is able to use the CPU is often strongly affected by the configuration of disks. For general time-sharing applications, the best strategy is to try to split the most actively-used sections among several disk arms. There are at least five components of the disk load that you can divide between the available disks:

1. The root file system.
2. The swap area.
3. The /tmp file system.
4. The /usr file system.
5. The user files.

Here are several possibilities for utilizing 2, 3 and 4 disks:

	disks		
what	2	3	4
root	1	1	1
tmp	1	3	4
usr	1	2	2
swapping	2	3	4
users	2	1+3	1+3
archive	x	x	4

The most important consideration is to even out the disk load as much as possible, and to do this by decoupling file systems (on separate arms) between which heavy copying occurs. Note that a long term average balanced load is not important; it is much more important to have instantaneously balanced load when the system is busy. When placing several busy file systems on the same disk, it is helpful to group them together to minimize arm movement, with less active file systems off to the side.

Intelligent experimentation with a few file system arrangements can pay off in much improved performance. It is particularly easy to move the root, the /tmp file system and the swapping areas. Note, though, that the disks containing the root and swapping area can never be removed while UNIX is running. Place the user files and the /usr directory as space needs dictate and experiment with the other, more easily moved file systems.

As an example, consider a system with RM03s. On the first RM03, **rm0**, we will put the root file system in **rm0a**, and the **/usr** file system in **rm0c**, which has enough space to hold it and then some. If we had only one RM03, we would put user files in the **rm0c** partition with the

system source and binaries, or split them between **rm0d** and **rm0e**. The /tmp directory will be part of the root file system, as no file system will be mounted on /tmp.

If we had a second RM03, we would create a file system in **rm1c** and put user files there, calling the file system /mnt. We would keep a backup copy of the root file system in the **rm1a** disk partition, a file system for /tmp on **rm0b**, and swap on **rm1b**.

3.1.4. Implementing a layout

Once a disk layout has been chosen, the appropriate special files for the disk partitions must be created (see Setting up the /dev directory, below). Empty file systems will then be created in the appropriate partitions with *mkfs*(8), and the files belonging in the file system can then be restored from tape. The section on setting up the /usr file system contains detailed information on this process. The swap device is specified when the kernel is configured, which is also discussed later. At that time, you may also want to consider whether to use the root device or another file system (e.g. /tmp) for the pipe device (the pipe device is a file system where the kernel keeps temporary files related to pipe I/O; it should be *mount*ed before any I/O through pipes is attempted).

3.2. Setting up the /dev directory

Devices are accessed through special files in the file system, made by the *mknod*(8) program and normally kept in the /dev directory. Devices to be supported by UNIX are implemented in the kernel by drivers; the proper driver is selected by the major device number and type specified to *mknod*. All devices supported by the distribution system already have nodes in /dev. They were created by the /dev/MAKE shell script. It is easiest to rebuild this directory from the beginning with the correct devices for your configuration. First, determine the UNIX names of the devices on your system (e.g. *dh, lp, xp*). Some will be the same as the names of devices on the generic system. Others need not be. See section 4 of the UNIX Programmer's Manual. Next create a new directory /newdev, copy /dev/MAKE into it, edit MAKE to provide an entry for local needs, replacing the case LOCAL, and run it to generate the desired devices in the /newdev directory. The LOCAL entry can be used for any unusual devices, and to rename standard devices as desired. It should also move the node for the disk partition being used as the swap area to *swap* (or, if swap is after a file system as on RK05 or RL disks, link the other node to *swap*). Different devices are specified to MAKE in various ways. Terminal multiplexors (DZ and DH) are specified by boards, and 8 or 16 nodes will be made, as appropriate. Disks are made by partition, for example xp0c, so that you may make the nodes corresponding to the file systems that you intend to use. Note that *hp, rm* and *xp* are actually synonyms, but you should use the name corresponding to the driver you plan to use. The kernel configuration section (section 5.4.1) has more information. For tape drives, there are different invocations for different types of controllers, although the nodes produced will have the same names. The different types are *ht, tm* and *ts*, as above, and also *ut*, which is used for the Emulex TC-11 and other TM-11 emulations that are also capable of selecting 1600 or 800 bpi under software control. Making *ht0* or *ut0* will result in nodes *mt0* and *mt1* (800 and 1600 bpi, respectively) and parallel nodes for other options; *ht1* uses the names *mt2* and *mt3*. See *ht*(4) and *tm*(4). In contrast, the MAKE script makes only one set of nodes for *tm* or *ts*, without changing the unit number specified. Different sites use different naming conventions for tapes; you could use the LOCAL entry in MAKE to move the tape files to your favorite names.

As an example, if your machine had a single DZ-11, two DH-11s, an RP03 disk, two RP06 disks, and a TM03 tape formatter you would do:

```
# cd /
# mkdir newdev
# cp /dev/MAKE /newdev/MAKE
# cd newdev
# ./MAKE dz0 dh1 ht0 std LOCAL
# ./MAKE rp0a rp0b rp0c hp0a hp0b hp0c hp1a hp1b hp1d hp1e
```

Note the "std" argument here that causes standard devices such as *console*, the console terminal, to be created.

You can then do

```
# cd /
# mv dev genericdev ; mv newdev dev
# sync
```

to install the new device directory. Once you are confident that the new directory is set up properly, you can remove /genericdev.

3.3. Editing system-dependent configuration files

There are a number of small files in */etc* that are used by various programs to determine things about the local configuration. At this point, several of these should be edited to describe the local configuration. You may have old versions of some of them which you may want to use, or you may edit the files that are provided as examples. Some of this may be done later at your convenience, but is presented here for organization. Both **/etc/dtab** and **/etc/fstab** should be edited now.

3.3.1. /etc/dtab

This file contains the list of devices which will be checked at boot time by *autoconfig*(8). The devices that are listed are tested to see whether they exist and have the correct register addresses and interrupt vectors. If they do, and the kernel has a corresponding driver routine, *autoconfig* notifies the driver that the device exists at that address. In this way, the addresses and vectors of most devices do not need to be compiled into the operating system. The exception is that disks must be preconfigured if they are to be used as root file systems.

This file should be edited to include all of the devices on the system with the exception of the clock and console device. Other device entries can be deleted or commented out with a '#' at the beginning of the line. The format of the entries is defined in *dtab*(5). *Autoconfig*(8) describes the autoconfiguration process. One word of caution: if a device fails to interrupt as expected, and if its unit number is specified (not a '?' wildcard), *autoconfig* will notify the driver that the device is **not** present, and preconfigured devices (like root disks) could be disconnected. Thus, it is probably best to use a '?' instead of a unit number for your root disks until you are confident that the probe always finds that disk, especially if your disk controller is an emulation of another disk type. Disks that are not used as boot devices for UNIX can be properly listed with unit numbers.

3.3.2. /etc/fstab

This file contains the list of file systems normally mounted on the system. Its format is defined in *fstab*(5). Programs like *df*(1) and *fsck*(8) use this list to control their actions. Each disk partition that has been assigned a function should be listed here. See the manual pages for specifics on how to configure this file.

3.3.3. /etc/ident

The banner printed by *getty*(8) is read from /etc/ident. Edit this file to the banner you wish to use. It may contain special characters to clear terminal screens, etc., but note that the same file is used for all terminals.

3.3.4. /etc/motd

The contents of /etc/motd, the "message of the day," is displayed at the terminal when a user is logged in by *login*(1).

3.3.5. /etc/passwd, /etc/group

These files obviously need local modifications. See the section on adding new users. Entries for pseudo-users (user IDs that are not used for logins) have password fields containing "***", since encrypted passwords never not contain asterisks.

3.3.6. /etc/rc

As the system begins multiuser operations, it executes the commands in /etc/rc (see *init*(8)). Most of the commands in this file are standard and should not be changed, including the section for checking file systems after a reboot. These commands will be ignored if autoreboot is not enabled. You should edit /etc/rc to set your machine's name. Look for the line

/etc/hostname hostnameunknown

and change *hostnameunknown* to the name of your machine. This name will be used by *Mail*(1) and *uucp*(1) (among others) and should correspond to the name by which your machine is known to external networks (if any). At this time you may wish to add additional commands to this file if you need to start additional daemons, remove old lock files, or perform any other cleanup as the system comes up.

3.3.7. Configuring terminals

If UNIX is to support simultaneous access from more than just the console terminal, the file /etc/ttys (*ttys*(5)) has to be edited.

Terminals connected via DZ interfaces are conventionally named **tty***dd* where *dd* is a decimal number, the "minor device" number. The lines on dz0 are named /dev/tty00, /dev/tty01, ... /dev/tty07. Lines on DH interfaces are conventionally named **ttyh***x*, where *x* is a hexadecimal digit. If more than one DH interface is present in a configuration, successive terminals would be named **ttyi***x*, **ttyj***x*, etc.

To add a new terminal be sure the device is configured into the system, that the special file for the device has been made by /dev/MAKE, and the special file exists. Then set the first character of the appropriate line of /etc/ttys to 1 (or add a new line). The first character may also be 3 if the line is also to be used in maintenance mode (see *init*(8)).

The second character of each line in the /etc/ttys file lists the speed and initial parameter settings for the terminal. The most common choices, from *getty*(8), are:

0	300-1200-150-110
3	1200-300
4	300 (e.g. console)
5	300-1200
6	1200
7	2400
8	4800
9	9600
B	autobaud

Here the first speed is the speed a terminal starts at, and "break" switches speeds. Thus a newly added terminal /dev/tty00 could be added as

19tty00

if it was wired to run at 9600 baud. The "B" indicates that *getty* should attempt to guess a line's speed when the user types a carriage return or control-C. Note that this requires kernel support.

See section 5.3.6 below.

Dialup terminals should be wired so that the carrier is asserted only when the phone line is dialed up. For non-dialup terminals from which modem control is not available, you must either wire back the signals so that the carrier always appears to be present, or (for lines on a DH-11 or DZ-11) add 0200 to the minor device number to indicate that carrier is to be ignored. See *dh*(4) and *dz*(4) for details.

You should also edit the file /etc/ttytype placing the type of each terminal there (see *ttytype*(5)).

When the system starts running multi-user, all terminals that are listed in /etc/ttys having a 1 or 3 as the first character of their line are enabled. If, during normal operations, it is desired to disable a terminal line, the super-user can edit the file /etc/ttys, change the first character of the corresponding line to 0 and then send a hangup signal to the *init* process, by typing (see *kill*(1))

```
# kill -1 1
  or
# kill -HUP 1
```

Terminals can similarly be enabled by changing the first character of a line from a 0 to a 1 and sending a hangup to *init*.

Note that if a special file is inaccessible when *init* tries to create a process for it, init will print a message on the console and try to reopen the terminal every minute, reprinting the warning message every 10 minutes.

Finally note that you should change the names of any dialup terminals to ttyd? where ? is in [0-9a-f] since some programs use this property of the names to decide whether a terminal is a dialup. Shell commands to do this should be put in the /dev/MAKE script under case LOCAL.

4. SETTING UP THE /usr FILE SYSTEM

The next step in bringing up the 2.9BSD distribution is to read in the binaries and sources on the /usr file system. This will also demonstrate how to add new file systems in general, and the overall procedure can be repeated to set up additional file systems. There are two portions of the /usr file system, one on each tape. The first tape contains the binary directories, manual pages and documentation, as well as skeletal directories such as spool and msgs. If you have room, it is easiest to extract everything. The size of the entire /usr file system image on the distribution tapes is 38 megabytes. It will not fit on a single RK05, RK06/7 or RL01/2. In these cases, the /usr file system will have to be extracted in sections or split across multiple disks. The *bin, include, lib,* and *ucb* subdirectories are essential. The system sources will also be needed to reconfigure the kernel; they are in /usr/src/sys. The *adm, dict, msgs, preserve, spool, sys* and *tmp* directories may also be extracted to provide a skeletal system. The first part of this section describes how to extract /usr as part of a full bootstrap; the second part explains how to install 2.9BSD as an upgrade to a 2.8BSD system if you decide not to perform a full bootstrap.

4.1. Full bootstrap procedure

This procedure will create a new file system and extract the /usr directory into it. First determine the name of the disk on which you plan to place the new file system, for example rm0c, and substitute it for *disk* throughout this section. You may want to create a small "prototype" file to describe the file system (see *mkfs*(8)) in order to change the size of the inode list. This is the same as the maximum number of files that can be created on the file system. The default is to allow 16 inodes (occupying one block) per 24 file system blocks, allowing the file system to be completely filled with small files (1-2 blocks). This is more than required for /usr and other file systems which have larger average file size. If you decide to set up a prototype file for *mkfs*, use its name for *proto* below. The prototype file needs to contain only the name of the bootstrap, the sizes, and the line for the root directory (don't forget the '$' to terminate). Look up the correct size for this file system in the manual section for the disk. Note that the size given to *mkfs* is in file system blocks of 1024 bytes, and thus the sizes in the manual page will have to be divided by 2. If not using a prototype file, substitute the size for *proto* in the mkfs command below. Finally, recall the interleaving parameters *m* and *n* that you used in making the root file system. They are in the table in section 2.2. Comments are enclosed in ( ); don't type these. Then execute the following commands (substituting *rmt1* and *nrmt1* for *rmt0* and *nrmt0* respectively if you have a 1600 bpi tape on an ht or tm controller):

```
# /etc/mkfs /dev/rdisk proto m n   (create empty user file system)
isize = nnnnn                      (the count of available inodes)
m/n = m n                          (free list interleave parameters)
                                   (this takes a few minutes)
# /etc/mount /dev/disk /usr        (mount the usr file system)
# cd /usr                          (make /usr the current directory)
                                   (make sure that the first tape is mounted)
# mt -t /dev/nrmt0 fsf 7           (skip first seven tape files)
# tar xpf /dev/rmt0                (extract the /usr file system binaries)
                                   (this takes about 20 minutes)
                                   (now mount the second tape)
# tar xpf /dev/rmt0                (extract the /usr file system sources)
                                   (this takes another 20 minutes)
```

You can now check the consistency of the /usr file system by doing

```
# cd /                        (back to root)
# /etc/umount /dev/disk       (unmount /usr)
# fsck /dev/rdisk
```

To use the /usr file system, you should now remount it by saying

```
# /etc/mount /dev/disk /usr
```

If you are installing the distribution on a PDP11/44, 11/45, 11/70, 11/73, 11/83, or 11/84 (machines with separate instruction and data space) you should test and install the separate I/D versions of csh, ex, etc. in /usr/70. Note, however, that these binaries assume the existence of hardware floating point support.

4.2. Bootstrap path 2: upgrading 2.8BSD

Begin by reading the other parts of this document to see what has changed since the last time you bootstrapped the system. Also look at the new manual sections provided to you. If you have local system modifications to the kernel to install, look at the document "Changes in the Kernel in 2.9BSD" to get an idea of how the system changes will affect your local mods. Disclaimer: there are a very large number of changes from 2.8BSD to 2.9. This section may not be complete, and if a new program fails to work after being recompiled, you may find that additional libraries or other components may also need to be updated.

There are 6 major areas of changes that you will need to incorporate to convert to the new system:

1. The new kernel and the associated programs that implement job control or read kernel memory: autoconfig, csh, the jobs library, login, ps, pstat, w, etc.
2. The programs related to system reboots and shutdowns.
3. The programs directly related to user text overlays: adb and ld.
4 The C compiler driver, C preprocessor, and assembler.
5 The new version of the standard I/O library.
6. Other programs with significant bug fixes, significant improvements, or which were previously unavailable because they had not been overlaid.

Here is a step-by-step guide to converting. Before you begin you should do a full backup of your root and /usr file systems as a precaution against irreversible mistakes.

1. Set the shell variable "nbsd" to the name of a directory where an empty file system can be mounted and a quantity of material from the tape (you should allow for about 38 megabytes) can be extracted. Choose a disk of sufficient size to hold this quantity of material, make a file system, and mount $nbsd on this disk. Next, restore (see *restor*(8)) the root file system dump image to this disk. Finally, change directory to "$nbsd/usr", and extract the eighth file from the first distribution tape and all of the second tape using *tar* (see *tar*(1)).
2. Install the new include files by copying $nbsd/usr/include/*.h to /usr/include and $nbsd/usr/include/sys/*.h to /usr/include/sys. Install the C compiler driver from the new system by copying $nbsd/bin/cc to /bin/cc. Install the assembler from the new system by copying $nbsd/bin/as to /bin/as and $nbsd/lib/as2 to /lib/as2. Install the new C preprocessor by copying $nbsd/lib/cpp to /lib/cpp. Install the new versions of adb and ld by copying $nbsd/bin/adb and $nbsd/bin/ld to /bin.
3. Reconfigure the system in $nbsd/usr/src/sys to correspond to your configuration according to the instructions in section 5.
4. Put in the new versions of the following programs:

   /bin: csh, kill, login, iostat, ps, pstat, vmstat

/etc: autoconfig, fsck, init, mount, reboot, savecore, shutdown, umount

/usr/ucb: ex, w

Merge any local changes to /etc/rc into $nbsd/etc/rc. Put the resulting file in /etc/rc. Create the directory /usr/sys and perhaps some files in this directory (read *savecore*(8)). Make a device description file for *autoconfig*. See *dtab*(5) and *autoconfig*(8).

5. Try bootstrapping the new system; it should now work. Make sure to write new instructions to your operators.

6. Incorporate some other important bug fixes or enhancements:

   a) Replace the file tmac.an in the directory /usr/lib/tmac with the version from $nbsd/usr/lib/tmac. Replace the file /usr/lib/me/local.me with the version from $nbsd/usr/lib/me; copy $nbsd/usr/lib/me/refs.me to /usr/lib/me.

   b) Install the new C library source, /usr/src/lib/c, rebuild and reinstall /lib/libc.a and /usr/lib/libovc.a.

   c) Install the jobs library, /usr/src/lib/jobs and build and install /usr/lib/libjobs.a and /usr/lib/libovjobs.a.

   d) Replace the directory /usr/src/cmd/refer. Then rebuild and reinstall the programs.

   e) Install the new Mail source, /usr/src/ucb/Mail and reinstall /usr/ucb/Mail.

   f) If the target machine is a nonseparate I/D CPU, install the new *lex* and *yacc* directories, compile and install the programs.

   g) Install the new version of *tar* from $nbsd/usr/src/cmd/tar.c and also the program *mt* from $nbsd/usr/src/ucb/mt.c.

   h) Merge your changes to /usr/src/ucb/termcap/reorder and reinstall the terminal data base, /etc/termcap. Install the new terminal library, /usr/src/ucb/termlib, remake and reinstall /usr/lib/libtermcap.a and /usr/lib/libovtermcap.a. Then make and install the new version of *ex*.

   i) If you want the new version of the Pascal system incorporating overlays (for non-separate I/D CPUs), remake the directories *pi* and *px* in $nbsd/usr/src/cmd and install the programs.

   j) Install the new F77 compiler, /usr/src/cmd/f77, and the new libraries, /usr/src/lib/lib*77. Then remake and reinstall them.

   k) Install the new library sources, /usr/src/lib/{ape,curses,m,mp,plot} and remake and reinstall the new libraries.

   l) Install new versions of as many of the following programs as you choose: 512dumpdir, 512restor, atrun, cat, catman, ccat, compact, checkobj, ctags, df, diff, du, egrep, error, expand, fgrep, find, from, grep, hostname, jove, l11, lint, ln, lock, login, lpr, ls, m11, make, man, mkfs, more, msgs, mv, ncheck, printenv, pq, ranm, rewind, rm, rmdir, sed, setquota, size, sort, split, sq, strings, strip, stty, sysline, tail, tbl, tset, ul, uncompact, unexpand, vsh, wc.

   m) Install the modified or new administrative programs: ac, getty, last.

   n) Install some security fixes in the mail systems by installing new sources for berknet (/usr/src/ucb/berknet), delivermail (/usr/src/ucb/delivermail), mail (/usr/src/cmd/mail.c), and secret mail (/usr/src/cmd/xsend), and remaking and reinstalling the new binaries.

   o) Install the new version of uucp (/usr/src/cmd/uucp).

   p) Install the news (/usr/contrib/news) or notes (/usr/contrib/notes) bulletin board system if you wish.

   q) Install the new *eqn*(1) symbol macros, /usr/public/eqnSyms.

r) Install manual pages corresponding to the new and changed programs.

s) Remove the old programs /bin/ovas, /bin/ovld, /lib/ovas2, and /bin/ovadb. Remove the libucbpath library. Remove the old version of reset and link the new version of tset to reset.

5. CONFIGURING AND COMPILING THE KERNEL

This section describes procedures used to set up a PDP-11 UNIX kernel (operating system). It explains the layout of the kernel code, compile time options, how files for devices are made and drivers for the devices are configured into the system and how the kernel is rebuilt to include the needed drivers. Procedures described here are used when a system is first installed or when the system configuration changes. Procedures for normal system operation are described in the next section. We also suggest ways to organize local changes to the kernel.

5.1. Kernel organization

The kernel source is kept in the subdirectories of /usr/src/sys. The directory /usr/src/sys/sys contains the mainline kernel code implementing system calls, the file system, memory management, etc. The directory /usr/src/sys/dev contains device drivers and other low-level routines. The header files and scripts used to compile the kernel are kept in /usr/src/sys/conf, and are copied from there into a separate directory for each machine configuration. It is in this directory, /usr/src/sys/*machine*, that the kernel is compiled.

5.2. Configuring a System

The kernel configuration of each PDP-11 UNIX system is described by a set of header files (one for each device driver) and one file of magic numbers (ioconf.c) stored in a subdirectory of /usr/src/sys for each configuration. Pick a name for your machine (call it PICKLE). Then in the /usr/src/sys/conf directory, create a configuration file PICKLE describing the system you wish to build, using the format in *config*(8). This is most easily done by making a copy of the GENERIC file used for the distributed UNIX binary. Many of the fields in the configuration file correspond to parameters listed in the remainder of this section, which should be scanned before proceeding. See especially section 5.4.3 on how to set up automatic reboots and dumps. Then use *config* to create a system directory ../PICKLE with "config PICKLE." Note the difference between *config* and *autoconfig*. *Config* sets up a directory in which the kernel will be compiled, with all of the system-specific files used in compilation, and specifies what devices will potentially be supported. *Autoconfig* adapts the running kernel to the hardware actually present, by testing and setting the register addresses and interrupt vectors.

Config does most of the work of configuration, but local needs will dictate some changes in the options and parameters in the header files. All of the options are listed in the next section. Examine whoami.h, localopts.h, param.h, and param.c and make any changes required; it might also be wise to look through the header files for the devices that you have configured, to check any options specific to the device drivers that are listed there. After you have finished configuring a kernel and tested it, you should install whoami.h in /usr/include, and copy localopts.h and param.h into /usr/include/sys. This will allow user-level programs to stay in sync with the running kernel.

If you wish to change any disk partition tables or device control status register addresses (other than those configured at boot time by *autoconfig*(8)), edit ioconf.c and change the appropriate line(s). The file l.s contains the interrupt vectors and interface code and may also be edited if necessary, but usually will require no change. Both c.c and l.s include support for all normal devices according to the header files per device, and with autoconfiguration, the actual vectors need not be specified in advance. Finally, examine the Makefile, especially the options near the top and the load rules. If you have placed the include files in the standard directories, you shouldn't have to make any changes to the options there.

The following sections give short descriptions of the various compile-time options for the kernel, and more extensive information on the autoreboot and disk monitoring setup. After verifying that those features are configured correctly for your system, you can proceed to kernel compilation.

5.3. Compile Time Options

The 2.9BSD kernel is highly tunable. This section gives a brief description of the many compile-time options available, and references to sections of the Berkeley PDP-11 UNIX Programmer's manual where more information can be found. Options fall into four categories; the letters following each will be used to mark the options throughout the rest of this section.

Standard (S) — These include options which we consider necessary for reasonable system performance or resiliency.

Desirable (D) — These include many other features that are convenient but which may be turned off if system size is critical. The user programs and libraries distributed with 2.9BSD generally assume that these are turned on, so turning them off may necessitate recompiling libraries or programs. These options, along with those designated "standard," have received the most thorough testing.

Configuration Dependent (C) — Options that depend on such things as the physical configuration or speed issues fall into this category.

Experimental (X) — New features that have not been well tested, options that have known problems, or ones that we do not normally use are listed as experimental. You should not use such options unless the problems listed are not considerations for your system, or you are willing to watch things closely and possibly do some debugging.

The following sections list the parameters and options used in the kernel. The parameters (section 5.3.2) have numeric values, usually table sizes, and most of them are in param.h or param.c. Those that are in param.h are typically not changed, with the possible exception of **MAXMEM**, as their values are set by convention. The option flags are either defined or undefined to enable or disable the corresponding feature, with the exception of **UCB_NKB**, which is unlikely to change. Each option is marked with a letter to indicate into which of the four categories above it falls.

5.3.1. Hardware

ENABLE34 — **C** Automatically detect and support Able Computer's ENABLE/34† memory management board. This option implies **UNIBUS_MAP**.

NONFP — **C** Do not compile in code to automatically detect and support an FP11 floating point processor. Also, include a fast illegal-instruction trap handler and modify the signal routines to make it possible to run programs using the floating-point interpreter under trace.

NONSEPARATE — **C** Do not attempt to support separate I/D user programs.

PARITY — **C** Recognize and deal with cache and memory parity traps.

PDP11 — **C** This should be set to the CPU type of the target machine (23, 24, 34, 40, 44, 45, 60, 70, 73, or GENERIC). You should use 34 for an 11/34A, 45 for an 11/55, and 73 for an 11/74. GENERIC should be used to build a system which runs on a variety of CPUs. It was used to make the distributed kernels. **MENLO_KOV** and **NONSEPARATE** are defined if **PDP11** is 23, 24, 34, 40, or 60. **MENLO_KOV** is also defined if **PDP11** is GENERIC. **UNIBUS_MAP** is defined if **PDP11** is 44, 70, 84, or GENERIC.

SMALL — **C** Use smaller (by about a factor of 8) queues and hash tables.

† ENABLE/34 is a trademark of Able Computer, Inc.

UNIBUS_MAP **C** Compile in code to detect (and support if present) a UNIBUS map.

5.3.2. Parameters

5.3.2.1. Global configuration

MAXUSERS This is the maximum number of users the system should normally expect to support. *Config* sets this from the corresponding field in the description file; the definition is copied into the system Makefile rather than a header file. It is not intended to be a hard limit. It is used in sizing other parameters (**CMAPSIZ**, **NFILE**, **NINODE**, **NPROC**, **NTEXT**, and **SMAPSIZ**). The formulae are found in *param.c*. Reasonable values for **MAXUSERS** might be 3 or 4 on a small system (11/34, 11/40), 15 for an 11/44 with a reasonable amount of memory, and 15-30 for an 11/70 system.

TIMEZONE The number of minutes westward from Greenwich. *Config* sets this from the corresponding field in the description file. Examples: for Pacific Standard time, 8 (* 60); for EST, 5.

DSTFLAG Should be 1 if daylight savings time applies in your locality and 0 otherwise. *Config* sets this from the field in the description file.

HZ This is the line clock frequency (e.g. 50 for a 50 Hz. clock).

5.3.2.2. Tunable parameters

CMAPSIZ This is the number of fragments into which memory can be broken. If this number is too low, the kernel's memory allocator may be forced to throw away a section of memory being freed because there is no room in the map to hold it. In this case, a diagnostic message is printed on the console. Normally scaled automatically according to **MAXUSERS**.

MAXMEM This sets an administrative limit on the amount of memory a process may have. It is specified as (*nn**16), where the first number is the desired value in kilobytes (the product is in clicks). This number is usually considerably lower than the theoretical maximum (304 Kb for a nonseparate I/D CPU, 464 Kb for a separate I/D CPU, assuming **MENLO_OVLY** is defined). Normal values are 128 Kb if there is no UNIBUS map (maximum physical memory 248 Kb), otherwise 200 Kb.

NBUF This sets the size of the system buffer cache. It can be no greater than 248. If **UCB_NKB** is defined, these are 1024 byte buffers. Otherwise, they are 512 byte buffers. The buffers are not in kernel data space, but are allocated at boot time. Normally scaled automatically according to **MAXUSERS**, but should be examined in the light of the disk load and amount of memory. For a small to medium system, around 20 buffers should be sufficient; a large system with many disks might use 40 to 60 or more.

NCALL This is the maximum number of simultaneous callouts (kernel event timers). Callouts are used to time events such as tab or carriage return delays. Normally scaled automatically according to **MAXUSERS**.

NCLIST This is the maximum number of clist segments. Clists are small buffer areas, used to hold tty characters while they are being processed. If **UCB_CLIST** is defined, they are not in kernel data space, and this number must be less than 512 if you are using 14

character clists (the default), or 256 for 30 character clists. (The clist size, **CBSIZE**, is in param.h.)

NDISK — This is the maximum number of disks and controllers for which I/O statistics can be gathered. See *iostat*(8). Care must be taken that this is large enough for the parameters for each disk (*XX*_DKN and number of disks; see the section on disk monitoring).

NFILE — This sets the maximum number of open files. An entry is made in this table each time a file is "opened" (see *creat*(2)), *open*(2)). Processes share these table entries across forks (see *fork*(2), *vfork*(2)). Normally scaled automatically according to **MAXUSERS**.

NINODE — This sets the size of the inode table. There is one entry in the inode table for each open file or device, current working or root directory, saved text segment, active quota node (if **UCB_QUOTAS** is defined), and mounted file system. Normally scaled automatically according to **MAXUSERS**.

NMOUNT — This indicates the maximum number of mountable file systems. It should be large enough that you don't run out at inconvenient times.

NPROC — This sets the maximum number of active processes. Normally scaled automatically according to **MAXUSERS**.

NTEXT — This sets the maximum number of active shared text images (including inactive saved text segments). Normally scaled automatically according to **MAXUSERS**.

SMAPSIZ — This is the analogy of **CMAPSIZ** for secondary memory (swap space). Normally scaled automatically according to **MAXUSERS**.

5.3.2.3. Parameters that are set by convention

CANBSIZ — This sets the maximum size of a terminal line input buffer. If using the old tty line discipline, exceeding this bound causes *all* characters to be lost. In the new tty line discipline, no more characters are accepted until there is room. Normally 256.

MAXSLP — This is the maximum time a process can sleep before it is no longer considered a "short term sleeper." It is used only if **UCB_METER** is defined. Normally 20.

MAXUPRC — This sets the maximum number of processes each user is allowed. Normally 20, but can be lower on heavily loaded systems.

MSGBUFS — This is the number of characters saved from system error messages. It is actually the size of circular buffer into which messages are temporarily saved. It is expected that *dmesg*(8) will be run by *cron*(8) frequently enough that no message is overwritten before it can be saved in the system error log. Normally 128.

NCARGS — This is the maximum size of an *exec*(2) argument list (in bytes). Normally 5120.

NOFILE — This sets the maximum number of open files each process is allowed. Normally 20.

SINCR — The increment (in clicks) by which a process's stack is expanded when a stack overflow segmentation fault occurs. Normally 20.

SSIZE The initial size (in clicks) of a process's stack. This should be made larger if commonly run processes have large data areas on their stacks. Normally 20.

5.3.3. General Options

ACCT D Enable code which (optionally) writes an accounting record for each process at exit. See *lastcomm*(1), *sa*(1), *acct*(2), *accton*(8).

CGL_RTP C Support a system call which marks a process as a "real time" process, giving it higher priority than all others. See *rtp*(2).

DIAGNOSTIC C Turn on more stringent error checking. This enables various kernel consistency checks which are considered extremely unlikely to fail. It is useful when the system is inexplicably crashing.

INSECURE C Do not turn off the set-user-id or set-group-id permissions on a file when it is written.

MENLO_JCL D Support reliable signal handling and enhanced process control features. See *sigsys*(2j), *jobs*(3j), *sigset*(3j). This option requires **UCB_NTTY**.

MENLO_KOV C Support automatic kernel text overlays. This is required for nonseparate I/D systems and is defined automatically if **PDP11** is defined to be 23, 24, 34, 40, 60, or GENERIC.

MENLO_OVLY D Support automatic user text overlays. This is required in order to run certain programs (e.g. *ex* version 3.7 or, on nonseparate I/D systems, the process control C shell).

OLDTTY C Support the standard V7 tty line discipline (see *tty*(4)). This must be defined if **UCB_NTTY** is not defined.

UCB_AUTOBOOT D Allows the kernel to automatically reboot itself, either on demand (see *reboot*(2) and *reboot*(8)) or after *panics*. This option requires a little planning; see section 5.4.3. **This option requires UCB_FSFIX.**

UCB_CLIST C Map clists out of kernel virtual data space. If there is sufficient space in kernel data for an adequate number of clists, this option should not used. Mostly used on large systems, or on systems where kernel data space is tight.

UCB_GRPMAST C Allow one user to be designated a "group super-user," able to perform various functions previously restricted to root or the file's owner alone. In the kernel, users whose group and user ids are the same are granted the same permissions with respect to files in the same group as is the owner. User level software implements other permissions, allowing the group super-user to change the password of a user in the same group. The most common use for this is in allowing teaching assistants to oversee students.

UCB_NET X Enable code implementing a PDP-11 port of Berkeley's version of TCP/IP. The code is experimental and the implementation is incomplete.

UCB_NTTY S Support the Berkeley tty line discipline (see *tty*(4) and *newtty*(4)). This must be defined if **OLDTTY** is not defined.

UCB_PGRP C Fix a bug in the way standard V7 counts a user's processes. This should be enabled only if **MENLO_JCL** is undefined, since the notion of process groups is completely different in the two cases. If **UCB_PGRP** and **MENLO_JCL** are both defined, the limit on

the number of processes allowed per user (**MAXUPRC**) is effectively eliminated.

UCB_SCRIPT **X** Allow scripts to specify their own interpreters. For example, executing a script beginning with "#! /bin/sh" causes /bin/sh to be executed to interpret the script. This is not the same as the facility on 4.1BSD VMUNIX, and probably needs a little work. The Bourne shell, /bin/sh, would need modification also.

UCB_UPRINTF **D** Write error messages directly on a user's terminal when the user causes a file system to run out of inodes or free blocks, or on certain mag tape errors.

UCB_VHANGUP **D** Support a system call which allows *init*(8) to revoke access to a user's terminal when the user has logged out. This is used to give new users "clean" terminals on login.

VIRUS_VFORK **D** Implement a much more efficient version of fork in which parent and child share resources until the child *execs*. See *vfork*(2). Note that this changes the way processes appear in memory. It makes swap operations slower, and thus might not be desirable on systems which swap heavily.

5.3.4. File system

INTRLVE **X** Allows interleaving of file systems across devices. See *intrlve*(4).

MPX_FILS **X** Include code for the V7 multiplexer. The code is buggy and unsupported.

UCB_FSFIX **S** Ensure that file system updates are done in the correct order, thus making damaged file systems less likely and more easily repairable. **This option is required by UCB_AUTOBOOT (actually, by the −p option of *fsck*(8), which makes certain assumptions about the state of the file systems).**

UCB_SYMLINKS **C** Add a new inode type to the file system: the symbolic link. Symbolic links cause string substitution during the pathname interpretation process. See *ln*(1), *readlink*(2), and *symlink*(2).

UCB_NKB **S** Use file system blocks of *N* KB, normally 1. Changes the fundamental file system unit from 512 byte blocks to 1024 byte blocks (with a corresponding reduction in the size of in-core inodes). This increases file system bandwidth by 100%. Note that **UCB_NKB** is not boolean, but is defined as 1 for 1KB blocks. Other values are possible, but require additional macro definitions. All file systems would have to be remade with new versions of *mkfs* and *restor*. **All supplied software expects UCB_NKB to be defined and equal to 1.**

UCB_QUOTAS **C** Support a simplistic (and easily defeated) dynamic disk quota scheme. See *ls*(1), *pq*(1), *quota*(2), and *setquota*(8).

5.3.5. Performance Monitoring

DISKMON **C** Keep statistics on the buffer cache. They are printed by the *−b* option of *iostat*(8).

UCB_LOAD **D** Enable code that computes a Tenex style load average. See *la*(1), *gldav*(2), *loadav*(3).

UCB_METER **D** Keep statistics on memory, queue sizes, process states, interrupts, traps, and many other (possibly useful) things. See *vmstat*(1) and

section 7.5 of this paper.

5.3.6. Device Drivers

In this section, an **XX_** prefix refers to the UNIX name of the device for which the option is intended to be enabled. For example, **TM_IOCTL** refers to mag tape *ioctl*s in tm.c. Most of these definitions go in the header file *xx.h* for the device. The exceptions are **BADSECT**, **MAXBAD**, **UCB_DEVERR**, and **UCB_ECC**.

BADSECT	**C**	Enable bad-sector forwarding. Sectors marked bad by the disk formatter are transparently replaced when read or written. Currently, only the hk driver's code has been thoroughly tested.
DDMT	**C**	Currently used only by the tm driver. Should be defined if you have a TM-11 emulator which supports 800/1600 bpi dual density drives with software selection.
DZ_PDMA	**C**	Configure the dz driver to do pseudo-dma.
MAXBAD	**C**	This sets the maximum number of replacement sectors available on a disk supporting DEC standard bad sector forwarding. It can be no larger than 126 but may be smaller to reduce the size of kernel data space. See the include file */usr/include/sys/dkbad.h*.
TEXAS_AUTOBAUD	**C**	Support an *ioctl* which defeats detection of framing or parity errors. This is used by *getty*(8) to accurately guess a line's speed when a carriage return is typed.
UCB_DBUF	**C**	If defined for a given disk driver, the driver will use one raw buffer per drive rather than one per controller. This increases throughput for controllers that are capable of seeking on one drive while simultaneously transferring on another.
UCB_DEVERR	**D**	Print device error messages in a human readable (mnemonic) format.
UCB_ECC	**C**	Recognize and correct soft ecc disk transfer errors.
VP_TWOSCOMPL	**C**	Used in the Versatec (vp) driver. If defined, the byte count register will be loaded with the twos-complement of the byte count, rather than the byte count itself. Check your controller manual to see whether your controller requires this.
XX_IOCTL	**D**	Turn on optional *ioctl*s for the corresponding device. See section 4 of the Berkeley PDP-11 UNIX Programmer's manual for details.
XX_SILO	**D**	Used in the dh and dz drivers. If defined, the drivers will use silo interrupts to avoid taking an interrupt for each character received.
XX_SOFTCAR	**C**	Currently used only by the dh and dz drivers. Should be defined if not all of the lines on a DH-11 or DZ-11 use modem control. It allows one to select lines on which modem control will be disabled. See *dh*(4) and *dz*(4). It can also be used with escape-code autodialers to allow modem control to be ignored while talking to the dialer.
XX_TIMEOUT	**D**	Enable a watchdog timer. This is used to kick devices prone to losing interrupts. It is currently available only for the tm driver.

5.3.7. Miscellaneous System Calls

UCB_LOGIN	**C**	Support a system call which can mark a process as a "login process" and set its recharge number (for accounting purposes). This is usually done by *login*(1). See *login*(2).

UCB_RENICE **D** Support a system call which allows a user to dynamically change a process's "nice" value over the entire range (-127 to 127) of values. See *renice*(1) and *renice*(2).

UCB_SUBM **C** Support a system call to mark a process as having been "submitted," permitting it to run after the user has logged out and enabling special accounting for its CPU use. See *submit*(1) and *submit*(2). If this option is enabled, *init*(8) sends a SIGKILL signal to a user's unsubmitted processes when that user logs out. It is ineffective if **MENLO_JCL** is defined.

5.3.8. Performance Tuning

NOKA5 **C** Simplify the code for kernel remapping by assuming that KDSA5 will not be used for normal kernel data. Kernel data space must end before 0120000 if this option is enabled. It is unfortunate but unavoidable that one must first make a kernel and size it to determine whether this option may be safely defined. It is usually possible on all but the largest separate I/D kernels, and on the small-to-medium nonseparate, overlaid kernels. The *checksys* utility will print a warning message if the data limit is exceeded when a new kernel is loaded.

PROFIL **C** Turn on system profiling. This requires a separate I/D cpu equipped with a KW11-P clock. It cannot be used on machines with ENABLE/34 boards since they have no spare page address registers. If profiling is enabled, you should change the definition of SPLFIX in the corresponding machine Makefile to *:splfix.profil.* The directory */usr/contrib/getsyspr* contains a program for extracting the profiling information from the kernel.

UCB_BHASH **D** Compile in code to hash buffer headers (and cut the time required by the *getblk* routine by 50% or more on large systems).

UCB_FRCSWAP **C** Force swaps on all forks and expands (but not vforks). This is used to transfer some of the load from a compute-bound CPU to an idle disk controller. This is probably not a good idea with **VIRUS_VFORK** defined, but then the load is better reduced by using vfork instead of fork.

UCB_IHASH **D** Compile in code to hash in-core inodes (and cut the time required by the *iget* routine by 50% or more on large systems).

UNFAST **C** Do not use inline macro expansions designed to speed up file system accesses at the cost of a larger text segment.

5.4. Additional configuration details

A few of the parameters and options require a little care to set up; those considerations are discussed here.

5.4.1. Alternate disk drivers

There are several disk drivers provided for SMD disks. The **hp** driver supports RP04/05/06 disks; **rm** supports RM02/03 disks, and **dvhp** supports 300 Mbyte drives on Diva controllers. In addition, there is an **xp** driver which handles any of the above, plus RM05 disks, multiple controllers, and disks which are similar to those listed but with different geometry (e.g. Fujitsu 160 Mbyte drives). It can be used with UNIBUS or MASSBUS controllers or both. In general, if you have only one type of disk and one controller, the **hp**, **rm** or **dvhp** drivers are the best choices, since they are smaller and simpler. If you use the **xp** driver, it can be set up in one of two ways. If **XP_PROBE** is defined in xp.h, the driver will attempt to determine the type of each disk and

controller by probing and using the drive type register. To save the space occupied by this routine, or to specify different drive parameters, the drive and controller structures can be initialized in ioconf.c if **XP_PROBE** is not defined. The controller addresses will have to be initialized in either case (at least the first, if it is a boot device). The file /usr/include/sys/hpreg.h provides the definitions for the flags and sizes. Ioconf.c has an example of initialized structures. *Xp*(4) gives more information about drive numbering, etc.

5.4.2. Disk monitoring parameters

The kernel is capable of maintaining statistics about disk activity for specified disks; this information can be printed by *iostat*(8). This involves some setup, however, and if parameters are set incorrectly can cause the kernel monitoring routines to overrun their array bounds. To set this up correctly, choose the disks to be monitored. *Iostat* is configured for a maximum of 4 disks, but that could be changed by editing the headers. The drivers that do overlapped seeks (hk, hp, rm and xp) use one field for each drive (N*XX*) plus one for the controller; the others use only one field, for the controller. When both drives and controllers are monitored, the drives come first, starting at *XX*_DKN, followed by the controller (or controllers, in the case of xp). Then set **NDISK** in param.c to the desired number. The number of the first slot to use for each driver is defined as *XX*_DKN in the device's header file, or is undefined if that driver is not using monitoring. *Iostat* currently expects that if overlapped seeks are being metered, those disks are first in the array (i.e., *XX*_DKN for that driver is 0). As an example, for 3 RP06 disks using the hp driver plus 1 RL02, HP_DKN should be 0, RL_DKN should be 4, and **NDISK** should be 5 (3 hp disks + 1 hp controller + 1 rl). The complete correspondence for *iostat* would then be:

0 (HP_DKN + 0)	hp0 seeks
1 (HP_DKN + 1)	hp1 seeks
2 (HP_DKN + 2)	hp2 seeks
3 (HP_DKN + NHP)	hp controller transfers
4 (RL_DKN + 0)	rl transfers

It is very important that NDISK be large enough, since the drivers do not check for overflow.

After the kernel disk monitoring is set up, *iostat* itself needs to be edited to reflect the numbers and types of the disks. The source is in /usr/src/cmd.

5.4.3. Automatic reboot

The automatic reboot facility (**UCB_AUTOBOOT**) includes a number of components, several of which must know details of the boot configuration. The kernel has an integral boot routine, found in boot.s in the configuration directory for the machine, which reads in a block 0 bootstrap from the normal boot device and executes it. The block 0 bootstrap normally loads **boot** from the first file system on drive 0 of the disk; this can be changed if necessary. The second-stage bootstrap, /boot, needs to know where to find unix.

The first step is to determine which kernel boot to use. Currently, there are boot modules supplied for the following disk types: hk, rl, rm, rp, dvhp, sc11 and sc21 (the last two are for Emulex SC11 and SC21 controllers, using the boot command). If one of these will work with your boot disk, place that entry in the **bootdev** field in the device configuration file before running *config*, or simply copy ../conf/*dk*boot.s to boot.s in the machine configuration directory. If no boot module supplied will work, it is not too difficult to create one for your machine. The easiest way to do this is to copy one of the other boot modules, and modify the last section which actually reads the boot block. If you have a bootstrap ROM, you can simply jump to the correct entry with any necessary addresses placed in registers first. Or, you can write a small routine to read in the first disk block. If you don't have a boot module, **bootdev** in the configuration file should be specified as **none**, and noboot.s will be installed. This is a dummy file that keeps the load rules from changing. The **UCB_AUTOBOOT** option should not be defined until a boot

module is obtained.

The other change that is normally required is to specify where /unix will be found. This is done by changing the definition of **RB_DEFNAME** in /usr/include/sys/reboot.h. The definition is a string in the same format as the manual input to boot, for example "xp(0,0)unix". After making this change, boot will need to be recompiled (in /usr/src/sys/stand/bootstrap) and installed. It can be installed initially as /newboot, and the original boot can be used to load it for testing:

>**boot**

*nn***Boot**
: *dk*(0,0)newboot

*nn***Boot**
: *dk*(0,0)unix

If you want to have core dumps made after crashes, this must be specified in the configuration file as well. Dumps are normally taken on the end of the swap device before rebooting, and after the system is back up and the file systems are checked, the dump will be copied into /usr/sys by *savecore*(8). Dump routines are available for the hk, hp, rm and xp drivers. To install, change the **dumpdev** entry to the same value as the swap device. Then set **dumplo** to a value that will allow as much as possible of memory to be saved. The dump routine will start the dump at dumplo and continue to the end of memory or the end of the swap device partition, whichever comes first. Dumplo should be larger than swplo so that any early swaps will not overwrite the dump, but if possible, should be low enough that there is room for all of memory. The **dumproutine** entry in the configuration file is then set to *dk*dump, where *dk* is the disk type. Finally, after running *config*, edit the header file *dk*.h in the new configuration directory to define *DK*_DUMP, so that that dump routine will be included when the driver is compiled.

5.4.4. Considerations on a PDP-11/23

If setting up a kernel on a PDP-11/23, it is necessary to consider the interrupt structure of the hardware. If there are any single-priority boards on the bus, they must be behind all multiple-priority devices. Otherwise, they may accept interrupts meant for another, higher-priority device farther from the processor, at a time when the system has set the processor priority to block the single-level device. The alternative is to use spl6 uniformly for any high processor priority (spl4, spl5, spl6). This may be accomplished by changing the _spl routines in mch.s, the definitions of br4 and br5 in l.s, and by changing the script :splfix.mtps (in the *conf* directory).

Berkeley UNIX does not support more than 256K bytes of memory on the 11/23. If you have extra memory and a way to use it (e.g. a disk driver capable of 22-bit addressing) you will want to change this.

5.5. Compiling the kernel

Once you have made any local changes, you are ready to compile the kernel. If you have made any changes which will affect the dependency rules in the Makefile, run "make depend" (N.B.: the output of this command is best appreciated on a crt). Then, "make unix." Note: although several shortcuts have been built into the makefile, the nonseparate I/D *make* occasionally runs out of space while recompiling the kernel. If this happens, just restart it and it will generally make it through the second time. The separate I/D version of *make* in /usr/70 should have no problem. Also note, it is imperative that overlaid kernels be compiled with the 2.9BSD versions of *cc, as* (and *as2*) and *ld*. Use of older C preprocessors or assemblers will result in compile-time errors or (worse) systems that will almost run, but crash after a short time.

After the unix binary is loaded, the makefile runs a small program called *checksys* which checks for size overflows. If you are building an overlaid system, check the size of the object file (see *size*(1)) and overlay layout. The overlay structure may be changed by editing the makefile.

For a nonseparate I/D system, the base segment size must be between 8194 and 16382 bytes and each overlay must be at most 8192 bytes. If you are building an overlaid system with ENABLE/34 support, note that the object module *enable34.o* must be loaded in the base segment. The final object file "unix" should be copied to the root, and then booted to try it out. It is best to name it /newunix so as not to destroy the working system until you're sure it does work:

```
# cp unix /newunix
# sync
```

It is also a good idea to keep the old system around under some other name. In particular, we recommend that you save the generic distribution version of the system permanently as /genericunix for use in emergencies.

To boot the new version of the system you should follow the bootstrap procedures outlined in section 2.4 above. A systematic scheme for numbering and saving old versions of the system is best.

You can repeat these steps whenever it is necessary to change the system configuration.

5.6. Making changes to the kernel

If you wish to make local mods to the kernel you should bracket them with

```
#ifdef PICKLE
...
#endif
```

perhaps saving old code between

```
#ifndef PICKLE
...
#endif
```

This will allow you to find changed code easily.

To add a device not supported by the distribution system you will have to place the driver for the device in the directory /usr/src/sys/dev, edit a line into the block and/or character device table in /usr/src/sys/PICKLE/c.c, add the name of the device to the OPTIONAL line of the file Depend, and to the makefile load rules. Place the device's address and interrupt vector in the files ioconf.c and l.s respectively if it is not going to be configured by *autoconfig*(8); otherwise, l.s will only need the normal interface to the C interrupt routine. If you use autoconfiguration, you will need an attach routine in the driver, and a probe routine in the driver or in *autoconfig*. Use the entries for a similar device as an example. If the device driver uses the UNIBUS map or system buffers, it will probably need modifications. Check "Changes in the Kernel in 2.9BSD" for more technical information regarding driver interfacing. You can then rebuild the system (be sure to make *depend* first). After rebooting the resulting kernel and making appropriate entries in the /dev directory, you can test out the new device and driver. Section 7.1 explains shutdown and reboot procedures.

6. RECOMPILING SYSTEM SOFTWARE

We now describe how to recompile system programs and install them. Some programs must be modified for the local system at this time, and other local changes may be desirable now or later. Before any of these procedures are begun, be certain that the include files <whoami.h>, <sys/localopts.h> and <sys/param.h> are correct for the kernel that has been installed. This is important for commands that wish to know the name of the local machine or that size their data areas appropriately for the type of CPU. The general procedures are given first, followed by more detailed information about some of the major systems that require some setup.

6.1. Recompiling and reinstalling system software

It is easy to regenerate the system, and it is a good idea to try rebuilding pieces of the system to build confidence in the procedures. The system consists of three major parts: the kernel itself, along with the bootstrap and standalone utilities (/usr/src/sys), the user programs (/usr/src/cmd, /usr/src/ucb, and subdirectories), and the libraries (/usr/src/lib). The major part of this is /usr/src/cmd.

We have already seen how to recompile the system itself. The commands and libraries can be recompiled in their respective source directories using the Makefile (or Ovmakefile if there are both overlaid and non-overlaid versions). However, it is generally easier to use one of the MAKE scripts set up for /usr/src/lib, /usr/src/cmd, and /usr/src/ucb. These are used in a similar fashion, such as

```
# ./MAKE -40 [ -cp ] [ -f ] file ...
```

The first, required flag sets the CPU class for which to compile. Three classes are used to used to set requirements for separate instruction and data and for floating point. "MAKE -40" makes nonseparate I/D versions that load the floating point interpreter as required. "MAKE -34" is similar but assumes a hardware floating point unit. "MAKE -70" is used for separate I/D machines and also assumes floating point hardware. "MAKE -70 -f" is used for separate I/D machines without floating point hardware. The use of these MAKE scripts automates the selection of CPU-dependent options and makes the optimal configuration of each program for the target computer. The optional argument -cp causes each program to be installed as it is made. They are installed in the normal directories, unless the environment variable DESTDIR is set, in which case the normal path is prepended by DESTDIR. This can be used to compile and create a new set of binary directories, e.g. /nbsd/bin, /nbsd/lib, etc. Running the command "MAKE -70 -cp *" in /usr/src/lib, /usr/src/cmd and /usr/src/ucb would thus create a whole new tree of system binaries. The six major libraries are the C library in /usr/src/lib/c, the jobs library, /usr/src/lib/jobs, the FORTRAN libraries /usr/src/lib/libF77, /usr/src/lib/libI77, and /usr/src/lib/libU77, and the math library /usr/src/lib/m. Most libraries are made in two versions, one each for use with and without process overlays. In each case the library is remade by changing into /usr/src/lib and doing

./MAKE *-cpu libname*

or made and installed by

./MAKE *-cpu* -cp *libname*

Similar to the system,

```
# make clean
```

cleans up in each subdirectory.

To recompile individual commands, change to /usr/src/cmd or /usr/src/ucb, as appropriate, and use the MAKE script in the same way. Thus to compile adb, do

```
# ./MAKE -cpu adb
```

where cpu is 34, 40, or 70. To recompile everything, use

```
# ./MAKE -cpu *
```

After installing new binaries, you can use the script in /usr/src to link files together as necessary and to set all the right set-user-id bits.

```
# cd /usr/src
# ./MAKE aliases
# ./MAKE modes
```

6.2. Making local modifications

To keep track of changes to system source we migrate changed versions of commands in /usr/src/cmd in through the directory /usr/src/new and out of /usr/src/cmd into /usr/src/old for a time before removing them. Locally written commands that aren't distributed are kept in /usr/src/local and their binaries are kept in /usr/local. This allows /usr/bin, /usr/ucb, and /bin to correspond to the distribution tape (and to the manuals that people can buy). People wishing to use /usr/local commands are made aware that they aren't in the base manual. As manual updates incorporate these commands they are moved to /usr/ucb.

A directory /usr/junk to throw garbage into, as well as binary directories /usr/old and /usr/new are useful. The *man*(1) command supports manual directories such as /usr/man/man*n* for *new* and /usr/man/man*l* for *local* to make this or something similar practical.

6.3. Setting up the mail system

The mail system can be set up in at least two ways. One strategy uses the *delivermail*(8) program to sort out network addresses according to the local network topology. It is not perfect, especially in the light of changing ARPAnet conventions. However, if you use the Berkeley network or are connected directly or indirectly to the ARPAnet, it is probably the method of choice for the time being. On the other hand, if you use only local mail and UUCP mail, /bin/mail (*mail*(1)) will suffice as a mail deliverer. In that case, you will only need to recompile *mail*(1) and *Mail*(1).

The entire mail system consists of the following commands:

/bin/mail	old standard mail program (from V7 or System III)
/usr/ucb/Mail	UCB mail program, described in Mail(1)
/usr/lib/Mail.rc	aliases and defaults for Mail(1)
/etc/delivermail	mail routing program
/usr/net/bin/v6mail	local mailman for berknet
/usr/spool/mail	mail spooling directory
/usr/spool/secretmail	secure mail directory
/usr/bin/xsend	secure mail sender
/usr/bin/xget	secure mail receiver
/usr/lib/aliases	mail forwarding information for delivermail
/usr/ucb/newaliases	command to rebuild binary forwarding database

Mail is normally sent and received using the *Mail*(1) command, which provides a front-end to edit the messages sent and received, and passes the messages to *delivermail*(8) or *mail*(1) for routing and/or delivery.

Mail is normally accessible in the directory /usr/spool/mail and is readable by all users.† To

† You can make your mail unreadable by others by changing the mode of the file /usr/spool/mail/*yourname* to 600 and putting the line "set keep" in your .mailrc file. The directory /usr/spool/mail must not be writable (mode 755) for this to work.

send mail which is secure against any possible perusal (except by a code-breaker) you should use the secret mail facility, which encrypts the mail so that no one can read it.

6.3.1. Setting up mail and Mail

Both /bin/mail and /usr/ucb/Mail should be recompiled to make local versions. Remake mail in /usr/src/cmd with the command

./MAKE –*cpu* mail

Install the new binary in /bin after testing; it must be setuserid root. Section 6.1 gives more details on the use of the MAKE scripts. To configure *Mail,* change directories to /usr/src/ucb/Mail. Edit the file v7.local.h to assign a letter to your machine with the definition of LOCAL; if you do not have a local area network, the choice is arbitrary as long as you pick an unused letter. If you wish to use *delivermail,* the definition of SENDMAIL should be uncommented. Then add your machine to the table in config.c; configdefs.h gives some information on this. The network field should specify which networks (if any) you are connected to (note: the Schmidt net, SN, is Berknet). After the changes are made, move to /usr/src/ucb and

./MAKE –40 Mail (on a nonseparate I/D machine)
or
./MAKE –70 Mail (on a separate I/D machine)

Install *Mail* in /usr/ucb; it should **not** be setuserid. The Mail.rc file in /usr/lib can be used to set up limited distribution lists or aliases if you are not using *delivermail.*

6.3.2. Setting up delivermail

To set up the *delivermail* facility you should read the instructions in the file READ_ME in the directory /usr/src/ucb/delivermail and then adjust and recompile the *delivermail* program, installing it as /etc/delivermail. The routing algorithm uses knowledge of network name syntax built into its tables and aliasing and forwarding information built into the file /usr/lib/aliases to process each piece of mail. Local mail is delivered by giving it to the program /usr/net/bin/v6mail which adds it to the mailboxes in the directory /usr/spool/mail/*username,* using a locking protocol to avoid problems with simultaneous updates. You should also set up the file /usr/lib/aliases for your installation, creating mail groups as appropriate.

6.4. Setting up a uucp connection

The version of *uucp* included in 2.9BSD is an enhanced version of that originally distributed with V7*. The enhancements include:

- support for many auto call units other than the DEC DN11,
- breakup of the spooling area into multiple subdirectories,
- addition of an *L.cmds* file to control the set of commands which may be executed by a remote site,
- enhanced "expect-send" sequence capabilities when logging in to a remote site,
- new commands to be used in polling sites and obtaining snap shots of *uucp* activity.

This section gives a brief overview of *uucp* and points out the most important steps in its installation.

To connect two UNIX machines with a *uucp* network link using modems, one site must have an automatic call unit and the other must have a dialup port. It is better if both sites have both.

You should first read the paper in volume 2B of the Unix Programmers Manual: "Uucp Implementation Description". It describes in detail the file formats and conventions, and will give

* The *uucp* included in this distribution is the result of work by many people; we gratefully acknowledge their contributions, but refrain from mentioning names in the interest of keeping this document current.

you a little context. In addition, the document setup.tblms, located in the directory /usr/src/usr.bin/uucp/UUAIDS, may be of use in tailoring the software to your needs.

The *uucp* support is located in three major directories: /usr/bin, /usr/lib/uucp, and /usr/spool/uucp. User commands are kept in /usr/bin, operational commands in /usr/lib/uucp, and /usr/spool/uucp is used as a spooling area. The commands in /usr/bin are:

/usr/bin/uucp	file-copy command
/usr/bin/uux	remote execution command
/usr/bin/uusend	binary file transfer using mail
/usr/bin/uuencode	binary file encoder (for *uusend*)
/usr/bin/uudecode	binary file decoder (for *uusend*)
/usr/bin/uulog	scans session log files
/usr/bin/uusnap	gives a snap-shot of *uucp* activity
/usr/bin/uupoll	polls remote system until an answer is received

The important files and commands in /usr/lib/uucp are:

/usr/lib/uucp/L-devices	list of dialers and hardwired lines
/usr/lib/uucp/L-dialcodes	dialcode abbreviations
/usr/lib/uucp/L.cmds	commands remote sites may execute
/usr/lib/uucp/L.sys	systems to communicate with, how to connect, and when
/usr/lib/uucp/SEQF	sequence numbering control file
/usr/lib/uucp/USERFILE	remote site pathname access specifications
/usr/lib/uucp/uuclean	cleans up garbage files in spool area
/usr/lib/uucp/uucico	*uucp* protocol daemon
/usr/lib/uucp/uuxqt	*uucp* remote execution server

while the spooling area contains the following important files and directories:

/usr/spool/uucp/C.	directory for command, "C." files
/usr/spool/uucp/D.	directory for data, "D.", files
/usr/spool/uucp/X.	directory for command execution, "X.", files
/usr/spool/uucp/D.*machine*	directory for local "D." files
/usr/spool/uucp/D.*machine*X	directory for local "X." files
/usr/spool/uucp/TM.	directory for temporary, "TM.", files
/usr/spool/uucp/LOGFILE	log file of *uucp* activity
/usr/spool/uucp/SYSLOG	log file of *uucp* file transfers

To install *uucp* on your system, start by selecting a site name (less than 8 characters). A *uucp* account must be created in the password file and a password set up. Then, create the appropriate spooling directories with mode 755 and owned by user *uucp*, group *daemon*.

If you have an auto-call unit, the L.sys, L-dialcodes, and L-devices files should be created. The L.sys file should contain the phone numbers and login sequences required to establish a connection with a *uucp* daemon on another machine. For example, our L.sys file looks something like:

```
adiron Any ACU 1200 out0123456789- ogin-EOT-ogin uucp
cbosg Never Slave 300
cbosgd Never Slave 300
chico Never Slave 1200 out2010123456
```

The first field is the name of a site, the second indicates when the machine may be called, the third field specifies how the host is connected (through an ACU, a hardwired line, etc.), then comes the phone number to use in connecting through an auto-call unit, and finally a login sequence. The phone number may contain common abbreviations which are defined in the L-

dialcodes file. The device specification should refer to devices specified in the L-devices file. Indicating only ACU causes the *uucp* daemon, *uucico,* to search for any available auto-call unit in L-devices. Our L-dialcodes file is of the form:

ucb 2
out 9%

while our L-devices file is:

ACU cul0 unused 1200 ventel

Refer to the README file in the *uucp* source directory for more information about installation.

As *uucp* operates it creates (and removes) many small files in the directories underneath /usr/spool/uucp. Sometimes files are left undeleted; these are most easily purged with the *uuclean* program. The log files can grow without bound unless trimmed back; *uulog* is used to maintain these files. Many useful aids in maintaining your *uucp* installation are included in a subdirectory UUAIDS beneath /usr/src/usr.bin/uucp. Peruse this directory and read the "setup" instructions also located there.

6.5. Miscellaneous software

The directory /usr/contrib contains programs and packages that you may wish to install on your system. Some were directly contributed; others were collected from the *usenet* news group *net.sources*. Also, some programs or libraries in other directories are sufficiently unique to be noteworthy. Here is a brief summary.

6.5.1. Ape

Ape (Arbitrary Precision Extended) is a replacement for the multiple precision arithmetic routines (*mp*(3)). It is much faster and contains numerous bug fixes.

6.5.2. L11, M11

M11 is a Macro-11 assembler. It recognizes and emulates almost all of the directives of standard DEC Macro-11 assemblers. *L11* is its loader.

6.5.3. Jove

Jove (Jonathan's Own Version of EMACS) is an EMACS style editor developed at Lincoln Sudbury Regional High School.

6.5.4. Kernel scheduler modifications

The scheduler modifications made by Darwyn Peachey at the University of Saskatchewan are included here but have not been incorporated into the distribution kernel (although it would not be hard). It improves the response of interactive jobs and provides a real time facility different from the one currently implemented.

6.5.5. News

The network bulletin board system developed at Duke University and the University of North Carolina and since heavily modified at Berkeley.

6.5.6. Notes

The network bulletin board system developed at the University of Illinois. This version contains many enhancements and clean *news* interfaces.

6.5.7. Ranm

Ranm is a fast uniform pseudorandom number generator package developed at Berkeley.

7. SYSTEM OPERATION

This section describes procedures used to operate a PDP-11 UNIX system. Procedures described here are used periodically, to reboot the system, analyze error messages from devices, do disk backups, monitor system performance, recompile system software and control local changes.

7.1. Bootstrap and shutdown procedures

The system boot procedure varies with the hardware configuration, but generally uses the console emulator or a ROM routine to boot one of the disks. /boot comes up and prompts (with ": ") for the name of the system to load. Simply hitting a carriage return will load the default system. The system will come up with a single-user shell on the console. To bring the system up to a multi-user configuration from the single-user status, all you have to do is hit ^D on the console (you should check and, if necessary, set the date before going multiuser; see *date*(1)). The system will then execute /etc/rc, a multi-user restart script, and come up on the terminals listed as active in the file /etc/ttys. See *init*(8) and *ttys*(5). Note, however, that this does not cause a file system check to be performed. Unless the system was taken down cleanly, you should run "fsck -p" or force a reboot with *reboot*(8) to have the disks checked.

In an automatic reboot, the system checks the disks and comes up multi-user without intervention at the console. If the file system check fails, or is interrupted (after it prints the date) from the console when a delete/rubout is hit, it will leave the system in special-session mode, allowing root to log in on one of a limited number of terminals (generally including a dialup) to repair file systems, etc. The system is then brought to normal multiuser operations by signaling init with a SIGINT signal (with "kill -INT 1").

To take the system down to a single user state you can use

```
# kill 1
```

or use the *shutdown*(8) command (which is much more polite if there are other users logged in) when you are up multi-user. Either command will kill all processes and give you a shell on the console, almost as if you had just booted. File systems remain mounted after the system is taken single-user. If you wish to come up multi-user again, you should do this by:

```
# cd /
# /etc/umount -a
# ^D
```

The system can also be halted or rebooted with *reboot*(8) if automatic reboots are enabled. Otherwise, the system is halted by switching to single-user mode to kill all processes, updating the disks with a "sync" command, and then halting.

Each system shutdown, crash, processor halt and reboot is recorded in the file /usr/adm/shutdownlog with the cause.

7.2. Device errors and diagnostics

When errors occur on peripherals or in the system, the system prints a warning diagnostic on the console. These messages are collected regularly and written into a system error log file /usr/adm/messages by *dmesg*(8).

Error messages printed by the devices in the system are described with the drivers for the devices in section 4 of the Berkeley PDP-11 UNIX Programmer's manual. If errors occur indicating hardware problems, you should contact your hardware support group or field service. It is a good idea to examine the error log file regularly (e.g. with "tail -r /usr/adm/messages").

If you have DEC field service, they should know how to interpret these messages. If they do not, tell them to contact the DEC UNIX Engineering Group.

7.3. File system checks, backups and disaster recovery

Periodically (say every week or so in the absence of any problems) and always (usually automatically) after a crash, all the file systems should be checked for consistency by *fsck*(8). The procedures of *boot*(8) or *reboot*(8) should be used to get the system to a state where a file system check can be performed manually or automatically.

Dumping of the file systems should be done regularly, since once the system is going it is easy to become complacent. Complete and incremental dumps are easily done with *dump*(8). You should arrange to do a towers-of-Hanoi dump sequence; we tune ours so that almost all files are dumped on two tapes and kept for at least a week in almost every case. We take full dumps every month (and keep these indefinitely).

Dumping of files by name is best done by *tar*(1) but the amount of data that can be moved in this way is limited to a single tape. Finally, if there are enough drives, entire disks can be copied with *dd*(1) using the raw special files and an appropriate block size.

It is desirable that full dumps of the root file system are made regularly. This is especially true when only one disk is available. Then, if the root file system is damaged by a hardware or software failure, you can rebuild a workable disk using a standalone restore in the same way that *restor* was used to build the initial root file system.

Exhaustion of user-file space is certain to occur now and then; the only mechanisms for controlling this phenomenon are occasional use of *df*(1), *du*(1), *quot*(8), threatening messages of the day, personal letters, and (probably as a last resort) quotas (see *setquota*(8)).

7.4. Moving file system data

If you have the equipment, the best way to move a file system is to dump it to magtape using *dump*(8), to use *mkfs*(8) to create the new file system, and restore, using *restor*(8), the tape. If for some reason you don't want to use magtape, dump accepts an argument telling where to put the dump; you might use another disk. Sometimes a file system has to be increased in logical size without copying. The super-block of the device has a word giving the highest address that can be allocated. For small increases, this word can be patched using the debugger *adb*(1) and the free list reconstructed using *fsck*(8). The size should not be increased greatly by this technique, since the file system will then be short of inode slots. Read and understand the description given in *filsys*(5) before playing around in this way.

If you have to merge a file system into another, existing one, the best bet is to use *tar*(1). If you must shrink a file system, the best bet is to dump the original and restor it onto the new file system. However, this will not work if the i-list on the smaller file system is smaller than the maximum allocated inode on the larger. If this is the case, reconstruct the file system from scratch on another file system (perhaps using *tar*(1)) and then dump it. If you are playing with the root file system and only have one drive the procedure is more complicated. What you do is the following:

1. GET A SECOND PACK!!!!
2. Dump the root file system to tape using *dump*(8).
3. Bring the system down and mount the new pack.
4. Load the standalone versions of *mkfs*(8) and *restor*(8) as in sections 2.1-2.3 above.
5. Boot normally using the newly created disk file system.

Note that if you add new disk drivers they should also be added to the standalone system in /usr/src/sys/stand.

7.5. Monitoring System Performance

The *iostat*(8) and *vmstat*(8) programs provided with the system are designed to aid in monitoring systemwide activity. By running them when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, swapping activity, disk and

CPU utilization. Ideally, there should be few blocked (DW) jobs, there should be little swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 30-35 tps in practice), and the user CPU utilization (US) should be high (above 60%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (DW).

If you run *vmstat* when the system is busy (a "vmstat 5" gives all the numbers computed by the system), you can find imbalances by noting abnormal job distributions. If many processes are blocked (DW), then the disk subsystem is overloaded or imbalanced. If you have several non-DMA devices or open teletype lines that are "ringing", or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-70% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (CS), interrupt activity (IN) or system call activity (SY).

If the system is heavily loaded, or if you have little memory for your load (248K is little in almost any case), then the system will be forced to swap. This is likely to be accompanied by a noticeable reduction in system performance and pregnant pauses when interactive jobs such as editors swap out. If you expect to be in a memory-poor environment for an extended period you might consider administratively limiting system load.

7.6. Adding users

New users can be added to the system by adding a line to the password file /etc/passwd. You should add accounts for the initial user community, giving each a directory and a password, and putting users who will wish to share software in the same group. User id's should be assigned starting with 16 or higher, as lower id's are treated specially by the system. Default startup files should probably provided for new users and can be copied from /usr/public. Initial passwords should be set also.

A number of guest accounts have been provided on the distribution system; these accounts are for people at Berkeley and at Bell Laboratories who have done major work on UNIX in the past. You can delete these accounts, or leave them on the system if you expect that these people would have occasion to login as guests on your system.

7.7. Accounting

UNIX currently optionally records two kinds of accounting information: connect time accounting and process resource accounting. The connect time accounting information is normally stored in the file /usr/adm/wtmp, which is summarized by the program *ac*(8). The process time accounting information is stored in the file /usr/adm/acct, and analyzed and summarized by the program *sa*(8).

If you need to implement recharge for computing time, you can implement procedures based on the information provided by these commands. A convenient way to do this is to give commands to the clock daemon /etc/cron to be executed every day at a specified time. This is done by adding lines to /usr/adm/crontab; see *cron*(8) for details.

7.8. Resource control

Resource control in the current version of UNIX is rather primitive. Disk space usage can be monitored by *du*(1) or *quot*(8) as was previously mentioned. Disk quotas can be set and changed with *setquota*(8) if the kernel has been configured for quotas. Our quota mechanism is simplistic and easily defeated but does make users more aware of the amount of space they use.

7.9. Files which need periodic attention

We conclude the discussion of system operations by listing the files and directories that continue to grow and thus require periodic truncation, along with references to relevant manual pages. *Cron*(8) can be used to run scripts to truncate these periodically, possibly summarizing first or saving recent entries. Some of these can be disabled if you don't need to collect the

information.

/usr/adm/acct	sa(8)	raw process account data
/usr/adm/messages	dmesg(8)	system error log
/usr/adm/shutdownlog	shutdown(8)	log of system reboots
/usr/adm/wtmp	ac(8)	login session accounting
/usr/spool/uucp/LOGFILE	uulog(1)	uucp log file
/usr/spool/uucp/SYSLOG	uulog(1)	more uucp logging
/usr/dict/spellhist	spell(1)	spell log
/usr/lib/learn/log	learn(1)	learn lesson logging
/usr/sys	savecore(8)	system core images

8. KERNEL MAGIC NUMBERS

This sections contains a collection of magic numbers for use in patching core or an executable unix binary. Some of them have also been mentioned earlier in this paper. With the exception of the *xp_type[i]* variables (which hold bytes) and *swplo* (which is a long) all locations given contain short integers. N.B.: in the case of paired interrupt vectors (for DHs and DZs) the address of the second vector of the pair is four more than the address of the first vector.

Interrupt Vectors

Vector	Handler	Contents	Block device	Character device
0160	rlio	01202	8	18
0210	hkio	01142	4	19
0220	rkio	01172	0	9
0224	tmio	01222	3	12
0224	htio	01152	7	15
0224	tsio	01232	9	20
0254	xpio	01242	6	14
0260	rpio	01212	1	11
†	dzin	01132	-	21
†	dzdma	02202	-	21
†	dhin	01112	-	4
†	dhou	01122	-	4
†	lpio	01162	-	2

† Set by *autoconfig*(8).

Other Variables

Name	Address	Contents
xp_addr	061464	0176700
xp_type[0]	061472	‡
xp_type[1]	061506	‡
xp_type[2]	061522	‡
xp_type[3]	061536	‡
HKADDR	061006	0177440
HTADDR	0114226	†
RKADDR	061152	0177400
RLADDR	061154	0174400
RPADDR	061236	0176710
TMADDR	0113320	†
TSADDR	0113612	†
dz_addr	0113324	†
dh_addr	0114136	†
lp_addr	0113452	†
rootdev	060772	*
pipedev	060776	*
swapdev	060774	*
swplo	061000	*
nswap	061004	*

† Set by *autoconfig*(8).
‡ Set by reading the corresponding drive type register.
* System dependent.

Changes in the Kernel in 2.9BSD

Michael J. Karels

Department of Molecular Biology
University of California, Berkeley
Berkeley, California 94720

Carl F. Smith

Department of Mathematics
University of California, Berkeley
Berkeley, California 94720

William F. Jolitz

Symmetric Computer Systems
Los Gatos, California

This document summarizes changes in the PDP-11† UNIX‡ kernel between the July 1981 2.8BSD release and the July 1983 2.9BSD distribution. The kernel remains highly tunable, and changing *#define*d options may affect the validity of remarks in this paper.

The major changes fall into these categories:

[1] The new signal mechanism needed for process control has been added to the system, making the job control facilities of 4.1BSD available.

[2] *Vfork*, a form of *fork* which spawns a new process without fully copying the address space of the parent, is available to create a new context for an *exec* much more efficiently.

[3] The system can reboot itself automatically, after crashes or manually. The system is more crash-resistant and is able to take crash dumps before rebooting.

[4] A fast and reliable method of accessing mapped buffers and clists without increasing processor priority is now available.

[5] The protocols for allocation of the UNIBUS map have been changed, and DMA into system buffers with 18-bit addressing devices is also different.

[6] Changes have been made in code organization, so that more than one system configuration may be built from a single set of sources. Each system is described by a single file that includes parameters such as system size, devices, etc. Most of the "magic numbers" such as device register addresses and disk partitions are in one file, ioconf.c, and the number of devices of each type are in header files local to that system.

[7] Most devices are configured at boot time rather than at compilation time, reducing the work in system configuration and making it possible for one binary to work on several similar systems. References to nonexistent devices are now rejected rather than causing a crash.

[8] System diagnostics have been changed to a standard, readable format; file system diagnostics refer to file systems by name rather than device number. Device diagnostics refer to devices by name and print error messages mnemonically as well as in octal.

---

†PDP-11, DEC, UNIBUS and MASSBUS are trademarks of Digital Equipment Corporation.
‡UNIX is a trademark of Bell Laboratories.

Many other performance enhancements and bug fixes have been made. Some conditional compilation flags have been removed because the feature they control is now considered standard (e.g. UCB_BUFOUT). Other features have been grouped together and are now controlled by the same flag (e.g. code previously conditional on UCB_SMINO now depends on UCB_NKB).

Many of the changes in 2.9BSD are based on work by many other people. Several features are modeled on those of the 4.1BSD VMUNIX system, and much of the code comes directly from that source.

Converting local software

Most local changes should be easily ported to the new system. The actual system configuration is much simpler than with previous kernels.

There are many changes that affect the device drivers. The appendices give the details of the conversions necessary. Device drivers that used the kernel's in-address-space buffers must be rewritten to use mapped buffers or their own dedicated buffers. "Abuffers" have been removed from the current system.

Appendix A contains a description of the new data mapping protocols used to access mapped buffers, clists, and some tables.

The UNIBUS map is allocated dynamically. Kernel data space is no longer guaranteed to be mapped by any portion of the UNIBUS map. Any local software making such assumptions must now explicitly allocate a section of the UNIBUS map; *mapalloc* and *mapfree* may be used for objects with buffer headers. See Appendix B for a description of the new UNIBUS map protocols.

The line discipline switch has been reorganized slightly to make it cleaner. Some unused fields in the *linesw* structure have been removed. There is a default line discipline, DFLT_LDISC, which may not be assumed to be 0. See Appendix C for a description of the new terminal and line discipline protocols.

As part of the implementation of *vfork*, process images are scatter loaded. Standard system monitoring programs (e.g. *ps* and *w*) have been modified. Local software must be changed accordingly. See Appendix D for a more detailed description of *vfork*.

Sites may wish to convert their device drivers to use the new autoconfiguration features described in Appendix E.

Processors are described by capabilities rather than cpu type. Separate I/D spaces and UNIBUS maps are detected and supported independently. Thus it is much easier to describe machines with foreign hardware enhancements. In particular, the Able ENABLE/34† is automatically detected and supported.

A new bootstrap loader that loads all object files except 0405 replaces the old version that loaded only 0407, 0411, and 0430. The kernel assumes that *boot* has already set the kernel mode segmentation registers and cleared bss. Other bootstraps that do not do so will not work.

Organizational changes

The system compilation procedure has been changed so that more than one set of binaries may be made with a single set of source code. System sources are kept in the directories **sys/sys** and **sys/dev**. No binaries are kept in either of these directories.

The directory **sys/conf** contains several files related to system configuration. For each machine to be configured, a single file should be created in this directory. Each such file describes all the parameters of the machine necessary for building a system. The format of the configuration files is described in *config*(8)‡. This procedure is more fully described in "Installing and Operating 2.9BSD."

---

†ENABLE/34 is a trademark of ABLE Computer, Inc.

‡References of the form *X*(*Y*) mean the subsection named *X* in section *Y* of the Berkeley PDP-11 UNIX Programmer's manual.

Corresponding to each system to be configured, there is a subdirectory of **sys**. One prototype directory, **GENERIC**, is already there. This directory is created and the appropriate files are installed by *config*, based on information in the machine description file. The configuration program processes the information in the configuration file and produces:

1) A set of header files (e.g. **dh.h**) which contain the number of devices available to the target system. These definitions force conditional compilation of drivers, resulting in the inclusion or exclusion of driver code and the sizing of driver tables. This technique, based on compilation, is more powerful than a loader-based technique, since small sections of code may be easily conditionalized. Only drivers that are needed are included in the resulting system. Option flags that are specific to individual drivers are also placed in these header files.

2) The assembly language vector interface, **l.s**, which turns the hardware generated UNIBUS interrupt sequences into C calls to the driver interrupt routines.

3) A table file, **ioconf.c**, which defines controller addresses for each disk controller in the configured system, and the partition tables for the larger disks.

4) The files **localopts.h**, **param.c**, **param.h**, and **whoami.h**. These can be edited if local taste so dictates. **Whoami.h** contains the definition of PDP11, which will have one of the following values: 23, 34, 40, 44, 45, 60, 70, or GENERIC. The distributed binary is compiled with PDP11=GENERIC, allowing the system to support most of the hardware on any supported processor. The definitions for the optional features of the system are in **localopts.h**. Finally, the files **param.c** and **param.h** contain the tunable sizes and parameters. These are mostly dependent on the definitions of PDP11 and MAXUSERS (in the Makefile). **Param.c** contains most of the commonly-changed parameters, so that only this file need be recompiled to retune the system. Also, because these parameters are now in global variables, system utilities may easily determine the current values by examining the running system.

5) The Makefile contains the default compilation and load rules for the type of kernel being made (overlaid or not overlaid). It also contains the specification of an editor script that implements in-line expansions of calls to spl, depending on the instruction set available. The makefile may need editing to change the overlay structure or to include optional device drivers in the load rules. MAXUSERS is defined here and used in **param.c** to gauge the sizes of data structures.

In order to add new files or device drivers to the system, it is necessary to explicitly add them to the Makefile load rules, to its extension Depend (used in the "make depend" command to rebuild the Makefile dependency rules), to the configuration file **c.c** and optionally to *autoconfig*(8) and *config*(8) or **l.s**.

Header files

Many new files have been added for use in device drivers. They contain definitions of the device structure and mnemonics used in referencing registers and printing diagnostics. Most files have been reorganized slightly to improve modularity or readability.

acct.h — The UCB_XACC option has been separated into UCB_LOGIN and UCB_SUBM.

buf.h — Unused flags have been deleted and the others compacted. Two flags have been added. B_RH70 indicates that a device is on an RH70 controller. B_UBAREMAP indicates that the buffer's address is being interpreted as UNIBUS virtual, not physical.

conf.h — A *d_root* field has been added to the *bdevsw* structure. The unused fields *l_rend* and *l_meta* have been deleted from the *linesw* structure. *L_rint* has been renamed *l_input*. *L_start* has been deleted and a new field, *l_output* added for *uprintf*. See Appendix C.

cpu.h New file. Contains mnemonics for fields in the cache and memory control registers of various processors.

dkbad.h New file. Contains mnemonics and structures used to implement DEC standard 144 bad sector forwarding.

filsys.h Two fields in the *filsys* structure, *s_fname* and *s_fpack*, have been replaced by *s_fsmnt*. The new field is used by the kernel to print diagnostics and by *fsck*(8).

inline.h New file. Definitions of inline expansions and macro replacements designed to speed up file system accesses at the cost of code expansion.

lstat.h Renamed *qstat.h*. The structure previously names *lstat* is now names *qstat* and all structure fields previously named *ls_** have been renamed *qs_**.

koverlay.h New file. Contains definitions relating to kernel text overlays. Both nonseparate I/D (0430) and separate I/D (0431) kernels can be overlaid. Most of the information in this file cannot be changed easily. It is provided to clarify the way kernel overlays work.

mtio.h An *mt_type* field has been added to the *mtget* structure. Tape drivers may be interrogated to determine formatter type. See *mt*(4).

param.h Many configuration constants (e.g. NINODE, NPROC) have moved from here to param.c and are referenced by global variables rather than manifest constants. Thus only one file need be recompiled to change them.

proc.h Numerous changes have been made to support job control and *vfork*s. The *xproc* structure is in a union in the *proc* structure so that it is easily possible to determine which fields are overlaid.

qstat.h Used to be called *lstat.h*. Contains declarations for the *qstat* and *qfstat* system calls (for quotas).

reboot.h New file. Contains options for the *reboot* system call.

reg.h The (unused) definition of ROV has been deleted.

seg.h New macros and definitions have been added to support the remapping of kernel data to access buffers and clists. Changes have been made to allow dynamic support of the ENABLE/34.

trap.h New file. Used in l.s, mch.s, and trap.c to encode trap types mnemonically.

tty.h Contains a macro for *lookc* if UCB_NTTY is defined and UCB_CLIST is not defined.

types.h More typedefs have been added.

uba.h New file. Most UNIBUS map specific structures and macros are collected here.

user.h Numerous changes have been made to support job control and *vfork*s.

vcmd.h New file. Contains commands used by the vp driver and user *ioctl* definitions.

System files: sys/sys

Major changes have taken place to support job control and *vfork*s. The *file, proc,* and *text* tables have been moved to the end of kernel data space (possibly in the region into which buffers and clists are mapped) and thus are not necessarily accessible at interrupt time; those functions that need to access these tables or the *u.* from interrupt level (currently *clock, gsignal,* and *wakeup*) must save and restore kernel mapping registers.

Inclusion of both the multiplexer and floating point support is conditional, reducing the size of systems that do not require them. Some consistency checks that we consider extremely unlikely to fail, and the accompanying *panics*, are uniformly conditional on the definition of DIAGNOSTIC. Calls to *splN* (where *N* is 0, ..., 7) that do not require the previous priority to be returned have been changed to *_splN* and are expanded in-line by editing the compiler's output.

acct.c The *sysphys* routine has been moved from here to machdep.c.

alloc.c File system error messages are identified by file system name rather than major/minor device number. They are printed directly on a user's terminal if that user causes a file system to run out of free space. *Getfs* no longer *panics* if it cannot find a device in the mount table. Callers of *getfs* have been modified to check for a NULL return value. This, together with a change to pipe.c, avoids a panic if *pipedev* is a file system that is not currently mounted.

clock.c *Clock* has been modified to use the new remapping protocols. Disk monitoring has been simplified and can monitor more (or fewer) than three disks. Free memory averaging is calculated in kilobytes, avoiding overflow.

enable34.c New file. Contains support routines for the ENABLE/34. Two routines, *fiobyte* and *fioword* are used to help solve the problem of probing the I/O page on machines with ENABLE/34 boards. Wherever *fuibyte* and *fuiword* would be used to probe a location *possibly* on the I/O page, these routines should be used instead.

fakemx.c This file is no longer necessary and has been deleted.

flo.c *Falloc* uses the *tablefull* routine. A bug in the *access* system call with the UCB_GRPMAST option has been fixed.

iget.c After reading blocks of inodes, both the error flag and the residual count are checked. This avoids destroying whole blocks of inodes on failure. The residual count is also checked in other places in the kernel (*bmap*, etc.). If an error occurs in *iget*, *iput* is not called for an invalid inode. *Iget* uses the *tablefull* routine.

l.s Both l.s and the old l40.s are merged into this file. The code is preprocessed with *cpp*, allowing consistency with C files for conditional compilation.

machdep.c A *boot* function has been added to cause the system to reboot itself and (optionally) take a crash dump automatically. The type of reboot is passed to /etc/init as an argument. *Mapalloc* and *mapfree* use a resource map to dynamically allocate sections of the UNIBUS map. *Mapalloc* translates physical addresses in buffer headers for cache buffers to UNIBUS addresses for transfers on UNIBUS devices. *Mapalloc* is thus called for both buffered and raw transfers now. *Ubinit* initializes the UNIBUS map and the resource map describing it. *Mapin* and *mapout* no longer run at elevated priorities to block interrupts. *Mapout* is eliminated if the kernel data segment is sufficiently small.

A new function, *dorti*, which is used by the new signal mechanisms has been added.

Buffer space is uniformly *malloc*ed in *startup* rather than in *start* (mch.s) The same is true for clists if UCB_CLIST is defined.

On machines without UNIBUS maps, no attempt is made to detect memory past 0760000, avoiding crashes when device registers are found at this address.

Clkstart calls *fioword* to probe for the line clock register. It is not a panic if no clock register is found since 11/23s may not have one; a message is printed in this case.

main.c The name of the root file system ("/") is copied into its superblock so that the name will be available for error messages (e.g. if the root file system becomes full).

malloc.c All addresses and sizes in **malloc.c** have been typedeffed and are unsigned. This makes it possible to use more than two megabytes of memory. A new function, *malloc3*, efficiently allocates memory for scatter loading, minimizing the cost of failing. *Mfree* contains many more consistency checks. Resource maps have a new structure that includes a limit. *Mfree* prints a console error message when it must discard a piece of a map because of fragmentation instead of overrunning the map or *panic*ing. When *malloc* cannot allocate enough swap space, it frees the swap space belonging to saved text segments, possibly avoiding panics caused by running out of swap space.

mch.s Both m40.s and the old mch.s have been merged into this file. The C preprocessor is used to produce the right code for different CPUs, including GENERIC. It is able to reboot after power failures if the contents of memory are intact.

Copyseg and *clearseg* have been converted to *copy* and *clear* respectively. They take an additional argument, a count of the number of clicks to copy or clear. They remap the kernel to access the source and target more efficiently. If real-time support is enabled, both are preemptible. A new routine, *copyu*, is available to copy the *u.* in non-preemptible mode.

Most *spl* calls are now done in-line; the old priorities are saved and restored as bytes (to allow the use of *mfps/mtps* instructions where available). Kernel red stack violations are detected, allowing normal *panics*.

System call traps are handled separately from other processor traps. This results in a 22% decrease in system call overhead. Emulator traps (used in automatic text overlays) are also handled separately from general traps. This decreases overlay switch overhead by 45%. On machines without hardware floating point, a fast illegal instruction trap routine reduces system overhead for interpreted floating point by 90%.

The kernel overlay support has been changed to use new, smaller subroutine entries ("thunks") in the base segment that are compatible with the loader used for user-level overlaid programs. The management of the kernel stack in the trap/interrupt code is simpler and faster.

The kernel text relocation that was done in mch.s if UCB_CLIST or UCB_BUFOUT were defined is no longer necessary and has been replaced by calls to *malloc* in *startup*.

nami.c File names are not allowed to contain characters with the parity bit (0200) set. File name comparisons stop at the first null. A bug that caused permissions to be checked incorrectly when searching to ".." from the root of a mounted filesystem has been fixed.

pipe.c Allocates inodes for pipes on the root device if *ialloc*s on *pipedev* fail. Inodes for pipes are marked for special handling.

prf.c *Panic* causes the system to reboot. A function, *uprintf*, has been added to print error messages on the terminal of the user causing the error rather than the console. *Printf* no longer uses recursion. It supports a %c format to print a single character, a %b format used to print register

values mnemonically, and a %X format for long hexadecimal. *Prdev* has been eliminated. *Deverror* is included only if UCB_DEVERR is undefined.

The routines *prdev* and *deverror*, that printed diagnostics that were difficult to interpret, are replaced by *harderr*, that begins a message about an unrecoverable device error, and the %b format mentioned above. *Tablefull* is a new function used to report that a table is full.

prim.c Uses new mapping protocols for *CMAPIN* and *CMAPOUT*. *Getw* has been discarded. *Putw* is included only if needed for the multiplexer driver. *Cpaddr* has been deleted. It is now a macro in dh.c. Other routines that are used only by the dh driver are eliminated if there are no dh's on a system. *Lookc* is eliminated (replaced by a macro) if UCB_CLIST is not defined.

rdwri.c Inodes allocated for pipes receive special handling: *writei* always uses *bdwrite* and *readi* cancels the disk write if it has not yet occurred. This results in a large improvement in pipe throughput, especially if the UCB_FSFIX option is in use (for more robust file systems).

sig.c This is now a dummy file that includes either sigjcl.c or signojcl.c depending on whether MENLO_JCL is defined.

sigjcl.c A new file that supports the signal mechanisms necessary for job control. The changes listed under *signojcl.c* are also included.

signojcl.c Used to be called sig.c. A race condition that occasionally caused ignored signals to generate bus errors has been fixed. *Ptrace* supports overlay changes, allowing breakpointing of overlaid subprocesses. If floating point arithmetic is being simulated by catching illegal instruction traps, traced subprocesses are allowed to process the signal normally without stopping. Stack growth is rounded to 8K boundaries, to allow the maximum theoretical stack size.

slp.c There are major changes in the *sleep/wakeup* mechanism for process control. Swapped processes are no longer kept on the run queue. *Newproc* has been modified to allow *vfork*s. The scheduling algorithm has been modified to avoid deadlocks possible with *vfork*. Processes are scatter loaded in three pieces (data, stack and *u.* area; text is handled separately), with changes in *newproc*, *expand* and *swapin*.

The unused routine *dequeue* has been removed.

subr.c *Bcopy* may now be called with a count of 0.

sys1.c *Fork* has been modified to allow *vfork*s and uses the *tablefull* routine. Support has been added for *wait2*, used in job control. *Bdwrite* is used instead of *bawrite* when copying out argument lists in *exece*, in an attempt to avoid disk I/O. A pointer to the last used proc table slot, *lastproc*, is used to shorten searches for processes. A message is printed if /etc/init cannot be executed.

sys3.c *Smount* copies the mounted file system's name (e.g. "/usr") into the s_fsmnt field of the superblock. The in-address-space buffers (abuffers) have been removed, and the superblocks of mounted file systems are in the mount table itself.

sys4.c The mechanism for sending signals to all processes has been changed so that the process broadcasting the signal does not receive it itself. This allows *reboot*(8) to shut down the system cleanly before rebooting.

The *#ifdef* for UCB_STICKYDIR has been removed. This is now standard. *Setpgrp* is included to support job control. A bug in *utime* has

been fixed.

syslocal.c The old *setpgrp* is replaced by the job control version. *Chfile* and *iwait* have been removed. A new system call, *vhangup*, is used by *init* to revoke access to terminals after logouts. Another new system call, *ucall*, allows *autoconfig*(8) to call internal kernel routines. Support for *qstat*, and *qfstat* (formerly *lstat* and *lfstat* respectively) is conditional on UCB_QUOTAS.

text.c *Xswap* has been modified for scatter loading. *Xumount* frees all saved text segments if called with argument NODEV. *Malloc* uses this to attempt to avoid *panics* when swap space is exhausted. *Xalloc* uses the *tablefull* routine.

trap.c *Trap* no longer handles system calls. Instead, a new routine, *syscall*, is called from mch.s when a system call trap occurs. *Trap* saves the previous kernel mapping on kernel faults.

ureg.c A new routine, *choverlay*, has been added to change overlays for user processes. It is called from mch.s when an overlay switch trap occurs. The units of the variables describing the overlay region (ovbase and dbase) have changed. Segmentation register prototypes are no longer maintained for the overlay region, necessitating a call to *choverlay* from *sureg*. *Estabur* and *sureg* support scatter loading. A bug has been fixed that caused overlaid processes to fail when the base segment length was a multiple of 8192. On machines without separate I/D space, *estabur* is simplified.

Device support: sys/dev

All of the drivers have been modified to support autoconfiguration. They have attach routines to record the csr addresses after the device has been probed by *autoconfig*(8). Appendix E describes the strategy. Drivers with attach routines properly reject attempts to access nonexistent controllers (instead of causing a crash). Each device driver has a corresponding header file indicating the number of such devices present and other configuration dependent options.

Devices that do DMA on machines with UNIBUS maps must ensure that their data areas are accessible through the UNIBUS map; UNIBUS addresses are not necessarily the same as physical addresses. see Appendix B. Only buffers and clists are statically mapped. It is possible to map in out-of-address space data at interrupt level (this was previously risky) provided the previous map is saved and restored; a mechanism is provided for this, as described in Appendix A. The structure of the line switch has been reorganized and the protocol to be used in opening a device and setting up a line discipline is well defined. See Appendix C.

Disks that are potentially RH70 MASSBUS disks have been provided with attach routines that detect RH70s, as well as root attach routines that force attachment before autoconfiguration occurs. Some disk drivers have been provided with crash dump routines. See *rmdump* in rm.c or *hkdump* in hk.c for examples.

The format of device option flags is now consistent. Optional device ioctls are enabled by XX_IOCTL (e.g. DH_IOCTL). Optional watchdog timers are enabled by XX_TIMER (e.g. TM_TIMER). The *dh* (respectively *dz*) driver, which is capable of managing the input siilo to reduce interrupts, does so if DH_SILO (respectively DZ_SILO) is defined. The disk cache monitoring numbers used by *iostat*(8), formerly called DK_N, have been renamed XX_DKN (e.g. HP_DKN) so that they can be placed in the header files.

All drivers use include files to define the device structures and register constants. The drivers themselves uniformly use mnemonics rather than magic numbers in device registers and error messages. Initialized device register addresses and disk driver partition tables reside in ioconf.c.

blo.c *Iodone* reverses the translation of buffer addresses (done by *mapalloc*) from physical to UNIBUS virtual when doing block I/O on UNIBUS disks. *Bwrite* now correctly supports the B_AGE flag on asynchronous writes. A portion of the disk monitoring code that was of questionable usefulness has been discarded. The *physio* subroutine has been divided into separate routines, allowing use of *bphysio* by drivers that allow byte-oriented rather than word-oriented transfers or don't use buffer headers.

bk.c The Berknet line discipline has been changed to use dedicated buffers instead of abuffers. It is still untested.

dh.c Changed to use the new UNIBUS map location of clists. Ioctls for setting and clearing *break* and *dtr* have been added. If DH_SOFTCAR is defined, modem control is ignored for lines whose minor device number is greater than or equal to 0200. Dhdm.c is now part of dh.c; the appropriate dm support is included only if needed.

dhdm.c This is now part of dh.c.

dhfdm.c This file is no longer necessary and has been deleted.

dvhp.c This driver is simplified if there is only one drive, as no seek is needed before a transfer. Error correction code has been added.

dz.c Optionally uses the dz silo. Ioctls for setting and clearing *break* and *dtr* are available. If DZ_SOFTCAR is defined, modem control is ignored for lines whose minor device number is greater than or equal to 0200. Pseudo-dma has been implemented.

hk.c New version of the RK06/7 driver. Now performs disk sorts, ECC corrections, and DEC standard 144 bad sector forwarding. A dump routine has been added.

hp.c This driver is simplified if there is only one drive, since no search is needed before a transfer. Error correction code works with mapped buffers and 1024 byte blocks. The driver waits for Drive Ready when doing positioning commands. A dump routine has been added. A preliminary, lightly tested version of DEC standard 144 bad sector forwarding has been added.

ht.c Tape ioctls are supported. Uses *bphysio* for byte-oriented transfers. *Clrbuf* is no longer called from interrupt level.

kl.c *Putchar* has been modified to support *uprintf.*

mem.c Some unneeded *spl*s have been deleted. Routines used to read and write memory set page protections correctly.

ml.c New file. A driver for the DEC ML11 solid state disk courtesy of the DEC UNIX Engineering Group.

mux.c Dropped from this distribution.

rf.c New version of an old driver missing from 2.8BSD.

rk.c Properly recovers the residual byte count at the end of a transfer.

rl.c Properly recovers the residual byte count at the end of a transfer.

rm.c This driver is simplified if there is only one drive; the *rmustart* routine is merged with *rmstart,* and no search is needed before a transfer. Error correction code works with mapped buffers and 1024 byte blocks. The software simulation of the current cylinder register has been fixed. The driver waits for Drive Ready when doing positioning commands. A dump routine has been added. A preliminary, lightly tested version of DEC standard 144 bad sector forwarding has been added.

rp.c Properly recovers the residual byte count at the end of a transfer.

rx2.c New file. A driver for the DEC RX211 floppy disk controller courtesy of the DEC UNIX Engineering Group.

rx3.c New file. A driver for the DSD480 floppy disk controller courtesy of Tektronix.

tm.c Uses *bphysio* for byte-oriented transfers. *Clrbuf* is no longer called from interrupt level. Contains code for an optional watchdog timer. Checks for density changes in mid-tape.

ts.c Tape ioctls are supported. Uses *bphysio* for byte-oriented transfers. *Clrbuf* is no longer called from interrupt level.

tty.c The *ttioctl* subroutine calls the line discipline's ioctl before any other processing. *Ttioctl* has also been changed to eliminate code for the old line discipline if it is not present, and when changing disciplines it checks that the new discipline is supported. These changes allow the old line discipline to be omitted. It is possible to flush either the input or output queues (or both) using TIOCFLUSH.

ttynew.c Tandem mode is supported with raw mode in the new tty driver. The t_char field is no longer disturbed by flow control in tandem mode. Backslashes are no longer printed before capital letters on upper-case-only terminals.

xp.c This driver (which supports an assortment of RP04/05/06, RM02/03/05, Diva and other disks) now is able to manage more than one controller. The probe routine is optional if the drive and controller structures are initialized. It is simplified if there is only one drive; no search is needed before a transfer. Error correction code works with mapped buffers and 1024 byte blocks. The driver waits for Drive Ready when doing positioning commands. A dump routine has been added. A preliminary, lightly tested version of DEC standard 144 bad sector forwarding has been added.

Appendix A: Kernel Data Mapping Protocols

1. Introduction

These protocols ultimately address the question of how to "expand" the kernel's data space beyond the severe limitations imposed by the PDP-11 hardware. This concern about methods of expanding kernel data space stems from the desirability of retaining large system buffer pools and clist areas despite hardware limitations. We do this by keeping certain data objects resident in core but without guaranteeing that they will be accessible through kernel virtual data space at all times. In this way the same virtual address range can be used for several different objects.

1.1. History

The original Berkeley PDP-11 kernel distribution (2.8BSD) provided the ability to move buffers and clists out of kernel data space. Buffers were accessed by mapping them in through KDSA5. A side effect was that the data that normally resided there were unavailable until buffers were mapped out again. Clists were mapped in through KDSA1 with the same side effect.

Because of this restriction, and the possibility of interrupts at any time, sections in which a kernel data register was repointed generally had to be protected by *spl6*()/*splx*() pairs. (The exception is that *spl*s were unnecessary for buffer mapping if KDSA5 was used only for that purpose, and this was not done from interrupt level.) This inevitably led to increased interrupt latency and sometimes caused the system clock to lose time perceptibly.

It is not at all clear why these registers were special. They were chosen after careful examination of the system namelist. On our 11/70s, the inode table used all virtual addresses referenced through KDSA1 and it was known that no part of the kernel required simultaneous access to clists and inodes. Similarly, it was observed that data referenced through KDSA5 typically consisted of tty structures and the kernel did not require simultaneous access to tty structures and buffers.

It should be obvious how vulnerable this method is to even the most trivial changes such as system load order or table sizes. Clearly something better was needed.

1.2. 2.9BSD Methods

We chose four goals for our new remapping protocols:

[1] They must be fast. Interrupt latency should not be increased by elevating the processor priority.

[2] They should be flexible, allowing objects other than buffers and clists to be remapped easily.

[3] Interrupt service routines should not be slowed unnecessarily by requiring that the map be changed on all interrupts.

[4] There must be a well-defined class of objects that the remapping will make inaccessible. Furthermore, any section of code that requires access to one of these objects during interrupt processing must itself ensure that the object is mapped in.

The implementation we chose uses KDSA5 as the primary mapping register. The only normally-resident objects allowed in this region (0120000 to 0140000) are the *proc*, *file*, and *text* tables. These objects were chosen because they are rarely accessed from interrupt level. If kernel data space is small enough that these tables end before this region, the code can be further simplified by defining the conditional-compilation flag NOKA5. In general, kernel functions are able to map in external data at will, with the caveat that interrupt routines must save the previous map (which may already point at some mapped-in object).

To make *copy* (previously *copyseg*) as fast as possible, yet interruptible, we also allow it to use KDSA6 as a mapping register. This makes the normal kernel stack (which lies in the region addressed by KDSA6) inaccessible, so the kernel uses a temporary stack while in *copy*.

Most of the segmentation map switching is done by macros for speed; some of the macros test whether any work need be done before calling a subroutine. The data structures and macros used in this scheme are in the include file *seg.h*, with the subroutines in *machdep.c*. These macros must be used for all kernel remapping or races will ensue (because the order in which registers are set is critical to the protocol).

1.2.1. Top Level Protocol

A global prototype page address/descriptor pair is maintained (if necessary) for virtual addresses from 0120000 to 0140000. It is initialized in *startup*. KDSA5 may be repointed to access other objects from the top level provided that the normal mapping is restored before the next context switch. The contents of KDSA5/KDSD5 are changed by the macro call

mapseg5(addr, desc);

where *addr* is the new value for KDSA5 and *desc* is the new value for KDSD5. The default mapping for this page is restored by the macro call

normalseg5();

The *mapin* and *mapout* functions use this method to provide access to a mapped buffer.

Unless the kernel data map has been explicitly reset by *mapin* or *mapseg5*, the *proc*, *file*, and *text* tables are guaranteed to be mapped in when the kernel is not at interrupt level.

1.2.2. Interrupt Level Protocol

Interrupt-level routines may not assume that the range controlled by KDSA5 or KDSA6 contains valid data unless the map is explicitly set to either the normal state (for the *proc*, *text* or *file* tables, or for the *u.*) or to map external data.

Interrupt routines that wish to repoint KDSA5 must first save the current contents of KDSA5 and KDSD5 in a local variable by

segm saveregs;
saveseg5(saveregs);

before changing their contents with *mapseg5*. Before returning, the old contents must be restored by the call

restorseg5(saveregs);

This method is used by *getc* and *putc* to access the clist area.

Note that *mapin* does not save the current map in this way. To use *mapin* and *mapout* from interrupt level, it is necessary to save the map with *saveseg5* before calling *mapin*, and then restore it with *restorseg5* after the last *mapout*.

If an interrupt routine must access either the *u.* or any of the tables, it must save the previous PARs and PDRs for pages 5 and 6 in a local variable and set the map to the normal state using

mapinfo map;
savemap(map);

and restore the old contents with

restormap(map);

This mechanism is used by *gsignal* and *wakeup,* which are frequently called from interrupt level and must access the *proc* table, and by *clock,* which needs access to the *proc* table and the *user* structure. It is also used in *trap,* which saves the map data in the global map *kernelmap* on kernel-mode traps for potential use in debugging.

Appendix B: UNIBUS Map Protocols

2. Introduction

UNIX as distributed by Bell Labs and in previous Berkeley releases made some tacit assumptions about the arrangement of kernel data space and the use of the UNIBUS map (or machines with 22-bit addressing):

- All kernel data space was statically covered by some portion of the UNIBUS map. This included mapped out objects such as buffers and clists. Kernel virtual data space addresses needed no conversion to UNIBUS or physical addresses. Thus no special action was taken on, for example, DMA transfers from kernel data space to ensure that the source or target area was accessible through the UNIBUS map.
- The remaining portion of the UNIBUS map was dedicated to only one I/O request at a time. Thus a fixed portion of the UNIBUS map was used for each physical I/O request.

Although these assumptions did result in much simpler code, they had the unfortunate side effect of degrading system performance. Two swaps could not occur simultaneously. When a slow device such as a tape drive was used for physical I/O, all other physical I/O suffered severely. This was most noticeable when file system dumps were occurring. It also made the use of raw I/O for real-time data acquisition impossible.

2.1. 2.9BSD Methods

The solution is to manage the UNIBUS map with a resource map, allocating and freeing groups of registers as required by the size of the I/O request. This has already been implemented independently at some sites. Our code is modeled after several of these.

In an effort to have as many UNIBUS map registers as possible available for allocation, only the clist area and buffer pool have statically allocated UNIBUS map registers. The clist area is mapped through UNIBUS register 0. It may therefore be at most 8192 bytes long, and begins at UNIBUS virtual address 0. The global variable *clstaddr* contains the UNIBUS address (in bytes) of clists (even if a UNIBUS map is not present). The appropriate number of registers is dedicated to the buffer pool at boot time and the rest are made available for allocation. When there is a UNIBUS map, the buffers begin at UNIBUS byte address BUF_UBADDR, whereas their physical address (in clicks) is *bpaddr*.

Routines that manipulate the UNIBUS map must be prepared to be called even if no UNIBUS map exists. They should check the boolean variable *ubmap*, which is nonzero if a UNIBUS map is present. For convenience, several useful macros have also been provided. See the include file *uba.h*.

The code for block I/O dynamically supports both MASSBUS and UNIBUS controllers. A buffer header associated with the buffer cache used for block I/O normally contains the physical address of the buffer area. This is translated into a UNIBUS address before beginning the I/O operation if the device does not use 22-bit addressing. This translation is performed by *mapalloc*; thus, UNIBUS disk and tape drivers should call *mapalloc* for both raw operations (B_PHYS set) and those in the buffer cache. While a buffer header contains the UNIBUS virtual address of the buffer area instead of the physical address, the B_UBAREMAP flag is set in its *b_flags* field. After the transfer is finished, *iodone* restores the physical address in the buffer header. Drivers for disks that may be either MASSBUS or UNIBUS generally set the B_RH70 flag in the *b_flags* of their *devtab* structures if they are 22-bit MASSBUS devices and test it before calling *mapalloc*.

3. Introduction

There have been several changes in the kernel terminal-handling routines. The initial incentive for these changes was to allow the old tty discipline to be removed. This required that line disciplines be symmetric and equivalent. Previously, line discipline 0 (the old tty driver) was treated specially and was assumed to exist.

3.1. Ttyopen and Ttyclose

The first group of changes is in the open and close sections. The routines *ttyopen* and *ttyclose* are no longer part of any discipline, but do the necessary initialization at the first open and the breakdown at the final close. They call the line discipline-specific open or close routine, and all the drivers (dh, dz, kl etc.) need do is call *ttyopen* and *ttyclose* from their open and close routines.

3.2. Ioctl Protocols

The second set of changes is in the ioctl-handling sections. The line disciplines are given the opportunity to reject or modify any *ioctl* call, or to do it themselves, before the common code is reached. Again, all the work is done by the discipline-independent routine, *ttioctl,* which calls the line discipline's ioctl routine. The device drivers thus call only *ttioctl.* There are three possible return conditions from *ttioctl*:

- a command is returned that the device driver is expected to execute
- 0 is returned with *u.u_error* clear, meaning that the command completed successfully
- 0 is returned with *u.u_error* set, meaning that the command completed abnormally

The typical device driver ioctl routine will thus look like this:

```
switch (ttioctl(tp, cmd, addr, flag)) {
	case TIOCSETP:
	case TIOCSETN:
		setparam(unit);
		break;
	case other_known_command:
		implement the command;
		break;
	default:
		u.u_error = ENOTTY;
	case 0:
		break;
}
```

3.3. Line Switch Changes

There are a few other differences in the terminal handlers from previous systems. The line discipline switch is no longer optional (the defined constant UCB_LDISC is gone). The linesw can have unused discipline entries in it, so that line discipline numbering is independent of the disciplines supported at any time; unused disciplines are marked by using *nodev* as their open routines, thus preventing entrance into them. This necessitates a new defined constant, DFLT_LDISC, which is the line discipline that device drivers should set on initial

open. Finally, the line discipline switch itself has been reorganized, with three entries being deleted and one field added. The previously-unused *l_rend* and *l_meta* pointers have been removed, and calls to *l_start* have been replaced with calls to *ttstart*. The *l_rint* entry has been renamed *l_input* and an *l_output* pointer has been added for the use of *uprintf*.

Appendix D: Vfork Implementation Notes

The kernel changes for the *vfork* system call are major and deserve a few notes. Processes are no longer in one piece, but instead the user structure, data segment, and stack segment are separate. They are located at *p->p_addr*, *p->p_daddr*, and *p->p_saddr* respectively (where *p* is a pointer to a proc entry) and their sizes are USIZE, *p->p_dsiz* and *p->p_ssiz*. The latter two are copies of the entries in the user structure. All segments are swapped if any are, and there is a new routine, *malloc3*, to allocate memory or swap for all three segments at once. When a *vfork* occurs, the *u.* is copied, and the data and stack are passed to the child. The parent sleeps until the child calls *exec* or *exit*. At that time, the child locks itself in core and waits for the parent to reclaim the data and stack.

The major advantages of these changes are the efficiency of avoiding the copy in *fork*, and more efficient utilization of memory, as processes are in smaller segments. The disadvantage is that swaps require three separate transfers in each direction. Except on heavily loaded systems with small main memory, the result should be a net gain. There is a potential for deadlock since the child must lock itself into core; this can only be a problem with small memories when the parent has been swapped out. To help avoid problems, the swapping algorithm has been changed to swap in the parent process in a vfork before any others.

Appendix E: Autoconfiguration

The kernel changes to add autoconfiguration are fairly small. The most global change is that device CSR addresses and interrupt vectors must be initialized only for disk drivers which service root devices. Most of the work of autoconfiguration is done in user mode by *autoconfig*(8). It reads the device table */etc/dtab*, then verifies the CSR address by reading from it (through /dev/kmem). If the CSR is present, *autoconfig* then tries to make the device interrupt in order to check that the vector specified is correct. To facilitate this check, l.s has two interrupt catchers, *CBAD* and *CGOOD*, that set the global variable *_conf_int* to −1 and 1 respectively when called. *Autoconfig* sets all unused vectors to *CBAD*, then sets the expected vector to *CGOOD*. After the probe, *autoconfig* checks the contents of *_conf_int* to see whether the device interrupted and whether it was through the expected vector. If everything is correct to this point, *autoconfig* calls the device driver's attach routine with the unit number and address, then sets up the interrupt vector.

The kernel support for autoconfiguration consists of two parts. The first includes the interrupt catchers in l.s and a new routine in syslocal.c that allows *autoconfig* to call the driver attach routines. This new system call, *ucall* (see *ucall*(2)), calls a specified kernel routine (by address) at a specified priority with two user-supplied arguments. The other group of changes is in the drivers. Most drivers have new attach routines which simply place the address specified into their address arrays, checking that the unit number is in range. Device open and/or strategy routines have been modified to test that the device address has been set before allowing the open, read, or write to succeed. Drivers that need to probe the hardware to test its type may do that as well in the attach routine. The drivers that handle both MASSBUS and UNIBUS devices check for bus address extension registers at this time. A new routine, *fioword*, is provided to read a word from the I/O page, returning -1 if the address does not exist. Because the disks must be attached before *autoconfig* runs if they are to be used for root file systems, their addresses and vectors are still initialized. A new entry in the block device switch, *d_root*, is used at boot time to call driver routines which disk drivers may use to attach all known devices before *iinit*. This allows them to determine controller and drive types. Drivers currently fall into three classes: UNIBUS only disks, MASSBUS/UNIBUS disks, and others. Prototypes of the attach and *d_root* routines for each class follow.

The probe routines that are used to make the devices interrupt may be either in *autoconfig* or in the kernel. If the kernel has a probe routine, that will be used, otherwise *autoconfig* will use its own probe. This mechanism is provided because it may be difficult to address some devices properly by reading and writing /dev/kmem. All current probe routines are internal to *autoconfig*.

Device drivers that have no *attach* routines are ignored by *autoconfig*. Old drivers that have not been converted to use autoconfiguration will thus work properly.

```
/*
 *	Example 1:  autoconfiguration prototype for devices other
 *	than disks.  Xxattach will be called by autoconfig(8).
 */

xxattach(addr, unit)
struct xxdevice *addr;
{
	if ((unsigned) unit >= NXX)
		return(0);
	xx_addr[unit] = addr;
	return(1);
}

/*ARGSUSED*/
xxopen(dev, flag)
dev_t dev;
int	flag;
{
	register int unit = XXUNIT(dev);

	if (xx_addr[unit] == (struct xxdevice *) NULL) {
		u.u_error = ENXIO;
		return;
	}
	if (unit >= NXX) {
		u.u_error = EINVAL;
		return;
	}
	.
	.
	.
}
```

```
/*
 *	Example 2:  autoconfiguration prototype for UNIBUS disks.
 *	Xxattach will be called by autoconfig(8).
 */

xxattach(addr, unit)
struct xxdevice *addr;
{
	if (unit != 0)
		return(0);
	XXADDR = addr;
	return(1);
}

xxstrategy(bp)
register struct buf *bp;
{
	if (XXADDR == (struct xxdevice *) NULL) {
		bp->b_error = ENXIO;
		goto errexit;
	}
	if (bp->b_blkno >= NXXBLK) {
		bp->b_error = EINVAL;
errexit:
		bp->b_flags |= B_ERROR;
		iodone(bp);
		return;
	}
	.
	.
	.
}
```

```
/*
 *	Example 3: autoconfiguration prototype for disks
 *	possibly on the MASSBUS. Xxroot will be called
 *	from binit (main.c).
 */

void
xxroot()
{
	xxattach(XXADDR, 0);
}

xxattach(addr, unit)
register struct xxdevice *addr;
{
	if (unit != 0)
		return(0);
	if ((addr != (struct xxdevice *) NULL) && (fioword(addr) != -1)) {
		XXADDR = addr;
#if	PDP11 == 70 || PDP11 == GENERIC
		if (fioword(&(addr->xxbae)) != -1)
			xxtab.b_flags |= B_RH70;
#endif
		return(1);
	}
	XXADDR = (struct xxdevice *) NULL;
	return(0);
}

xxstrategy(bp)
register struct buf *bp;
{
	register unit;
	long	bn;

	if (XXADDR == (struct xxdevice *) NULL) {
		bp->b_error = ENXIO;
		goto errexit;
	}

	unit = minor(bp->b_dev) & 077;
	if (unit >= (NXX << 3) || bp->b_blkno < 0 ||
	    (bn = dkblock(bp)) + ((bp->b_bcount + 511) >> 9)
	    > xx_sizes[unit & 07].nblocks) {
		bp->b_error = EINVAL;
errexit:
		bp->b_flags |= B_ERROR;
		iodone(bp);
		return;
	}
	.
	.
	.
}
```

I N G R E S

VERSION 6.3
R E F E R E N C E   M A N U A L

4/2/81

by

John Woodfill
Nick Whyte
Mike Ubell
Polly Siegal
Dan Ries
Marc Meyer
Paula Hawthorn
Bob Epstein
Rick Berman
Eric Allman

This manual is a reference manual for the INGRES data base system. It documents the use of INGRES in a very terse manner. To learn how to use INGRES, refer to the document called "A Tutorial on INGRES".

The INGRES reference manual is subdivided into four parts:

- Quel describes the commands and features which are used inside of INGRES.
- Unix describes the INGRES programs which are executable as UNIX commands.
- Files describes some of the important files used by INGRES.
- Error lists all the user generatable error messages along with some elaboration as to what they mean or what we think they mean.

Each entry in this manual has one or more of the following sections:

NAME section
: This section repeats the name of the entry and gives an indication of its purpose.

SYNOPSIS section
: This section indicates the form of the command (statement). The conventions which are used are as follows:

  Bold face names are used to indicate reserved keywords.
  Lower case words indicate generic types of information which must be supplied by the user; legal values for these names are described in the DESCRIPTION section.
  Square brakets ([]) indicate that the enclosed item is optional.
  Braces ({}) indicate an optional item which may be repeated. In some cases they indicate simple (non-repeated) grouping; the usage should be clear from context.

  When these conventions are insufficient to fully specify the legal format of a command a more general form is given and the allowable subsets are specified in the DESCRIPTION section.

DESCRIPTION section
: This section gives a detailed description of the entry with references to the generic names used in the SYNOPSIS section.

EXAMPLE section
: This section gives one or more examples of the use of the entry. Most of these examples are based on the following relations:

  emp(name,sal,mgr,bdate)
  and
  newemp(name,sal,age)
  and
  parts(pnum, pname, color, weight, qoh)

SEE ALSO section
: This section gives the names of entries in the manual which are closely related to the current entry or which are referenced in the description of the current entry.

BUGS section
: This section indicates known bugs or deficiencies in the command.

To start using INGRES you must be entered as an INGRES user; this is done by the INGRES administrator who will enter you in the "users" file (see users(files)). To start using ingres see the section on ingres(unix), quel(quel), and monitor(quel).

ACKNOWLEDGEMENTS

We would like to acknowledge the people who have worked on INGRES in the past:

William Zook
Karel Youssefi
Peter Rubinstein
Peter Kreps
Gerald Held
James Ford

FOOTNOTE

UNIX is a trademark of Bell Laboratories.

APPEND(QUEL) – append tuples to a relation
    append [**to**] relname (target_list) [**where** qual]

COPY(QUEL) – copy data into/from a relation from/into a UNIX file.
    copy relname (domname = format {, domname = format })
        direction "filename"

CREATE(QUEL) – create a new relation
    create relname (domname*1* = format {, domname*2* = format })

DEFINE(QUEL) – define subschema
    define view name (target list) [**where** qual]
    define permit oplist { **on** | **of** | **to** } var [(attlist)] **to** name [**at** term] [**from** time **to** time] [**on** day **to** day] [**where** qual]
    define integrity on var **is** qual

DELETE(QUEL) – delete tuples from a relation
    delete tuple_variable [**where** qual]

DESTROY(QUEL) – destroy existing relation(s)
    destroy relname {, relname}
    destroy [**permit** | **integrity**] relname [integer {, integer}| **all**]

HELP(QUEL) – get information about how to use INGRES or about relations in the database.
    help [relname] ["section"] {, relname}{, "section"}
    help view relname {, relname}
    help permit relname {, relname}
    help integrity relname {, relname}

INDEX(QUEL) – create a secondary index on an existing relation.
    index on relname **is** indexname (domain*1* { ,domain*2*})

INTEGRITY(QUEL) – define integrity constraints
    define integrity on var **is** qual

MACROS(QUEL) – terminal monitor macro facility

MODIFY(QUEL) – convert the storage structure of a relation
    modify relname **to** storage-structure [**on** key1 [: *sortorder*] [{ , key2 [: *sortorder*] }]] [**where** [**fillfactor** = *n*] [**, minpages** = *n*] [**, maxpages** = *n*]]

MONITOR(QUEL) – interactive terminal monitor

PERMIT(QUEL) – add permissions to a relation
    define permit oplist { **on** | **of** | **to** } var [(attlist)]
        to name [**at** term] [**from** time **to** time]
        [**on** day **to** day] [**where** qual]

PRINT(QUEL) – print relation(s)
    print relname {, relname}

QUEL(QUEL) – **QUE**ry **L**anguage for INGRES

RANGE(QUEL) – declare a variable to range over a relation
    range of variable **is** relname

REPLACE(QUEL) – replace values of domains in a relation
    replace tuple_variable (target_list) [**where** qual]

RETRIEVE(QUEL) – retrieve tuples from a relation
    retrieve [[**into**] relname] (target_list) [**where** qual]
    retrieve unique (target_list) [**where** qual]

SAVE(QUEL) – save a relation until a date.
save relname **until** month day year

VIEW(QUEL) – define a virtual relation
define view name (target-list) [**where** qual]

CREATDB(UNIX) – create a data base
creatdb [**–u***name*] [**–e**] [**–m**] [±**c**] [±**q**] dbname

DESTROYDB(UNIX) – destroy an existing database
destroydb [**–s**] [**–m**] dbname

EQUEL(UNIX) – Embedded QUEL interface to C
equel [**–d**] [**–f**] [**–r**] file.q ...

HELPR(UNIX) – get information about a database.
helpr [**–u***name*] [±**w**] database relation ...

INGRES(UNIX) – INGRES relational data base management system
ingres [*flags*] dbname [process_table]

PRINTR(UNIX) – print relations
printr [*flags*] database relation ...

PURGE(UNIX) – destroy all expired and temporary relations
purge [**–f**] [**–p**] [**–a**] [**–s**] [±**w**] [database ...]

RESTORE(UNIX) – recover from an INGRES or UNIX crash.
restore [**–a**] [**–s**] [±**w**] [database ...]

SYSMOD(UNIX) – modify system relations to predetermined storage structures.
sysmod [**–s**] [**–w**] dbname [**relation**] [**attribute**] [**indexes**] [**tree**] [**protect**] [**integrities**]

USERSETUP(UNIX) – setup users file
.../bin/usersetup [pathname]

DAYFILE(FILES) – INGRES login message

DBTMPLT(FILES) – database template

ERROR(FILES) – files with INGRES errors

LIBQ(FILES) – Equel run-time support library

PROCTAB(FILES) – INGRES runtime configuration information

STARTUP(FILES) – INGRES startup file

USERS(FILES) – INGRES user codes and parameters

INTRODUCTION(ERROR) – Error messages introduction

EQUEL(ERROR) – EQUEL error message summary
Error numbers 1000 – 1999.

PARSER(ERROR) – Parser error message summary
Error numbers 2000 – 2999.

QRYMOD(ERROR) – Query Modification error message summary
Error numbers 3000 – 3999.

OVQP(ERROR) – One Variable Query Processor error message summary
Error numbers 4000 – 4499.

DECOMP(ERROR) – Decomposition error message summary
Error numbers 4500 – 4999.

DBU(ERROR) – Data Base Utility error message summary
Error numbers 5000 – 5999

NAME

append – append tuples to a relation

SYNOPSIS

append [**to**] relname (target_list) [**where** qual]

DESCRIPTION

Append adds tuples which satisfy the qualification to *relname*. *Relname* must be the name of an existing relation. The *target_list* specifies the values of the attributes to be appended to *relname*. The domains may be listed in any order. Attributes of the result relation which do not appear in the *target_list* as result_attnames (either explicitly or by default) are assigned default values of 0, for numeric attributes, or blank, for character attributes.

Values or expressions of any numeric type may be used to set the value of a numeric type domain. Conversion to the result domain type takes place. Numeric values cannot be directly assigned to character domains. Conversion from numeric to character can be done using the **ascii** operator (see quel(quel)). Character values cannot be directly assigned to numeric domains. Use the **int1**, **int2**, etc. functions to convert character values to numeric (see quel(quel)).

The keyword **all** can be used when it is desired to append all domains of a relation.

An *append* may only be issued by the owner of the relation or a user with *append* permission on the given relation.

EXAMPLE

```
/* Make new employee Jones work for Smith */
  range of n is newemp
  append to emp(n.name, n.sal, mgr = "Smith", bdate = 1975–n.age)
              where n.name = "Jones"
/* Append the newemp1 relation to newemp */
  range of n1 is newemp1
  append to newemp(n1.all)
```

SEE ALSO

copy(quel), permit(quel), quel(quel), retrieve(quel)

DIAGNOSTICS

Use of a numeric type expression to set a character type domain or vice versa will produce diagnostics.

BUGS

Duplicate tuples appended to a relation stored as a "paged heap" (unkeyed, unstructured) are not removed.

NAME

copy – copy data into/from a relation from/into a UNIX file.

SYNOPSIS

copy relname (domname = format {, domname = format })
direction "filename"

DESCRIPTION

Copy moves data between INGRES relations and standard UNIX files. *Relname* is the name of an existing relation. In general *domname* identifies a domain in *relname*. *Format* indicates the format the UNIX file should have for the corresponding domain. *Direction* is either **into** or **from**. *Filename* is the full UNIX pathname of the file.

On a copy **from** a file to a relation, the relation cannot have a secondary index, it must be owned by you, and it must be updatable (not a secondary index or system relation).

Copy cannot be used on a relation which is a view. For a copy **into** a UNIX file, you must either be the owner of the relation or the relation must have retrieve permission for all users, or all permissions for all users.

The formats allowed by copy are:

i1,i2,i4 – The data is stored as an integer of length 1, 2, or 4 bytes in the UNIX file.

f4,f8 – The data is stored as a floating point number (either single or double precision) in the UNIX file.

c1,c2,...,c255 – The data is stored as a fixed length string of characters.

c0 – Variable length character string.

d0,d1,...,d255 – Dummy domain.

Corresponding domains in the relation and the UNIX file do not have to be the same type or length. *Copy* will convert as necessary. When converting anything except character to character, *copy* checks for overflow. When converting from character to character, *copy* will blank pad or truncate on the right as necessary.

The domains should be ordered according to the way they should appear in the UNIX file. Domains are matched according to name, thus the order of the domains in the relation and in the UNIX file does not have to be the same.

Copy also provides for variable length strings and dummy domains. The action taken depends on whether it is a copy **into** or a copy **from**. Delimitors for variable length strings and for dummy domains can be selected from the list of:

nl – new line character
tab – tab character
sp – space
nul or **null** – null character
comma – comma
colon – colon
dash – dash
lparen – left parenthesis
rparen – right parenthesis
x – any single character 'x'

The special meaning of any delimitor can be turned off by preceeding the delimitor with a '\'. The delimitor can optionally be in quotes ("delim"). This is usefully if you wish to use a single character delimitor which has special meaning to

the QUEL parser.

When the *direction* is from, *copy* appends data into the relation **from** the UNIX file. Domains in the INGRES relation which are not assigned values from the UNIX file are assigned the default value of zero for numeric domains, and blank for character domains. When copying in this direction the following special meanings apply:

c0*delim* – The data in the UNIX file is a variable length character string terminated by the delimitor *delim*. If *delim* is missing then the first comma, tab, or newline encountered will terminate the string. The delimitor is not copied.

For example:

pnum=c0 – string ending in comma, tab, or nl.
pnum=c0nl – string ending in nl.
pnum=c0sp – string ending in space.
pnum=c0"Z" – string ending in the character 'Z'.
pnum=c0"%" – string ending in the character '%'.

A delimitor can be escaped by preceeding it with a '\'. For example, using **name = c0**, the string "Blow\, Joe," will be accepted into the domain as "Blow, Joe".

d0*delim* – The data in the UNIX file is a variable length character string delimited by *delim*. The string is read and discarded. The delimitor rules are identical for **c0** and **d0**. The domain name is ignored.

d1,d2,...,d255 – The data in the UNIX file is a fixed length character string. The string is read and discarded. The domain name is ignored.

When the direction is **into**, *copy* transfers data **into** the UNIX file from the relation. If the file already existed, it is truncated to zero length before copying begins. When copying in this direction, the following special meanings apply:

c0 – The domain value is converted to a fixed length character string and writted into the UNIX file. For character domains, the length will be the same as the domain length. For numeric domains, the standard INGRES conversions will take place as specified by the '**–i**', '**–f**', and '**–c**' flags (see ingres(unix)).

c0*delim* – The domain will be converted according to the rules for **c0** above. The one character delimitor will be inserted immediately after the domain.

d1,d2,...,d255 – The domain name is taken to be the name of the delimitor. It is written into the UNIX file 1 time for **d1**, 2 times for **d2**, etc.

d0 – This format is ignored on a copy **into**.

d0*delim* – The *delim* is written into the file. The domain name is ignored.

If no domains appear in the copy command (i.e. copy relname () into/from "filename") then *copy* automatically does a "bulk" copy of all domains, using the order and format of the domains in the relation. This is provided as a convenient shorthand notation for copying and restoring entire relations.

To *copy* into a relation, you must be the owner or all users must have all permissions set. Correspondingly, to *copy* from a relation you must own the relation or all users must have at least retrieve permission on the relation. Also, you may not *copy* a view.

EXAMPLE

```
/* Copy data into the emp relation */
  copy emp (name=c10,sal=f4,bdate=i2,mgr=c10,xxx=d1)
```

```
          from "/mnt/me/myfile"

/* Copy employee names and their salaries into a file */
  copy emp (name=c0,comma=d1,sal=c0,nl=d1)
          into "/mnt/you/yourfile"

/* Bulk copy employee relation into file */
  copy emp ()
          into "/mnt/ours/ourfile"

/* Bulk copy employee relation from file */
  copy emp ()
          from "/mnt/thy/thyfile"
```

SEE ALSO

append(quel), create(quel), quel(quel), permit(quel), view(quel), ingres(unix)

BUGS

Copy stops operation at the first error.

When specifying *filename*, the entire UNIX directory pathname must be provided, since INGRES operates out of a different directory than the user's working directory at the time INGRES is invoked.

NAME

create — create a new relation

SYNOPSIS

create relname (domname*1* = format {, domname*2* = format })

DESCRIPTION

Create will enter a new relation into the data base. The relation will be "owned" by the user and will be set to expire after seven days. The name of the relation is *relname* and the domains are named *domname1*, *domname2*, etc. The domains are created with the type specified by *format*. Formats are described in the quel(quel) manual section.

The relation is created as a paged heap with no data initially in it.

A relation can have no more than 49 domains. A relation cannot have the same name as a system relation.

EXAMPLE

```
/* Create relation emp with domains name, sal and bdate */
  create emp (name = c10, salary = f4, bdate = i2)
```

SEE ALSO

append(quel), copy(quel), destroy(quel), save(quel)

BUGS

NAME

define — define subschema

SYNOPSIS

define view name (target list) [**where** qual]
define permit oplist { **on** | **of** | **to** } var [(attlist)] **to** name [**at** term] [**from** time **to** time] [**on** day **to** day] [**where** qual]
define integrity on var **is** qual

DESCRIPTION

The *define* statement creates entries for the subschema definitions. See the manual sections listed below for complete descriptions of these commands.

SEE ALSO

integrity(quel), permit(quel), view(quel)

NAME

delete — delete tuples from a relation

SYNOPSIS

delete tuple_variable [**where** qual]

DESCRIPTION

Delete removes tuples which satisfy the qualification *qual* from the relation that they belong to. The *tuple_variable* must have been declared to range over an existing relation in a previous *range* statement. *Delete* does not have a *target_list*. The *delete* command requires a tuple variable from a *range* statement, and not the actual relation name. If the qualification is not given, the effect is to delete all tuples in the relation. The result is a valid, but empty relation.

To *delete* tuples from a relation, you must be the owner of the relation, or have *delete* permission on the relation.

EXAMPLE

```
/* Remove all employees who make over $30,000 */
  range of e is emp
  delete e where e.sal > 30000
```

SEE ALSO

destroy(quel), permit(quel), quel(quel), range(quel)

BUGS

NAME

destroy – destroy existing relation(s)

SYNOPSIS

destroy relname { , relname }
destroy [**permit** | **integrity**] relname [integer { , integer } | **all**]

DESCRIPTION

Destroy removes relations from the data base, and removes constraints or permissions from a relation. Only the relation owner may destroy a relation or its permissions and integrity constraints. A relation may be emptied of tuples, but not destroyed, using the delete statement or the modify statement.

If the relation being destroyed has secondary indices on it, the secondary indices are also destroyed. Destruction of just a secondary index does not affect the primary relation it indexes.

To destroy individual permissions or constraints for a relation, the *integer* arguments should be those printed by a **help permit** (for **destroy permit**) or a **help integrity** (for **destroy integrity**) on the same relation. To destroy all constraints or permissions, the **all** keyword may be used in place of individual integers. To destroy constraints or permissions, either the *integer* arguments or the **all** keyword must be present.

EXAMPLE

```
/* Destroy the emp relation */
   destroy emp
   destroy emp, parts

/* Destroy some permissions on parts, and all integrity
 * constraints on employee -
 */
  destroy permit parts 0, 4, 5
  destroy integrity employee
```

SEE ALSO

create(quel), delete(quel), help(quel), index(quel), modify(quel)

NAME

help – get information about how to use INGRES or about relations in the data-base.

SYNOPSIS

help [relname] ["section"] {, relname}{, "section"}
help view relname {, relname}
help permit relname {, relname}
help integrity relname {, relname}

DESCRIPTION

Help may be used to obtain sections of this manual, information on the content of the current data base, information about specific relations in the data base, view definitions, or protection and integrity constraints on a relation. The legal forms are as follow:

help "*section* " – Produces a copy of the specified section of the INGRES Reference Manual, and prints it on the standard output device.
help – Gives information about all relations that exist in the current database.
help *relname* {, *relname*} – Gives information about the specified relations.
help "" – Gives the table of contents.
help view *relname* {, *relname*} – Prints view definitions of specified views.
help permit *relname* {, *relname*} – Prints permissions on specified relations.
help integrity *relname* {, *relname*} – Prints integrity constraints on specified relations.

The **permit** and **integrity** forms print out unique identifiers for each constraint. These identifiers may be used to remove the constraints with the *destroy* statement.

EXAMPLE

```
help
help help  /* prints this page of the manual */
help quel
help emp
help emp, parts, "help", supply
help view overp_view
help permit parts, employee
help integrity parts, employee
```

SEE ALSO

destroy(quel)

BUGS

Alphabetics appearing within the section name must be in lower-case to be recognized.

NAME

index – create a secondary index on an existing relation.

SYNOPSIS

index on relname **is** indexname (domain*1* { ,domain*2*})

DESCRIPTION

Index is used to create secondary indices on existing relations in order to make retrieval and update with secondary keys more efficient. The secondary key is constructed from relname domains 1, 2,...,6 in the order given. Only the owner of a relation is allowed to create secondary indices on that relation.

In order to maintain the integrity of the index, users will NOT be allowed to directly update secondary indices. However, whenever a primary relation is changed, its secondary indices will be automatically updated by the system. Secondary indices may be modified to further increase the access efficiency of the primary relation. When an index is first created, it is automatically modified to an isam storage structure on all its domains. If this structure is undesirable, the user may override the default isam structure by using the **–n** switch (see ingres(unix)), or by entering a *modify* command directly.

If a *modify* or *destroy* command is used on *relname*, all secondary indices on *relname* are destroyed.

Secondary indices on other indices, or on system relations are forbidden.

EXAMPLE

```
/* Create a secondary index called "x" on relation "emp" */
  index on emp is x(mngr,sal)
```

SEE ALSO

copy(quel), destroy(quel), modify(quel)

BUGS

At most 6 domains may appear in the key.

The *copy* command cannot be used to copy into a relation which has secondary indices.

The default structure isam is a poor choice for an index unless the range of retrieval is small.

NAME

integrity — define integrity constraints

SYNOPSIS

define integrity on var **is** qual

DESCRIPTION

The *integrity* statement adds an integrity constraint for the relation specified by *var*. After the constraint is placed, all updates to the relation must satisfy *qual*. *Qual* must be true when the *integrity* statement is issued or else a diagnostic is issued and the statement is rejected.

In the current implementation, *integrity* constraints are not flagged — bad updates are simply (and silently) not performed.

Qual must be a single variable qualification and may not contain any aggregates.

integrity statements may be issued only by the relation owner.

EXAMPLE

```
/* Ensure all employees have positive salaries */
 range of e is employee
 define integrity on e is e.salary > 0
```

SEE ALSO

destroy(quel)

NAME

macros – terminal monitor macro facility

DESCRIPTION

The terminal monitor macro facility provides the ability to tailor the QUEL language to the user's tastes. The macro facility allows strings of text to be removed from the query stream and replaced with other text. Also, some built in macros change the environment upon execution.

Basic Concepts

All macros are composed of two parts, the *template* part and the *replacement* part. The template part defines when the macro should be invoked. For example, the template "ret" causes the corresponding macro to be invoked upon encountering the word "ret" in the input stream. When a macro is encountered, the template part is removed and replaced with the replacement part. For example, if the replacement part of the "ret" macro was "retrieve", then all instances of the word "ret" in the input text would be replaced with the word "retrieve", as in the statement

    ret (p.all)

Macros may have parameters, indicated by a dollar sign. For example, the template "get $1" causes the macro to be triggered by the word "get" followed by any other word. The word following "get" is remembered for later use. For example, if the replacement part of the "get" macro where

    retrieve (p.all) where p.pnum = $1

then typing "get 35" would retrieve all information about part number 35.

Defining Macros

Macros can be defined using the special macro called "define". The template for the define macro is (roughly)

    {define; $t; $r}

where $t and $r are the template and replacement parts of the macro, respectively.

Let's look at a few examples. To define the "ret" macro discussed above, we would type:

    {define; ret; retrieve}

When this is read, the macro processor removes everything between the curly braces and updates some tables so that "ret" will be recognized and replaced with the word "retrieve". The define macro has the null string as replacement text, so that this macro seems to disappear.

A useful macro is one which shortens range statements. It can be defined with

    {define; rg $v $r; range of $v is $r}

This macro causes the word "rg" followed by the next two words to be removed and replaced by the words "range of", followed by the first word which followed "rg", followed by the word "is", followed by the second word which followed "rg". For example, the input

    rg p parts

becomes the same as

    range of p is parts

Evaluation Times

When you type in a define statement, it is not processed immediately, just as queries are saved rather than executed. No macro processing is done until the query buffer is evaluated. The commands \go, \list, and \eval evaluate the query buffer. \go sends the results to INGRES, \list prints them on your terminal, and \eval puts the result back into the query buffer.

It is important to evaluate any define statements, or it will be exactly like you did not type them in at all. A common way to define macros is to type

```
{define . . . }
\eval
\reset
```

If the \eval was left out, there is no effect at all.

Quoting

Sometimes strings must be passed through the macro processor without being processed. In such cases the grave and acute accent marks (` and ´) can be used to surround the literal text. For example, to pass the word "ret" through without converting it to "retrieve" we could type

```
`ret´
```

Another use for quoting is during parameter collection. If we want to enter more than one word where only one was expected, we can surround the parameter with accents.

The backslash character quotes only the next character (like surrounding the character with accents). In particular, a grave accent can be used literally by preceeding it with a backslash.

Since macros can normally only be on one line, it is frequently useful to use a backslash at the end of the line to hide the newline. For example, to enter the long "get" macro, you might type:

```
{define; get $n; retrieve (e.all) \
where e.name = "$n"}
```

The backslash always quotes the next character even when it is a backslash. So, to get a real backslash, use two backslashes.

More Parameters

Parameters need not be limited to the word following. For example, in the template descriptor for define:

```
{define; $t; $r}
```

the $t parameter ends at the first semicolon and the $r parameters ends at the first right curly brace. The rule is that the character which follows the parameter specifier terminates the parameter; if this character is a space, tab, newline, or the end of the template then one word is collected.

As with all good rules, this one has an exception. Since system macros are always surrounded by curly braces, the macro processor knows that they must be properly nested. Thus, in the statement

```
{define; x; {sysfn}}
```

The first right curly brace will close the "sysfn" rather than the "define". Otherwise this would have to be typed

```
{define; x; `{sysfn}´}
```

Words are defined in the usual way, as strings of letters and digits plus the underscore character.

Other Builtin Macros

There are several other macros built in to the macro processor. In the following description, some of the parameter specifiers are marked with two dollar signs rather than one; this will be discussed in the section on prescanning below.

{define; $$t; $$r} defines a macro as discussed above. Special processing occurs on the template part which will be discussed in a later section.

{rawdefine; $$t; $$r} is another form of define, where the special processing does not take place.

{remove; $$n} removes the macro with name $n. It can remove more than one macro, since it actually removes all macros which might conflict with $n under some circumstance. For example, typing

```
{define; get part $n; . . . }
{define; get emp $x; . . . }
{remove; get}
```

would cause both the get macros to be removed. A call to

```
{remove; get part}
```

would have only removed the first macro.

{type $$s} types $s onto the terminal.

{read $$s} types $s and then reads a line from the terminal. The line which was typed replaces the macro. A macro called "{readcount}" is defined containing the number of characters read. A control-D (end of file) becomes −1, a single newline becomes zero, and so forth.

{readdefine; $$n; $$s} also types $s and reads a line, but puts the line into a macro named $n. The replacement text is the count of the number of characters in the line. {readcount} is still defined.

{ifsame; $$a; $$b; $t; $f} compares the strings $a and $b. If they match exactly then the replacement text becomes $t, otherwise it becomes $f.

{ifeq; $$a; $$b; $t; $f} is similar, but the comparison is numeric.

{ifgt; $$a; $$b; $t; $f} is like ifeq, but the test is for $a strictly greater than $b.

{substr; $$f; $$t; $$s} returns the part of $s between character positions $f and $t, numbered from one. If $f or $t are out of range, they are moved in range as much as possible.

{dump; $$n} returns the value of the macro (or macros) which match $n (using the same algorithm as remove). The output is a rawdefine statement so that it can be read back in. {dump} without arguments dumps all macros.

Metacharacters

Certain characters are used internally. Normally you will not even see them, but they can appear in the output of a dump command, and can sometimes be used to create very fancy macros.

\| matches any number of spaces, tabs, or newlines. It will even match zero, but only between words, as can occur with punctuation. For example, \| will match the spot between the last character of a word and a comma following it.

\^ matches exactly one space, tab, or newline.

\& matches exactly zero spaces, tabs, or newlines, but only between words.

The Define Process

When you define a macro using define, a lot of special processing happens. This processing is such that define is not functionally complete, but still adequate for most requirements. If more power is needed, rawdefine can be used; however, rawdefine is particularly difficult to use correctly, and should only be used by gurus.

In define, all sequences of spaces, tabs, and newlines in the template, as well as all "non-spaces" between words, are turned into a single \| character. If the template ends with a parameter, the \& character is added at the end.

If you want to match a real tab or newline, you can use \t or \n respectively. For example, a macro which reads an entire line and uses it as the name of an employee would be defined with

```
{define; get $n\n; \
   ret (e.all) where e.name = "$n"}
```

This macro might be used by typing

```
get *Stan*
```

to get all information about everyone with a name which included "Stan". By the way, notice that it is ok to nest the "ret" macro inside the "get" macro.

Parameter Prescan

Sometimes it is useful to macro process a parameter before using it in the replacement part. This is particularly important when using certain builtin macros.

For prescan to occur, two things must be true: first, the parameter must be specified in the template with two dollar signs instead of one, and second, the actual parameter must begin with an "at" sign ("@") (which is stripped off).

For an example of the use of prescan, see "Special Macros" below.

Special Macros

Some special macros are used by the terminal monitor to control the environment and return results to the user.

{begintrap} is executed at the beginning of a query.

{endtrap} is executed after the body of a query is passed to INGRES.

{continuetrap} is executed after the query completes. The difference between this and endtrap is that endtrap occurs after the query is submitted, but before the query executes, whereas continuetrap is executed after the query executes.

{editor} can be defined to be the pathname of an editor to use in the \edit command.

{shell} can be defined to be the pathname of a shell to use in the \shell command.

{tuplecount} is set after every query (but before continuetrap is sprung) to be the count of the number of tuples which satisfied the qualification of the query in a retrieve, or the number of tuples changed in an update. It is not set for DBU functions. If multiple queries are run at once, it is set to the number of tuples which satisfied the last query run.

For example, to print out the number of tuples touched automatically after each query, you could enter:

```
{define; {begintrap}; {remove; {tuplecount}}}
```

```
{define; {continuetrap}; \
  {ifsame; @{tuplecount}; {tuplecount};; \
   {type @{tuplecount} tuples touched}}}
```

SEE ALSO

monitor(quel)

NAME

modify – convert the storage structure of a relation

SYNOPSIS

modify relname **to** storage-structure [**on** key1 [: *sortorder*] [{ , key2 [: *sortorder*] }]] [**where** [**fillfactor** = *n*] [, **minpages** = *n*] [, **maxpages** = *n*]]

DESCRIPTION

Relname is modified to the specified storage structure. Only the owner of a relation can modify that relation. This command is used to increase performance when using large or frequently referenced relations. The storage structures are specified as follows:

- **isam** – indexed sequential storage structure
- **cisam** – compressed isam
- **hash** – random hah storage structure
- **chash** – compressed hash
- **heap** – unkeyed and unstructured
- **cheap** – compressed heap
- **heapsort** – heap with tuples sorted and duplicates removed
- **cheapsort** – compressed heapsort
- **truncated** – heap with all tuples deleted

The paper "Creating and Maintaining a Database in INGRES" (ERL Memo M77–71) discusses how to select storage structures based on how the relation is used.

The current compression algorithm only suppresses trailing blanks in character fields. A more effective compression scheme may be possible, but tradeoffs between that and a larger and slower compression algorithm are not clear.

If the *on* phrase is omitted when modifying to isam, cisam, hash or chash, the relation will automatically be keyed on the first domain. When modifying to heap or cheap the *on* phrase must be omitted. When modifying to heapsort or cheapsort the *on* phrase is optional.

When a relation is being sorted (isam, cisam, heapsort and cheapsort), the primary sort keys will be those specified in the *on* phrase (if any). The first key after the *on* phrase will be the most significant sort key and each successive key specified will be the next most significant sort key. Any domains not specified in the *on* phrase will be used as least significant sort keys in domain number sequence.

When a relation is modified to heapsort or cheapsort, the *sortorder* can be specified to be **ascending** or **descending**. The default is always **ascending**. Each key given in the *on* phrase can be optionally modified to be:

key:descending

which will cause that key to be sorted in descending order. For completeness, **ascending** can be specified after the colon (':'), although this is unnecessary since it is the default. **Descending** can be abbreviated by a single '**d**' and, correspondingly, **ascending** can be abreviated by a single '**a**'.

Fillfactor specifies the percentage (from 1 to 100) of each primary data page that should be filled with tuples, under ideal conditions. *Fillfactor* may be used with isam, cisam, hash and chash. Care should be taken when using large fillfactors since a non-uniform distribution of key values could cause overflow pages to be created, and thus degrade access performance for the relation.

Minpages specifies the minimum number of primary pages a hash or chash relation must have. *Maxpages* specifies the maximum number of primary pages a hash or chash relation may have. *Minpages* and *maxpages* must be at least one.

If both **minpages** and **maxpages** are specified in a modify, **minpages** cannot exceed **maxpages**.

Default values for **fillfactor**, **minpages**, and **maxpages** are as follows:

	FILLFACTOR	*MINPAGES*	*MAXPAGES*
hash	50	10	no limit
chash	75	1	no limit
isam	80	NA	NA
cisam	100	NA	NA

EXAMPLES

```
/* modify the emp relation to an indexed
     sequential storage structure with
     "name" as the keyed domain */

modify emp to isam on name

/* if "name" is the first domain of the emp relation,
     the same result can be achieved by */

modify emp to isam

/* do the same modify but request a 60% occupancy
     on all primary pages */

modify emp to isam on name where fillfactor = 60

/* modify the supply relation to compressed hash
     storage structure with "num" and "quan"
     as keyed domains */

modify supply to chash on num, quan

/* now the same modify but also request 75% occupancy
     on all primary, a minimum of 7 primary pages
     pages and a maximum of 43 primary pages */

modify supply to chash on num, quan
     where fillfactor = 75, minpages = 7,
     maxpages = 43

/* again the same modify but only request a minimum
     of 16 primary pages */

modify supply to chash on num, quan
     where minpages = 16

/* modify parts to a heap storage structure */

modify parts to heap

/* modify parts to a heap again, but have tuples
     sorted on "pnum" domain and have any duplicate
     tuples removed */

modify parts to heapsort on pnum

/* modify employee in ascending order by manager,
     descending order by salary and have any
     duplicate tuples removed */
```

```
modify employee to heapsort on manager, salary:descending
```

SEE ALSO

sysmod(unix)

NAME

monitor – interactive terminal monitor

DESCRIPTION

The interactive terminal monitor is the primary front end to INGRES. It provides the ability to formulate a query and review it before issuing it to INGRES. If changes must be made, one of the UNIX text editors may be called to edit the *query buffer.*

Messages and Prompts.

The terminal monitor gives a variety of messages to keep the user informed of the status of the monitor and the query buffer.

As the user logs in, a login message is printed. This typically tells the version number and the login time. It is followed by the dayfile, which gives information pertinant to users.

When INGRES is ready to accept input, the message "go" is printed. This means that the query buffer is empty. The message "continue" means that there is information in the query buffer. After a \go command the query buffer is automatically cleared if another query is typed in, unless a command which affects the query buffer is typed first. These commands are \append, \edit, \print, \list, \eval, and \go. For example, typing

```
help parts
\go
print parts
```

results in the query buffer containing
print parts
whereas

```
help parts
\go
\print
print parts
```

results in the query buffer containing

```
help parts
print parts
```

An asterisk is printed at the beginning of each line when the monitor is waiting for the user to type input.

Commands

There are a number of commands which may be entered by the user to affect the query buffer or the user's environment. They are all preceeded by a backslash ('\'), and all are executed immediately (rather than at execution time like queries).

Some commands may take a filename, which is defined as the first significant character after the end of the command until the end of the line. These commands may have no other commands on the line with them. Commands which do not take a filename may be stacked on the line; for example

```
\date\go\date
```

will give the time before and after execution of the current query buffer.

\r
\reset Erase the entire query (reset the query buffer). The former contents of the buffer are irretrieveably lost.

\p
\print Print the current query. The contents of the buffer are printed on the user's terminal.

\l
\list Print the current query as it will appear after macro processing. Any side effects of macro processing, such as macro definition, will occur.

\eval Macro process the query buffer and replace the query buffer with the result. This is just like \list except that the output is put into the query buffer instead of to the terminal.

\e
\ed
\edit
\editor Enter the UNIX text editor (see ED in the UNIX Programmer's Manual); use the ED command 'w' followed by 'q' to return to the INGRES monitor. If a filename is given, the editor is called with that file instead of the query buffer. If the macro "{editor}" is defined, that macro is used as the pathname of an editor, otherwise "/bin/ed" is used. It is important that you do not use the "e" command inside the editor; if you do the (obscure) name of the query buffer will be forgotten.

\g
\go Process the current query. The contents of the buffer are macro processed, transmitted to INGRES, and run.

\a
\append Append to the query buffer. Typing \a after completion of a query will override the auto-clear feature and guarantees that the query buffer will not be reset.

\time
\date Print out the current time of day.

\s
\sh
\shell Escape to the UNIX shell. Typing a control-d will cause you to exit the shell and return to the INGRES monitor. If there is a filename specified, that filename is taken as a shell file which is run with the query buffer as the parameter "$1". If no filename is given, an interactive shell is forked. If the macro "{shell}" is defined, it is used as the pathname of a shell; otherwise, "/bin/sh" is used.

\q
\quit Exit from INGRES.

\cd
\chdir Change the working directory of the monitor to the named directory.

\i
\include
\read Switch input to the named file. Backslash characters in the file will be processed as read.

\w
\write Write the contents of the query buffer to the named file.

\branch Transfer control within a \include file. See the section on branching below.

\mark Set a label for \branch.

\<any other character>
Ignore any possible special meaning of character following '\'. This allows the '\' to be input as a literal character. (See also quel(quel) - strings). It is important to note that backslash escapes are sometimes eaten up by the macro processor also; in general, send two backslashes

if you want a backslash sent (even this is too simplistic [sigh] – try to avoid using backslashes at all).

Macros

For simplicity, the macros are described in the section macros(quel).

Branching

The \branch and \mark commands permit arbitrary branching within a \include file (similar to the "goto" and ":" commands in the shell). \mark should be followed with a label. \branch should be followed with either a label, indicating unconditional branch, or an expression preceeded by a question mark, followed by a label, indicating a conditional branch. The branch is taken if the expression is greater than zero. For example,

    \branch ?{tuplecount}<=0 notups

branches to label "notups" if the "{tuplecount}" macro is less than or equal to zero.

The expressions usable in \branch statements are somewhat restricted. The operators +, –, *, /, <=, >=, <, >, =, and != are all defined in the expected way. The left unary operator "!" can be used as to indicate logical negation. There may be no spaces in the expression, since a space terminates the expression.

Initialization

At initialization (login) time a number of initializations take place. First, a macro called "{pathname}" is defined which expands to the pathname of the INGRES subtree (normally "/mnt/ingres"); it is used by system routines such as demodb. Second, the initialization file .../files/startup is read. This file is intended to define system-dependent parameters, such as the default editor and shell. Third, a user dependent initialization file, specified by a field in the users file, is read and executed. This is normally set to the file ".ingres" in the user's home directory. The startup file might be used to define certain macros, execute common range statements, and soforth. Finally, control is turned over to the user's terminal.

An interrupt while executing either of the initialization files restarts execution of that step.

Flags

Certain flags may be included on the command line to INGRES which affect the operation of the terminal monitor. The –a flag disables the autoclear function. This means that the query buffer will never be automatically cleared; equivalently, it is as though a \append command were inserted after every \go. Note that this means that the user must explicitly clear the query buffer using \reset after every query. The –d flag turns off the printing of the dayfile. The –s flag turns off printing of all messages (except errors) from the monitor, including the login and logout messages, the dayfile, and prompts. It is used for executing "canned queries", that is, queries redirected from files.

SEE ALSO

ingres(unix), quel(quel), macros(quel)

DIAGNOSTICS

go	You may begin a fresh query.
continue	The previous query is finished and you are back in the monitor.
Executing . . .	The query is being processed by INGRES.

>>ed	You have entered the UNIX text editor.
>>sh	You have escaped to the UNIX shell.

Funny character nnn converted to blank

INGRES maps non-printing ASCII characters into blanks; this message indicates that one such conversion has just been made.

INCOMPATIBILITIES

Note that the construct

\rprint parts

(intended to reset the query buffer and then enter "print parts") no longer works, since "rprint" appears to be one word.

BUGS

NAME

permit – add permissions to a relation

SYNOPSIS

define permit oplist { **on** | **of** | **to** } var [(attlist)]
to name [**at** term] [**from** time **to** time]
[**on** day **to** day] [**where** qual]

DESCRIPTION

The *permit* statement extends the current permissions on the relation specified by *var*. *Oplist* is a comma separated list of possible operations, which can be **retrieve**, **replace**, **delete**, **append**, or **all**; *all* is a special case meaning all permissions. *Name* is the login name of a user or the word **all**. *Term* is a terminal name of the form 'tty*x*' or the keyword **all**; omitting this phrase is equivalent to specifying *all*. *Times* are of the form 'hh:mm' on a twenty-four hour clock which limit the times of the day during which this permission applies. *Days* are three-character abbreviations for days of the week. The *qual* is appended to the qualification of the query when it is run.

Separate parts of a single *permit* statement are conjoined (ANDed). Different *permit* statements are disjoined (ORed). For example, if you include

. . . to eric at tty4 . . .

the *permit* applies only to eric when logged in on tty4, but if you include two *permit* statements

. . . to eric at all . . .
. . . to all at tty4 . . .

then when eric logs in on tty4 he will get the union of the permissions specified by the two statements. If eric logs in on ttyd he will get only the permissions specified in the first *permit* statement, and if bob logs in on tty4 he will get only the permissions specified in the second *permit* statment.

The *permit* statement may only be issued by the owner of the relation. Although a user other than the DBA may issue a *permit* statement, it is useless because noone else can access her relations anyway.

Permit statements do not apply to the owner of a relation or to views.

The statements

```
define permit all on x to all
define permit retrieve of x to all
```

with no further qualification are handled as special cases and are thus particularly efficient.

EXAMPLES

```
range of e is employee
define permit retrieve of e (name, sal) to marc
        at ttyd from 8:00 to 17:00
        on Mon to Fri
        where e.mgr = "marc"

range of p is parts
define permit retrieve of e to all
```

SEE ALSO

destroy(quel)

NAME

print – print relation(s)

SYNOPSIS

print relname {, relname}

DESCRIPTION

Print displays the contents of each relation specified on the terminal (standard output). The formats for various types of domains can be defined by the use of switches when *ingres* is invoked. Domain names are truncated to fit into the specified width.

To print a relation one must either be the owner of the relation, or the relation must have "retrieve to all" or "all to all" permissions.

See ingres(quel) for details.

EXAMPLE

```
/* Print the emp relation */
  print emp
  print emp, parts
```

SEE ALSO

permit(quel), retrieve(quel), ingres(unix), printr(unix) handle long lines of output correctly – no wrap around.

Print should have more formating features to make printouts more readable.

Print should have an option to print on the line printer.

NAME

quel – QUEry Language for INGRES

DESCRIPTION

The following is a description of the general syntax of QUEL. Individual QUEL statements and commands are treated separately in the document; this section describes the syntactic classes from which the constituent parts of QUEL statements are drawn.

1. Comments

A comment is an arbitrary sequence of characters bounded on the left by "/*" and on the right by "*/":

/* This is a comment */

2. Names

Names in QUEL are sequences of no more than 12 alphanumeric characters, starting with an alphabetic. Underscore (_) is considered an alphabetic. All upper-case alphabetics appearing anywhere except in strings are automatically and silently mapped into their lower-case counterparts.

3. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

abs	**all**	**and**
any	**append**	**ascii**
at	**atan**	**avg**
avgu	**by**	**concat**
copy	**cos**	**count**
countu	**create**	**define**
delete	**destroy**	**exp**
float4	**float8**	**from**
gamma	**help**	**in**
index	**int1**	**int2**
int4	**integrity**	**into**
is	**log**	**max**
min	**mod**	**modify**
not	**of**	**on**
onto	**or**	**permit**
print	**range**	**replace**
retrieve	**save**	**sin**
sqrt	**sum**	**sumu**
to	**unique**	**until**
view	**where**	

4. Constants

There are three types of constants, corresponding to the three data types available in QUEL for data storage.

4.1. String constants

Strings in QUEL are sequences of no more than 255 arbitrary ASCII characters bounded by double quotes (" "). Upper case alphabetics within strings are accepted literally. Also, in order to imbed quotes within strings, it is necessary to prefix them with '\' . The same convention applies to '\' itself.

Only printing characters are allowed within strings. Non-printing characters (i.e. control characters) are converted to blanks.

4.2. Integer constants

Integer constants in QUEL range from −2,147,483,647 to +2,147,483,647. Integer constants beyond that range will be converted to floating point. If the integer is greater than 32,767 or less than −32,767 then it will be left as a two byte integer. Otherwise it is converted to a four byte integer.

4.3. Floating point constants

Floating constants consist of an integer part, a decimal point, and a fraction part or scientific notation of the following format:

{<dig>} [.<dig>] [e|E [+|−] {<dig>}]

Where <dig> is a digit, [] represents zero or one, {} represents zero or more, and | represents alternation. An exponent with a missing mantissa has a mantissa of 1 inserted. There may be no extra characters embedded in the string. Floating constants are taken to be double-precision quantities with a range of approximately -10^{38} to 10^{38} and a precision of 17 decimal digits.

5. Attributes

An attribute is a construction of the form:

variable.domain

Variable identifies a particular relation and can be thought of as standing for the rows or tuples of that relation. A variable is associated with a relation by means of a *range* statement. *Domain* is the name of one of the columns of the relation over which the variable ranges. Together they make up an attribute, which represents values of the named domain.

6. Arithmetic operators

Arithmetic operators take numeric type expressions as operands. Unary operators group right to left; binary operators group left to right. The operators (in order of descending precedence) are:

+,− (unary) plus, minus
** exponentiation
*,/ multiplication, division
+,− (binary) addition, subtraction

Parentheses may be used for arbitrary grouping. Arithmetic overflow and divide by zero are not checked on integer operations. Floating point operations are checked for overflow, underflow, and divide by zero only if the appropriate machine hardware exists and has been enabled.

7. Expressions (a_expr)

An expression is one of the following:

constant
attribute
functional expression
aggregate or aggregate function
a combination of numeric expressions and arithmetic operators

For the purposes of this document, an arbitrary expression will be refered to by the name *a_expr*.

8. Formats

Every *a_expr* has a format denoted by a letter (**c**, **i**, or **f**, for character, integer,

or floating data types respectively) and a number indicating the number of bytes of storage occupied. Formats currently supported are listed below. The ranges of numeric types are indicated in parentheses.

c1 – c255	character data of length 1–255 characters
i1	1-byte integer (–128 to +127)
i2	2-byte integer (–32768 to +32767)
i4	4-byte integer (–2,147,483,648 to +2,147,483,647)
f4	4-byte floating (-10^{38} to $+10^{38}$, 7 decimal digit precision)
f4	8-byte floating (-10^{38} to $+10^{38}$, 17 decimal digit precision)

One numeric format can be converted to or substituted for any other numeric format.

9. Type Conversion.

When operating on two numeric domains of different types, INGRES converts as necessary to make the types identical.

When operating on an integer and a floating point number, the integer is converted to a floating point number before the operation. When operating on two integers of different sizes, the smaller is converted to the size of the larger. When operating on two floating point number of different size, the larger is converted to the smaller.

The following table summarizes the possible combinations:

	i1	i2	i4	f4	f8
i1 –	i1	i2	i4	f4	f8
i2 –	i2	i2	i4	f4	f8
i4 –	i4	i4	i4	f4	f8
f4 –	f4	f4	f4	f4	f4
f8 –	f8	f8	f8	f4	f8

INGRES provides five type conversion operators specifically for overriding the default actions. The operators are:

int1(a_expr)	result type i1
int2(a_expr)	result type i2
int4(a_expr)	result type i4
float4(a_expr)	result type f4
float8(a_expr)	result type f8

The type conversion operators convert their argument a_expr to the requested type. *A_expr* can be anything including character. If a character value cannot be converted, an error occures and processing is halted. This can happen only if the syntax of the character value is incorrect.

Overflow is not checked on conversion.

10. Target_list

A target list is a parenthesized, comma separated list of one or more elements , each of which must be of one of the following forms:

a) *result_attname* **is** *a_expr*

Result_attname is the name of the attribute to be created (or an already existing attribute name in the case of update statements.) The equal sign ("=") may be used interchangeably with **is**. In the case where *a_expr* is anything other than a single attribute, this form must be used to assign a result name to the expression.

b) *attribute*

In the case of a *retrieve*, the resultant domain will acquire the same name as that of the attribute being retrieved. In the case of update statements (*append*, *replace*), the relation being updated must have a domain with exactly that name.

Inside the target list the keyword **all** can be used to represent all domains. For example:

```
range of e is employee
retrieve (e.all) where e.salary > 10000
```

will retrieve all domains of employee for those tuples which satisfy the qualification. **All** can be used in the target list of a *retrieve* or an **append**. The domains will be inserted in their "create" order, that is, the same order they were listed in the *create* statement.

11. Comparison operators

Comparison operators take arbitrary expressions as operands.

<	(less than)
<=	(less than or equal)
>	(greater than)
>=	(greater than or equal)
=	(equal to)
!=	(not equal to)

They are all of equal precedence. When comparisons are made on character attributes, all blanks are ignored.

12. Logical operators

Logical operators take clauses as operands and group left-to-right:

not	(logical not; negation)
and	(logical and; conjunction)
or	(logical or; disjunction)

Not has the highest precedence of the three. **And** and **or** have equal precedence. Parentheses may be used for arbitrary grouping.

13. Qualification (qual)

A *qualification* consists of any number of clauses connected by logical operators. A clause is a pair of expressions connected by a comparison operator:

a_expr comparison_operator a_expr

Parentheses may be used for arbitrary grouping. A qualification may thus be:

clause
not *qual*
qual **or** *qual*
qual **and** *qual*
(*qual*)

14. Functional expressions

A *functional expression* consists of a function name followed by a parenthesized (list of) operand(s). Functional expressions can be nested to any level. In the following list of functions supported (*n*) represents an arbitrary numeric type expression. The format of the result is indicated on the right.

abs(*n*) –	same as *n* (absolute value)
ascii(*n*) –	character string (converts numeric to character)
atan(*n*) –	f8 (arctangent)
concat(*a*, *b*) –	character (character concatenation. See 16.2)

cos(n) — f8 (cosine)
exp(n) — f8 (exponential of n)
gamma(n) — f8 (log gamma)
log(n) — f8 (natural logarithm)
mod(n,b) — same as b (n modulo b. n and b must be i1, i2, or i4)
sin(n) — f8 (sine)
sqrt(n) — f8 (square root)

15. Aggregate expressions

Aggregate expressions provide a way to aggregate a computed expression over a set of tuples.

15.1. Aggregation operators

The definitions of the aggregates are listed below.

count — (i4) count of occurrences
countu — (i4) count of unique occurrences
sum — summation
sumu — summation of unique values
avg — (f8) average (sum/count)
avgu — (f8) unique average (sumu/countu)
max — maximum
min — minimum
any — (i2) value is 1 if any tuples satisfy the qualification, else it is 0

15.2. Simple aggregate

aggregation_operator (*a_expr* [**where** *qual*])

A simple aggregate evaluates to a single scalar value. *A_expr* is aggregated over the set of tuples satisfying the qualification (or all tuples in the range of the expression if no qualification is present). Operators *sum* and *avg* require numeric type *a_expr; count, any, max* and *min* permit a character type attribute as well as numeric type *a_expr.*

Simple aggregates are completely local. That is, they are logically removed from the query, processed separately, and replaced by their scalar value.

15.3. "**any**" aggregate

It is sometimes useful to know if any tuples satisfy a particular qualification. One way of doing this is by using the aggregate *count* and checking whether the return is zero or non-zero. Using *any* instead of *count* is more efficient since processing is stopped, if possible, the first time a tuple satisfies a qualification.

Any returns 1 if the qualification is true and 0 otherwise.

15.4. Aggregate functions

aggregation_operator (*a_expr* by *by_domain*
{, *by_domain*} [**where** *qual*])

Aggregate functions are extensions of simple aggregates. The *by* operator groups (i.e. partitions) the set of qualifying tuples by *by_domain* values. For more than one *by_domain*, the values which are grouped by are the concatenation of individual *by_domain* values. *A_expr* is as in simple aggregates. The aggregate function evaluates to a set of aggregate results, one for each partition into which the set of qualifying tuples has been grouped. The aggregate value used during evaluation of the query is the value associated with the partition into which the tuple currently being processed would fall.

Unlike simple aggregates, aggregate functions are not completely local. The *by_list*, which differentiates aggregate functions from simple aggregates, is global to the query. Domains in the *by_list* are automatically linked to the other domains in the query which are in the same relation.

Example:

```
/* retrieve the average salary for the employees
working for each manager */
range of e is employee
retrieve (e.manager, avesal=avg(e.salary by e.manager))
```

15.5 Aggregates on Unique Values.

It is occasionally necessary to aggregate on unique values of an expression. The *avgu*, *sumu*, and *countu* aggregates all remove duplicate values before performing the aggregation. For example:

count(e.manager)

would tell you how many occurrences of *e.manager* exist. But

countu(e.manager)

would tell you how many unique values of *e.manager* exist.

16. Special character operators

There are three special features which are particular to character domains.

16.1 Pattern matching characters

There are four characters which take on special meaning when used in character constants (strings):

- * matches any string of zero or more characters.
- ? matches any single character.
- [..] matches any of characters in the brackets.

These characters can be used in any combination to form a variety of tests. For example:

where e.name = "*" – matches any name.
where e.name = "E*" – matches any name starting with "E".
where e.name = "*ein" – matches all names ending with "ein"
where e.name = "*[aeiou]*" – matches any name with at least one vowel.
where e.name = "Allman?" – matches any seven character name starting with "Allman".
where e.name = "[A–J]*" – matches any name starting with A,B,..,J.

The special meaning of the pattern matching characters can be disabled by preceding them with a '\'. Thus "*" refers to the character "*". When the special characters appear in the target list they must be escaped. For example:

title = "*** ingres ***"

is the correct way to assign the string "*** ingres ***" to the domain "title".

16.2 Concatenation

There is a concatenation operator which can form one character string from two. Its syntax is "concat(field1. field2)". The size of the new character string is the sum of the sizes of the original two. Trailing blanks are trimmed from the first field, the second field is concatenated and the remainder is blank padded. The result is never trimmed to 0 length, however. Concat can be arbitrarily nested inside other concats. For example:

```
name = concat(concat(x.lastname, ","), x.firstname)
```

will concatenate x.lastname with a comma and then concatenate x.firstname to that.

16.3 Ascii (numeric to character translation)

The *ascii* function can be used to convert a numeric field to its character representation. This can be useful when it is desired to compare a numeric value with a character value. For example:

```
retrieve ( ... )
        where x.chardomain = ascii(x.numdomain)
```

Ascii can be applied to a character value. The result is simply the character value unchanged. The numeric conversion formats are determined by the printing formats (see ingres(unix)).

SEE ALSO

append(quel), delete(quel), range(quel), replace(quel), retrieve(quel), ingres(unix)

BUGS

The maximum number of variables which can appear in one query is 10.

Numeric overflow, underflow, and divide by zero are not detected.

When converting between numeric types, overflow is not checked.

NAME

range – declare a variable to range over a relation

SYNOPSIS

range of variable **is** relname

DESCRIPTION

Range is used to declare variables which will be used in subsequent QUEL statements. The *variable* is associated with the relation specified by *relname*. When the *variable* is used in subsequent statements it will refer to a tuple in the named relation. A range declaration remains in effect for an entire INGRES session (until exit from INGRES), until the variable is redeclared by a subsequent range statement, or until the relation is removed with the destroy command.

EXAMPLE

```
/* Declare tuple variable e to range over relation emp */
  range of e is emp
```

SEE ALSO

quel(quel), destroy(quel)

BUGS

Only 10 variable declarations may be in effect at any time. After the 10th range statement, the least recently referenced variable is re-used for the next range statement.

NAME

replace – replace values of domains in a relation

SYNOPSIS

replace tuple_variable (target_list) [**where** qual]

DESCRIPTION

Replace changes the values of the domains specified in the *target_list* for all tuples which satisfy the qualification *qual*. The *tuple_variable* must have been declared to range over the relation which is to be modified. Note that a tuple variable is required and not the relation name. Only domains which are to be modified need appear in the *target_list*. These domains must be specified as result_attnames in the *target_list* either explicitly or by default (see quel(quel)).

Numeric domains may be replaced by values of any numeric type (with the exception noted below). Replacement values will be converted to the type of the result domain.

Only the owner of a relation, or a user with replace pemission on the relation can do *replace*.

If the tuple update would violate an integrity constraint (see integrity(quel)), it is not done.

EXAMPLE

```
/* Give all employees who work for Smith a 10% raise */
  range of e is emp
  replace e(sal = 1.1 * e.sal) where e.mgr = "Smith"
```

SEE ALSO

integrity(quel), permit(quel), quel(quel), range(quel)

DIAGNOSTICS

Use of a numeric type expression to replace a character type domain or vice versa will produce diagnostics.

BUGS

NAME

retrieve – retrieve tuples from a relation

SYNOPSIS

retrieve [[**into**] relname] (target_list) [**where** qual]
retrieve unique (target_list) [**where** qual]

DESCRIPTION

Retrieve will get all tuples which satisfy the qualification and either display them on the terminal (standard output) or store them in a new relation.

If a relname is specified, the result of the query will be stored in a new relation with the indicated name. A relation with this name owned by the user must not already exist. The current user will be the owner of the new relation. The relation will have domain names as specified in the *target_list* result_attnames. The new relation will be saved on the system for seven days unless explicitly saved by the user until a later date.

If the keyword **unique** is present, tuples will be sorted on the first domain, and duplicates will be removed, before being displayed.

The keyword **all** can be used when it is desired to retrieve all domains.

If no result *relname* is specified then the result of the query will be displayed on the terminal and will not be saved. Duplicate tuples are not removed when the result is displayed on the terminal.

The format in which domains are printed can be defined at the time ingres is invoked (see ingres(unix)).

If a result relation is specified then the default procedure is to modify the result relation to an cheapsort storage structure removing duplicate tuples in the process.

If the default cheapsort structure is not desired, the user can override this at the time INGRES is invoked by using the **–r** switch (see ingres(unix)).

Only the relation's owner and users with retrieve permission may *retrieve* from it.

EXAMPLE

```
/* Find all employees who make more than their manager */
  range of e is emp
  range of m is emp
  retrieve (e.name) where e.mgr = m.name
         and e.sal > m.sal
/* Retrieve all domains for those who make more
       than the average salary */
  retrieve into temp (e.all) where e.sal > avg(e.sal)
/* retrieve employees's names sorted */
  retrieve unique (e.name)
```

SEE ALSO

modify(quel), permit(quel), quel(quel), range(quel), save(quel), ingres(unix)

DIAGNOSTICS

BUGS

NAME

save — save a relation until a date.

SYNOPSIS

save relname **until** month day year

DESCRIPTION

Save is used to keep relations beyond the default 7 day life span.

Month can be an integer from 1 through 12, or the name of the month, either abbreviated or spelled out.

Only the owner of a relation can *save* that relation. There is an INGRES process which typically removes a relation immediately after its expiration date has passed.

The actual program which destroys relations is called purge. It is not automatically run. It is a local decision when expired relations are removed.

System relations have no expiration date.

EXAMPLE

```
/* Save the emp relation until the end of February 1987 */
  save emp until feb 28 1987
```

SEE ALSO

create(quel), retrieve(quel), purge(unix)

NAME

view – define a virtual relation

SYNOPSIS

define view name (target-list) [**where** qual]

DESCRIPTION

The syntax of the *view* statement is almost identical to the *retrieve into* statment; however, the data is not retrieved. Instead, the definition is stored. When the relation *name* is later used, the query is converted to operate on the relations specified in the *target-list*.

All forms of retrieval on the view are fully supported, but only a limited set of updates are supported because of anomolies which can appear. Almost no updates are supported on views which span more than one relation. No updates are supported that affect a domain in the qualification of the view or that affect a domain which does not translate into a simple attribute.

In general, updates are supported if and only if it can be guaranteed (without looking at the actual data) that the result of updating the view is identical to that of updating the corresponding real relation.

The person who defines a view must own all relations upon which the view is based.

EXAMPLE

```
range of e is employee
range of d is dept
define view empdpt (ename = e.name, e.sal, dname = d.name)
      where e.mgr = d.mgr
```

SEE ALSO

retrieve(quel), destroy(quel)

NAME

creatdb – create a data base

SYNOPSIS

creatdb [**–u***name*] [**–e**] [**–m**] [±**c**] [±**q**] dbname

DESCRIPTION

Creatdb creates a new INGRES database, or modifies the status of an existing database. The person who executes this command becomes the Database Administrator (DBA) for the database. The DBA has special powers not granted to ordinary users.

Dbname is the name of the database to be created. The name must be unique among all INGRES users.

The flags ±**c** and ±**q** specify options on the database. The form +*x* turns an option on, while –*x* turns an option off. The **–c** flag turns off the concurrency control scheme (default on). The **+q** flag turns on query modification (default on).

Concurrency control should not be turned off except on databases which are never accessed by more than one user. This applies even if users do not share data relations, since system relations are still shared. If the concurrency control scheme is not installed in UNIX, or if the special file /dev/lock does not exist or is not accessible for read-write by INGRES, concurrency control acts as though it is off (although it will suddenly come on when the lock driver is installed in UNIX).

Query modification must be turned on for the protection, integrity, and view subsystems to work, however, the system will run slightly slower in some cases if it is turned on. It is possible to turn query modification on if it is already off in an existing database, but it is not possible to turn it off if it is already on.

Databases with query modification turned off create new relations with all access permitted for all users, instead of no access except to the owner, the default for databases with query modification enabled.

Database options for an existing database may be modified by stating the **–e** flag. The database is adjusted to conform to the option flags. For example:

creatdb –e +q mydb

turns query modification on for database "mydb" (but leaves concurrency control alone). Only the database administrator (DBA) may use the **–e** flag.

When query modification is turned on, new relations will be created with no access, but previously created relations will still have all access to everyone. The *destroy* command may be used to remove this global permission, after which more selective permissions may be specified with the *permit* command.

The INGRES user may use the **–u** flag to specify a different DBA: the flag should be immediately followed by the login name of the user who should be the DBA.

The **–m** flag specifies that the UNIX directory in which the database is to reside already exists. This should only be needed if the directory if a mounted file system, as might occur for a very large database. The directory must exist (as **.../data/base/***dbname*), must be mode 777, and must be empty of all files.

The user who executes this command must have the U_CREATDB bit set in the status field of her entry in the users file.

The INGRES superuser can create a file in **.../data/base** containing a single line which is the full pathname of the location of the database. The file must be owned by INGRES and be mode 600. When the database is created, it will be created in the file named, rather than in the directory **.../data/base**. For example, if

the file **.../data/base/ericdb** contained the line

/mnt/eric/database

then the database called "ericdb" would be physically stored in the directory /mnt/eric/database rather than in the directory .../data/base/ericdb.

EXAMPLE

creatdb demo
creatdb −ueric −q erics_db
creatdb −e +q −c −u:av erics_db

FILES

.../files/dbtmplt6.3
.../files/data/base/*
.../files/datadir/* (for compatibility with previous versions)

SEE ALSO

demodb(unix), destroydb(unix), users(files), chmod(I), destroydb(quel), permit(quel)

DIAGNOSTICS

No database name specified.
: You have not specified the name of the database to create (or modify) with the command.

You may not access this database
: Your entry in the users file says you are not authorized to access this database.

You are not a valid INGRES user
: You do not have a users file entry, and can not do anything with INGRES at all.

You are not allowed this command
: The U_CREATDB bit is not set in your users file entry.

You may not use the −u flag
: Only the INGRES superuser may become someone else.

<name> does not exist
: With **−c** or **−m**, the directory does not exist.

<name> already exists
: Without either **−e** or **−m**, the database (actually, the directory) already exists.

<name> is not empty
: With the **−m** flag, the directory you named must be empty.

You are not the DBA for this database
: With the **−e** flag, you must be the database administrator.

NAME

destroydb – destroy an existing database

SYNOPSIS

destroydb [**–s**] [**–m**] dbname

DESCRIPTION

Destroydb will remove all reference to an existing database. The directory of the database and all files in that directory will be removed.

To execute this command the current user must be the database administrator for the database in question, or must be the INGRES superuser and have the **–s** flag stated.

The **–m** flag causes *destroydb* not to remove the UNIX directory. This is useful when the directory is a separate mounted UNIX file system.

EXAMPLE

destroydb demo
destroydb –s erics_db

FILES

.../data/base/*
.../datadir/* (for compatibility with previous versions)

SEE ALSO

creatdb(unix)

DIAGNOSTICS

invalid dbname – the database name specified is not a valid name.
you may not reference this database – the database may exist, but you do not have permission to do anything with it.
you may not use the –s flag – you have tried to use the **–s** flag, but you are not the INGRES superuser.
you are not the dba – someone else created this database.
database does not exist – this database does not exist.

NAME

equel – Embedded QUEL interface to C

SYNOPSIS

equel [**–d**] [**–f**] [**–r**] file.q ...

DESCRIPTION

Equel provides the user with a method of interfacing the general purpose programming language "C" with INGRES. It consists of the EQUEL pre-compiler and the EQUEL runtime library.

Compilation

The precompiler is invoked with the statement:

> **equel** [<flags>] file1.q [<flags>] file2.q ...

where file*n*.q are the source input file names, which must end with **.q**. The output is written to the file "file*n*.c". As many files as wished may be specified. The flags that may be used are:

–d Generate code to print source listing file name and line number when a run-time error occurs. This can be useful for debugging, but takes up process space. Defaults to off.

–f Forces code to be on the same line in the output file as it is in the input file to ease interpreting C diagnostic messages. EQUEL will usually try to get all C code lines in the output file on the same lines as they were in the input file. Sometimes it must break up queries into several lines to avoid C-preprocessor line overflows, possibly moving some C code ahead some lines. With the **–f** flag specified this will never happen and, though the line buffer may overflow, C lines will be on the right line. This is useful for finding the line in the source file that C error diagnostics on the output file refer to.

–r Resets flags to default values. Used to supress other flags for some of the files in the argument list.

The output files may than be compiled using the C compiler:

> **cc** file1.c file2.c . **–lq**

The **–lq** requests the use of the EQUEL object library.

All EQUEL routines and globals begin with the characters "II", and so all globals variables and procedure names of the form II*xxx* are reserved for use by EQUEL and should be avoided by EQUEL users.

Basic Syntax

EQUEL commands are indicated by lines which begin with a double pound sign ("##"). Other lines are simply copied as is. All normal INGRES commands may be used in EQUEL and have the same effect as if invoked through the interactive terminal monitor. Only retrieve commands with no result relation specified have a different syntax and meaning.

The format of retrieve without a result relation is modified to:

```
## retrieve (C-variable = a_fcn { , C-variable = a_fcn } )
```

optionally followed (immediately) by:

```
##      [ where qual ]
## {
        /* C-code */
## }
```

This statement causes the "C-code" to be executed once for each tuple retrieved, with the "C-variable"s set appropriately. Numeric values of any type are converted as necessary. No conversion is done between numeric and character values. (The normal INGRES *ascii* function may be used for this purpose.)

Also, the following EQUEL commands are permitted.

ingres [ingres flags] data_base_name

This command starts INGRES running, and directs all dynamically following queries to the database *data_base_name*. It is a run-time error to execute this command twice without an intervening "## exit", as well as to issue queries while an "## ingres" statement is not in effect. Each flag should be enclosed in quotes to avoid confusion in the EQUEL parser:

```
## ingres "-f4f10.2" "-i2l2" demo
```

exit

Exit simply exits from INGRES. It is equivalent to the **\q** command to the teletype monitor.

Parametrized Quel Statements

Quel statements with target lists may be "parametrized". This is indicated by preceding the statement with the keyword "param". The target list of a parametrized statement has the form:

(*tl_var*, *argv*)

where *tl_var* is taken to be a string pointer at execution time (it may be a string constant) and interpreted as follows. For any parametrized EQUEL statement except a retrieve without a result relation (no "into rel") (i.e. append, copy, create, replace, retrieve into) the string *tl_var* is taken to be a regular target list except that wherever a '%' appears a valid INGRES type (f4, f8, i2, i4, c) is expected to follow. Each of these is replaced by the value of the corresponding entry into *argv* (starting at 0) which is interpreted to be a pointer to a variable of the type indicated by the '%' sequence. Neither *argv* nor the variables which it points to need be declared to EQUEL. For example:

```
    char    *argv[10];

            argv[0] = &double_var;
            argv[1] = &int_var;
    ##      param append to rel
    ##              ("dom1 = %f8, dom2 = %i2", argv)
    ##      /* to escape the "%<ingres_type>" mechanism use "%%" */
    ##      /* This places a single '%' in the string. */
```

On a retrieve to C-variables, within *tl_var*, instead of the C-variable to retrieve into, the same '%' escape sequences are used to denote the type of the corresponding argv entry into which the value will be retrieved.

The qualification of any query may be replaced by a string valued variable, whose contents is interpreted at run time as the text of the qualification.

The *copy* statement may also be parametrized. The form of the parametrized *copy* is analogous to the other parametrized statements: the target list may be parametrized in the same manner as the *append* statements, and furthermore, the **from/into** keyword may be replaced by a string valued variable whose content at run time should be **into** or **from**.

Declarations

Any valid C variable declaration on a line beginning with a "##" declares a C-

variable that may be used in an EQUEL statement and as a normal variable. All variables must be declared before being used. Anywhere a constant may appear in an INGRES command, a C-variable may appear. The value of the C-variable is substituted at execution time.

Neither nested structures nor variables of type *char* (as opposed to pointer to char or array of char) are allowed. Furthermore, there are two restrictions in the way variables are referenced within EQUEL statements. All variable usages must be dereferenced and/or subscripted (for arrays and pointers), or selected (for structure variables) to yield lvalues (scalar values). *Char* variables are used by EQUEL as a means to use *strings*. Therefore when using a *char* array or pointer it must be dereferenced only to a "*char* *". Also, variables may not have parentheses in their references. For example:

```
## struct xxx
## {
        int     i;
##      int     *ip;
## }    **struct_var;

/* not allowed */
##      delete p where p.ifield = *(*struct_var)->ip

/* allowed */
##      delete p where p.ifield = *struct_var[0]->ip
```

C variables declared to EQUEL have either global or local scope. Their scope is local if their declaration is within a free (not bound to a retrieve) block declared to EQUEL. For example:

```
/* globals scope variable */
## int Gint;

func(i)
int     i;
## {
        /* local scope variable */
##      int     *gintp;
        ...
## }
```

If a variable of one of the char types is used almost anywhere in an EQUEL statement the content of that variable is used at run time. For example:

```
##      char    *dbname[MAXDATABASES + 1];
        int     current_db;

        dbname[current_db] = "demo";
##      ingres dbname[current_db]
```

will cause INGRES to be invoked with data base "demo". However, if a variable's name is to be used as a constant, then the non-referencing operator '#' should be used. For example:

```
## char        *demo;

        demo = "my_database";

        /* ingres -d my_database */
##      ingres "-d" demo

        /* ingres -d demo */
##      ingres "-d" #demo
```

The C-preprocessor's #include feature may be used on files containing equel statements and declarations if these files are named *anything*.**q.h**. An EQUEL processed version of the file, which will be #included by the C-preprocessor, is left in *anything*.**c.h**.

Errors and Interrupts

INGRES and run-time EQUEL errors cause the routine **IIerror** to be called, with the error number and the parameters to the error in an array of string pointers as in a C language main routine. The error message will be looked up and printed. before printing the error message, the routine (*IIprint_err)() is called with the error number that ocurred as its single argument. The error message corresponding to the error number returned by (*IIprint_err)() will be printed. Printing will be supressed if (*IIprint_err)() returns 0. IIprint_err may be reassigned to, and is useful for programs which map INGRES errors into their own error messages. In addition, if the "**-d**" flag was set the file name and line number of the error will be printed. The user may write an IIerror routine to do other tasks as long as the setting of IIerrflag is not modified as this is used to exit retrieves correctly.

Interrupts are caught by equel if they are not being ignored. This insures that the rest of INGRES is in sync with the EQUEL process. There is a function pointer, IIinterrupt, which points to a function to call after the interrupt is caught. The user may use this to service the interrupt. It is initialized to "exit()" and is called with -1 as its argument. For example:

```
extern int (*IIinterrupt)();
extern reset();

setexit();
IIinterrupt = reset;
mainloop();
```

To ignore interrupts, signal() should be called before the ## ingres statement is executed.

FILES

.../files/error6.3_*
    Can be used by the user to decipher INGRES error numbers.
/lib/libq.a
    Run time library.

SEE ALSO

.../doc/other/equeltut.q, C reference manual, ingres(UNIX), quel(QUEL)

BUGS

The C-code embedded in the tuple-by-tuple retrieve operation may not contain additional QUEL statements or recursive invocations of INGRES.

There is no way to specify an i1 format C-variable.

Includes of an equel file within a parameterized target list, or within a C variable's array subscription brackets, isn't done correctly.

NAME

helpr – get information about a database.

SYNOPSIS

helpr [**–u***name*] [±**w**] database relation ...

DESCRIPTION

Helpr gives information about the named relation(s) out of the database specified, exactly like the *help* command.

Flags accepted are **–u** and ±**u**. Their meanings are identical to the meanings of the same flags in INGRES.

SEE ALSO

ingres(unix), help(quel)

DIAGNOSTICS

bad flag – you have specified a flag which is not legal or is in bad format.
you may not access database – this database is prohibited to you based on status information in the users file.
cannot access database – the database does not exist.

NAME

ingres — INGRES relational data base management system

SYNOPSIS

ingres [*flags*] dbname [process_table]

DESCRIPTION

This is the UNIX command which is used to invoke INGRES. Dbname is the name of an existing data base. The optional flags have the following meanings (a "±" means the flag may be stated "+*x*" to set option *x* or "−*x*" to clear option *x*. "−" alone means that "−*x*" must be stated to get the *x* function):

±U Enable/disable direct update of the system relations and secondary indicies. You must have the 000004 bit in the status field of the users file set for this flag to be accepted. This option is provided for system debugging and is strongly discouraged for normal use.

−u*name* Pretend you are the user with login name *name* (found in the users file). If *name* is of the form :*xx*, *xx* is the two character user code of a user. This may only be used by the DBA for the database or by the INGRES superuser.

−c*N* Set the minimum field width for printing character domains to *N*. The default is 6.

−i*lN* Set integer output field width to *N*. *l* may be 1, 2, or 4 for i1's, i2's, or i4's repectively.

−f*lxM.N* Set floating point output field width to *M* characters with *N* decimal places. *l* may be 4 or 8 to apply to f4's or f8's respectively. *x* may be **e**, **E**, **f**, **F**, **g**, **G**, **n**, or **N** to specify an output format. **E** is exponential form, **F** is floating point form, and **G** and **N** are identical to **F** unless the number is too big to fit in that field, when it is output in **E** format. **G** format guarantees decimal point alignment; **N** does not. The default format for both is **n10.3**.

−v*X* Set the column seperator for retrieves to the terminal and print commands to be *X*. The default is vertical bar.

−r*M* Set modify mode on the *retrieve* command to *M*. *M* may be **isam**, **cisam**, **hash**, **chash**, **heap**, **cheap**, **heapsort**, or **cheapsort**, for ISAM, compressed ISAM, hash table, compressed hash table, heap, compressed heap, sorted heap, or compressed sorted heap. The default is "cheapsort".

−n*M* Set modify mode on the *index* command to *M*. *M* can take the same values as the **−r** flag above. Default is "isam".

±a Set/clear the autoclear option in the terminal monitor. It defaults to set.

±b Set/reset batch update. Users must the 000002 bit set in the status field of the users file to clear this flag. This flag is normally set. When clear, queries will run slightly faster, but no recovery can take place. Queries which update a secondary index automatically set this flag for that query only.

±d Print/don't print the dayfile. Normally set.

±s Print/don't print any of the monitor messages, including prompts. This flags is normally set. If cleared, it also clears the **−d** flag.

±w Wait/don't wait for the database. If the **+w** flag is present, INGRES will wait if certain processes are running (purge,restore, and/or sysmod) on the given data base. Upon completion of those processes INGRES will proceed. If the **−w** flag is present, a message is returned and execution stopped if the data base is not available. If the **±w** flag is omitted and the data base is unavailable, the error message is returned if INGRES is running in foreground (more precisely if the standard input is from a terminal), otherwise the wait option is invoked.

Process_table is the pathname of a UNIX file which may be used to specify the run-time configuration of INGRES. This feature is intended for use in system maintenance only, and its unenlightened use by the user community is strongly discouraged.

Note: It is possible to run the monitor as a batch-processing interface using the '<', '>' and '|' operators of the UNIX shell, provided the input file is in proper monitor-format.

EXAMPLE

ingres demo
ingres −d demo
ingres −s demo < batchfile
ingres −f4g12.2 −i13 +b −rhash demo

FILES

.../files/users – valid INGRES users
.../data/base/* – data bases
.../datadir/* – for compatability with previous versions
.../files/proctab6.3 – runtime configuration file

SEE ALSO

monitor(quel)

DIAGNOSTICS

Too many options to INGRES – you have stated too many flags as INGRES options.

Bad flag format – you have stated a flag in a format which is not intelligible, or a bad flag entirely.

Too many parameters – you have given a database name, a process table name, and "something else" which INGRES doesn't know what to do with.

No database name specified

Improper database name – the database name is not legal.

You may not access database *name* – according to the users file, you do not have permission to enter this database.

You are not authorized to use the *flag* flag – the flag specified requires some special authorization, such as a bit in the users file, which you do not have.

Database *name* does not exist

You are not a valid INGRES user – you have not been entered into the users file, which means that you may not use INGRES at all.

You may not specify this process table – special authorization is needed to specify process tables.

Database temporarily unavailable – someone else is currently performing some operation on the database which makes it impossible for you to even log in. This condition should disappear shortly.

NAME

printr – print relations

SYNOPSIS

printr [*flags*] database relation ...

DESCRIPTION

Printr prints the named relation(s) out of the database specified, exactly like the *print* command. Retrieve permission must be granted to all people to execute this command.

Flags accepted are **–u**, ±**w**, **–c**, **–i**, **–f**, and **–v**. Their meanings are identical to the meanings of the same flags in INGRES.

SEE ALSO

ingres(unix), print(quel)

DIAGNOSTICS

bad flag – you have specified a flag which is not legal or is in bad format.
you may not access database – this database is prohibited to you based on status information in the users file.
cannot access database – the database does not exist.

NAME

purge – destroy all expired and temporary relations

SYNOPSIS

purge [**−f**] [**−p**] [**−a**] [**−s**] [±**w**] [database ...]

DESCRIPTION

Purge searches the named databases deleting system temporary relations. When using the **−p** flag, expired user relations are deleted. The **−f** flag will cause unrecognizable files to be deleted, normally purge will just report these files.

Only the database administrator (the DBA) for a database may run purge, except the INGRES superuser may purge any database by using the **−s** flag.

If no databases are specified all databases for which you are the DBA will be purged. All databases will be purged if the INGRES superuser has specified the **−s** flag. The **−a** flag will cause purge to print a message about the pending operation and execute it only if the response if a 'y'. Any other response is interpreted as "no".

Purge will lock the data base while it is being processed, since errors may occur if the database is active while purge is working on the database. If a data base is busy *purge* will report this and go on to the next data base, if any. If standard input is not a terminal *purge* will wait for the data base to be free. If **−w** flag is stated *purge* will not wait, regardless of standard input. The **+w** flag causes *purge* to always wait.

EXAMPLES

purge −p +w tempdata
purge −a −f

SEE ALSO

save(quel), restore(unix)

DIAGNOSTICS

who are you? – you are not entered into the users file.

not ingres superuser – you have tried to use the **−s** flag but you are not the INGRES superuser.

you are not the dba – you have tried to purge a database for which you are not the DBA.

cannot access database – the database does not exist.

BUGS

If no database names are given, only the databases located in the directory **data/base** are purged, and not the old databases in **datadir**. Explicit database names still work for databases in either directory.

NAME

restore – recover from an INGRES or UNIX crash.

SYNOPSIS

restore [**–a**] [**–s**] [±**w**] [database ...]

DESCRIPTION

Restore is used to restore a data base after an INGRES or UNIX crash. It should always be run after any abnormal termination to ensure the integrity of the data base.

In order to run restore, you must be the DBA for the database you are restoring or the INGRES superuser and specify the **–s** flag.

If no databases are specified then all databases for which you are the DBA are restored. All databases will be restored if the INGRES superuser has specified the **–s** flag.

If the **–a** flag is specified you will be asked before restore takes any serious actions. It is advisable to use this flag if you suspect the database is in bad shape. Using /dev/null as input with the **–a** flag will provide a report of problems in the data base. If there were no errors while restoring a database, *purge* will be called, with the same flags that were given to *restore*, to remove unwanted files and system temporaries. *Restore* may be called with the **–f** and/or **–p** flags for *purge*. Unrecognized files and expired relations are not removed unless the proper flags are given. In the case of an incomplete destroy, create or index *restore* will not delete files for partially created or destroyed relations. *Purge* must be called with the **–f** flag to accomplish this.

Restore locks the data base while it is being processed. If a data base is busy *restore* will report this and go on to the next data base. If standard input is not a terminal *restore* will wait for the data base to be free. If the **–w** flag is set *restore* will not wait regardless of standard input. If **+w** is set it will always wait.

Restore can recover a database from an update which had finished filling the batch file. Updates which did not make it to this stage should be rerun. Similarly modifies which have finished recreating the relation will be completed (the relation relation and attribute relations will be updated). If a destroy was in progress it will be carried to completion, while a create will almost always be backed out. Destroying a relation with an index should destroy the index so *restore* may report that a secondary relation has been found with no primary.

If interrupt (signal 2) is received the current database is closed and the next, if any, is processed. Quit (signal 3) will cause restore to terminate.

EXAMPLE

restore –f demo
restore –a grants < /dev/null

DIAGNOSTICS

All diagnostics are followed by a tuple from a system relations.

"No relation for attribute(s)" – the attributes listed have no corresponding entry in the relation relation

"No primary relation for index" – the tuple printed is the relation tuple for a secondary index for which there is no primary relation. The primary probably was destroyed the secondary will be.

"No indexes entry for primary relation" – the tuple is for a primary relation, the relindxd domain will be set to zero. This is the product of an incomplete destroy.

"No indexes entry for index" – the tuple is for a secondary index, the index will be destroyed. This is the product of an incomplete destroy.

"*relname* is index for" – an index has been found for a primary which is not marked as indexed. The primary will be so marked. This is probably the product of an incomplete index command. The index will have been created properly but not modified.

"No file for" – There is no data for this relation tuple, the tuple will be deleted. If, under the **–a** option, the tuple is not deleted purge will not be called.

"No secondary index for indexes entry" – An entry has been found in the indexes relation for which the secondary index does not exist (no relation relation tuple). The entry will be deleted.

SEE ALSO

purge(unix)

BUGS

If no database names are given, only the databases located in the directory **data/base** are restored, and not the old databases in **datadir**. Explicit database names still work for databases in either directory.

NAME

sysmod – modify system relations to predetermined storage structures.

SYNOPSIS

sysmod [−s] [**−w**] dbname [**relation**] [**attribute**] [**indexes**] [**tree**] [**protect**] [**integrities**]

DESCRIPTION

Sysmod will modify the relation, attribute, indexes, tree, protect, and integrities relations to hash unless at least one of the **relation, attribute, indexes, tree, protect,** or **integrities** parameters are given, in which case only those relations given as parameters are modified. The system relations are modified to gain maximum access performance when running INGRES. The user must be the data base administrator for the specified database, or be the INGRES superuser and have the **−s** flag stated.

Sysmod should be run on a data base when it is first created and periodically thereafter to maintain peak performance. If many relations and secondary indices are created and/or destroyed, *sysmod* should be run more often.

If the data base is being used while *sysmod* is running, errors will occur. Therefore, *sysmod* will lock the data base while it is being processed. If the data base is busy, *sysmod* will report this. If standard input is not a terminal *sysmod* will wait for the data base to be free. If **−w** flag is stated *sysmod* will not wait, regardless of standard input. The **+w** flag causes *sysmod* to always wait.

The system relations are modified to hash; the relation relation is keyed on the first domain, the indexes, attribute, protect, and integrities relations are keyed on the first two domains, and the tree relation is keyed on domains one, two, and five. The relation and attribute relations have the minpages option set at 10, the indexes, protect, and integrities relations have the minpages value set at 5.

SEE ALSO

modify(quel)

NAME

usersetup – setup users file

SYNOPSIS

.../bin/usersetup [pathname]

DESCRIPTION

The **/etc/passwd** file is read and reformatted to become the INGRES users file, stored into **.../files/users.** If *pathname* is specified, it replaces "...". If pathname is "–", the result is written to the standard output.

The user name, user, and group id's are initialized to be identical to the corresponding entry in the **/etc/passwd** file. The status field is initialized to be zero, except for user **ingres**, which is initialized to all permission bits set. The "initialization file" parameter is set to the file **.ingres** in the user's login directory. The user code field is initialized with sequential two-character codes. All other fields are initialized to be null.

After running *usersetup*, the **users** file must be edited. Any users who are to have any special authorizations should have the status field changed, according to the specifications in users(files). To disable a user from executing INGRES entirely, completely remove her line from the users file.

As UNIX users are added or deleted from the **/etc/passwd** file, the **users** file will need to be editted to reflect the changes. For deleted users, it is only necessary to delete the line for that user from the **users** file. To add a user, you must assign that user a code in the form "aa" and enter a line in the users file in the form:

name:cc:uid:gid:status:flags:proctab:initfile::databases

where *name* is the user name (taken from the first field of the **/etc/passwd** file entry for this user), *cc* is the user code assigned, which must be exactly two characters long and must not be the same as any other existing user codes, *uid* and *gid* are the user and group ids (taken from the third and fourth fields in the **/etc/passwd** entry), *status* is the status bits for this user, normally 000000, *flags* are the default flags for INGRES (on a per-user basis), *proctab* is the default process table for this user (which defaults to =**proctab6.3**), and *databases* is a list of the databases this user may enter. If null, she may use all databases. If the first character is a dash ("–"), the field is a comma separated list of databases which she may not enter. Otherwise, it is a list of databases which she may enter.

The *databases* field includes the names of databases which may be created.

Usersetup may be executed only once, to initially create the **users** file.

FILES

.../files/users
/etc/passwd

SEE ALSO

ingres(unix), passwd(V), users(files)

BUGS

It should be able to bring the **users** file up to date.

NAME

.../files/dayfile6.1 – INGRES login message

DESCRIPTION

The contents of the dayfile reflect user information of general system interest, and is more or less analogous to **/etc/motd** in UNIX. The file has no set format; it is simply copied at login time to the standard output device by the monitor if the **–s** or **–d** options have not been requested. Moreover the dayfile is not mandatory, and its absence will not generate errors of any sort; the same is true when the dayfile is present but not readable.

NAME

.../files/dbtmplt6.3 – database template

DESCRIPTION

This file contains the template for a database used by *creatdb*. It has a set of entries for each relation to be created in the database. The sets of entries are separated by a blank line. Two blank lines or an end of file terminate the file.

The first line of the file is the database status and the default relation status, separated by a colon. The rest of the file describes relations. The first line of each group gives the relation name followed by an optional relation status, separated by a colon. The rest of the lines in each group are the attribute name and the type, separated by a tab character.

All the status fields are given in octal, and have a syntax of a single number followed by a list of pairs of the form

$\pm x \pm N$

which says that if the $\pm x$ flag is asserted on the *creatdb* command line then set (clear) the bits specified by *N*.

The first set of entries must be for the relation catalog, and the second set must be for the attribute catalog.

EXAMPLE

```
3-c-1+q+2:010023
relation:-c-20
relid	c12
relowner	c2
relspec	i1

attribute:-c-20
attrelid	c12
attowner	c2
attname	c12

	(other relation descriptors)
```

SEE ALSO

creatdb(unix)

NAME

.../files/error6.3_? – files with INGRES errors

DESCRIPTION

These files contain the INGRES error messages. There is one file for each thousands digit; hence, error number 2313 will be in file error6.3_2.

Each file consists of a sequence of error messages with associated error numbers. When an error enters the front end, the appropriate file is scanned for the correct error number. If found, the message is printed; otherwise, the first message parameter is printed.

Each message has the format
errnum <TAB> message tilde.
Messages are terminated by the tilde character ('~'). The message is scanned before printing. If the sequence *%n* is encountered (where *n* is a digit from 0 to 9), parameter *n* is substituted, where %0 is the first parameter.

The parameters can be in any order. For example, an error message can reference %2 before it references %0.

SEE ALSO

error(UTIL)

EXAMPLE

```
1003	line %0, bad database name %1~
1005	In the purge of %1,
a bad %0 caused execution to halt~
1006	No process, try again.~
```

NAME

libq — Equel run-time support library

DESCRIPTION

Libq all the routines necessary for an equel program to load. It resides in **/lib/libq.a**, and must be specified when loading equel pre-processed object code. It may be referenced on the command line of *cc* by the abbreviation **–lq**.

Several useful routines which are used by equel processes are included in the library. These may be employed by the equel programmer to avoid code duplication. They are:

```
int     IIatoi(buf, i)
char    *buf;
int     i;

char    *IIbmove(source, destination, len)
char    *source, *destination;
int     len;

char    *IIconcatv(buf, arg1, arg2, ..., 0)
char    *buf, *arg1, ...;

char    *IIitos(i)
int     i;

int     IIsequal(s1, s2)
char    *s1, *s2;

int     IIlength(string)
char    *string;

IIsyserr(string, arg1, arg2, ...);
char    *string;
```

IIatoi	IIatoi is equivalent to atoi(UTIL).
IIbmove	Moves *len* bytes from *source* to *destination*, returning a pointer to the location after the last byte moved. Does not append a null byte.
IIconcatv	Concatenates into *buf* all of its arguments, returning a pointer to the null byte at the end of the concatenation. *Buf* may not be equal to any of the arg-n but arg1.
IIitos	IIitos is equivalent to itoa(III).
IIsequal	Returns 1 iff strings s1 is identical to s2.
IIlength	Returns max(length of *string* without null byte at end, 255)
IIsyserr	IIsyserr is diferrent from syserr(util) only in that it will print the name in IIproc_name, and in that there is no 0 mode. Also, it will always call exit(−1) after printing the error message.

There are also some global Equel variables which may be manipulated by the user:

```
int       IIerrflag;
char      *IImainpr;
char      (*IIprint_err)();
int       IIret_err();
```

int IIno_err();

IIerrflag Set on an error from INGRES to be the error number (see the error message section of the "INGRES Reference Manual") that ocurred. This remains valid from the time the error occurrs to the time when the next equel statement is issued. This may be used just after an equel statement to see if it succeded.

IImainpr This is a string which determines which ingres to call when a "## ingres" is issued. Initially it is "/usr/bin/ingres".

IIprint_err This function pointer is used to call a function which determines what (if any) error message should be printed when an ingres error occurs. It is called from IIerror() with the error number as an argument, and the error message corresponding to the error number returned will be printed. If (*IIprint_err)(<*errno*>) returns 0, then no error message will be printed. Initially IIprint_err is set to IIret_err() to print the error that ocurred.

IIret_err Returns its single integer argument. Used to have (*IIprint_err)() cause printing of the error that ocurred.

IIno_err Returns 0. Used to have (*IIprint_err)() suppress error message printing. IIno_err is used when an error in a parametrized equel statemenr occurs to suppress printing of the corresponding parser error.

SEE ALSO

atoi(util), bmove(util), cc(I), equel(unix), exit(II), itoa(III), length(util), sequal(util), syserr(util)

NAME

.../files/proctab6.3 – INGRES runtime configuration information

DESCRIPTION

The file **.../files/proctab6.3** describes the runtime configuration of the INGRES system.

The process table is broken up into logical sections, separated by lines with a single dollar sign. The first section describes the configuration of the system and the parameters to pass to each system module. All other sections contain strings which may be macro-substituted into the first section.

Each line of the first section describes a single process. The lines consist of a series of colon-separated fields.

The first field contains the pathname of the module to be executed.

The second field is a set of flags which allow the line to be ignored in certain cases. If this field is null, the line is accepted; otherwise, it should be a series of items of the form "+–=*X*", where any of "+–=" may be omitted, and *X* is a flag which may appear on the INGRES command line. The characters "+–=" are interpreted as the sense of the flag: "+" will accept the line if the flag "+*X*" is stated on the command line, "–" will accept if "–*X*" is stated, and "=" will accept if the "*X*" flag is not stated at all. These may be combined in the forms "+=" and "–=". For example, the field:

+=&

will acept the line if the EQUEL flag ("&") is stated as "+&" or is not stated, but the line will be ignored if the "–&" flag is stated.

If any flag item rejects the line, the entire line is rejected.

The third field is a status word. The number in this word is expressed in octal. The bits have the following meaning:

000010	close diagnostic output
000004	close standard input
000002	run in user's directory, not database
000001	run as the user, not as INGRES

The fourth field is a file name to which the standard output should be redirected. It is useful for debugging.

The fifth field describes the pipes which should be connected to this process. The field must be six characters long, with the characters corresponding to the internal variables R_up, W_up, R_down, W_down, R_front, and W_front respectively. The characters may be a question mark, which leaves the pipe closed; a digit, which is filled in from the file descriptors provided by the EQUEL flag or the "@" flag; or a lower case letter, which is connected via a pipe with any other pipes having the same letter; this last action is done on the fly to conserve file descriptors.

The sixth and subsequent fields are arbitrary parameters to be sent to the modules. There must be a colon after the fifth field even if no parameters are present, but there need not be a colon after the last parameter.

The last module executed (usually the last line in the first section) becomes the parent of all the other processes.

The second through last sections of the process table consist of a single line which names the section followed by arbitrary text. The pathname field and all parameter fields of each line of the first section are scanned for strings of the form "$name"; this string is replaced by the text from the corresponding section. For convenience, the name **$pathname** is automatically defined to be the

pathname of the root of the INGRES subtree.

The DBU routines want to see two parameters. The first parameter is the pathname of the "ksort" routine. The second parameter is a series of lines of the form:

command_name:place_list

where command_name is the name of one of the possible INGRES commands executed by the DBU routines, and place_list is a comma-separated list of the actual location(s) of that command. Each "place" is a two-character descriptor: the first character is the overlay in which that command resides, and the second character is the function within that overlay. If a command is in more than one place, INGRES will try to avoid calling in another overlay. For example:

create:a0,m1

means that the *create* command may be found in overlay "a" function 0 or in overlay "m" function 1. If already in overlay "a" or "m" the create command resident in that overlay will be called; otherwise, overlay "a" will be called.

EXAMPLE

The following example will execute a three process system unless the "&" flag is specified (as "−&"), when a two-process system will be executed with the monitor dropped out and the calling (EQUEL) program in its place. Notice that there are two lines for the parser entry, one for the EQUEL case and one for the non-EQUEL case. In the EQUEL case, output from the parser is diverted to a file called "debug.out".

```
$pathname/bin/overlaya::000014::bc??23:$pathname/bin/ksort:$dbutab
$pathname/bin/parser:+=&:000014::adcb??:
$pathname/bin/parser:-&:000014:debug.out:01cb??:
$pathname/bin/monitor:+=&:000003::??da??:$pathname/files/startup
$
dbutab
create:a0,m1
destroy:a1,m2
modify:m0
help:a2
$
```

NAME

.../files/startup – INGRES startup file

DESCRIPTION

This file is read by the monitor at login time. It is read before the user startup file specified in the users file. The primary purpose is to define a new editor and/or shell to call with the **\e** or **\s** commands.

SEE ALSO

monitor(quel), users(files)

NAME

.../files/users – INGRES user codes and parameters

DESCRIPTION

This file contains the user information in fields seperated by colons. The fields are as follows:

* User name, taken directly from **/etc/passwd** file.
* User code, assigned by the INGRES super-user. It must be a unique two character code.
* UNIX user id. This MUST match the entry in the **/etc/passwd** file.
* UNIX group id. Same comment applies.
* Status word in octal. Bit values are:

0000001	creatdb permission
0000002	permits batch update override
0000004	permits update of system catalogs
0000020	can use trace flags
0000040	can turn off qrymod
0000100	can use arbitrary proctabs
0000200	can use the =proctab form
0100000	ingres superuser

* A list of flags automatically set for this user.
* The process table to use for this user.
* An initialization file to read be read by the monitor at login time.
* Unassigned.
* Comma seperated list of databases. If this list is null, the user may enter any database. If it begins with a '–', the user may enter any database except the named databases. Otherwise, the user may only enter the named databases.

EXAMPLE

ingres:aa:5:2:177777:–d:=special:/mnt/ingres/.ingres::
guest:ah:35:1:000000:::::demo,guest

SEE ALSO

initucode(util)

NAME

Error messages introduction

DESCRIPTION

This document describes the error returns which are possible from the INGRES data base system and gives an explanation of the probable reason for their occurrence. In all cases the errors are numbered *nxxx* where *n* indicates the source of the error, according to the following table:

1 = EQUEL preprocessor
2 = parser
3 = query modification
4 = decomposition and one variable query processor
5 = data base utilities
30 = GEO-QUEL errors

For a description of these routines the reader is referred to *The Design and Implementation of INGRES*. The *xxx* in an error number is an arbitrary identifier.

The error messages are stored in the file **.../files/error6.3_*n***, where *n* is defined as above. The format of these files is the error number, a tab character, the message to be printed, and the tilde character ("~") to delimit the message.

In addition many error messages have "%*i*" in their body where *i* is a digit interpreted as an offset into a list of parameters returned by the source of the error. This indicates that a parameter will be inserted by the error handler into the error return. In most cases this parameter will be self explanatory in meaning.

Where the error message is thought to be completely self explanatory, no additional description is provided.

NAME

EQUEL error message summary

SYNOPSIS

Error numbers 1000 – 1999.

DESCRIPTION

The following errors can be generated at run time by EQUEL programs.

ERRORS

1000 In domain %0 numeric retrieved into char field.

Equel does not support conversion at run-time of numeric data from the data base to character string representation. Hence, if you attempt to assign a domain of numeric type to a C-variable of type character string, you will get this error message. To convert numerics to characters use the "ascii" function in QUEL.

1001 Numeric overflow during retrieve on domain %0.

You will get this error if you attempt to assign a numeric data base domain to a C-variable of a numeric type but with a shorter length. In this case the conversion routines may generate an overflow. For example, this error will result from an attempt to retrieve a large floating point number into a C-variable of type integer.

1002 In domain %0, character retrieved into numeric variable.

This error is the converse of error 1000.

1003 Bad type in target list of parameterized retrieve "%0".

Valid types are %f8, %f4, %i4, %i2, %c.

1004 Bad type in target list of parameterized statement "%0".

Valid types are %f8, %f4, %i4, %i2, %i1, %c.

NAME

Parser error message summary

SYNOPSIS

Error numbers 2000 – 2999.

DESCRIPTION

The following errors can be generated by the parser. The parser reads your query and translates it into the appropriate internal form; thus, almost all of these errors indicate syntax or type conflict problems.

ERRORS

2100 line %0, Attribute '%1' not in relation '%2'

This indicates that in a given line of the executed workspace the indicated attribute name is not a domain in the indicated relation.

2103 line %0, Function type does not match type of attribute '%1'

This error will be returned if a function expecting numeric data is given a character string or vice versa. For example, it is illegal to take the SIN of a character domain.

2106 line %0, Data base utility command buffer overflow

This error will result if a utility command is too long for the buffer space allocated to it in the parser. You must shorten the command or recompile the parser.

2107 line %0, You are not allowed to update this relation: %1

This error will be returned if you attempt to update any system relation or secondary index directly in QUEL (such as the RELATION relation). Such operations which compromise the integrity of the data base are not allowed.

2108 line %0, Invalid result relation for APPEND '%1'

This error message will occur if you execute an append command to a relation that does not exist, or that you cannot access. For example, append to junk(...) will fail if junk does not exist.

2109 line %0, Variable '%1' not declared in RANGE statement

Here, a symbol was used in a QUEL expression in a place where a tuple variable was expected and this symbol was not defined via a RANGE statement.

2111 line %0, Too many attributes in key for INDEX

A secondary index may have no more than 6 keys.

2117 line %0, Invalid relation name '%1' in RANGE statement

You are declaring a tuple variable which ranges over a relation which does not exist.

2118 line %0, Out of space in query tree - Query too long

You have the misfortune of creating a query which is too long for the parser to digest. The only options are to shorten the query or recompile the parser to have more buffer space for the query tree.

2119 line %0, MOD operator not defined for floating point or character attributes

The *mod* operator is only defined for integers.

2120 line %0, no pattern match operators allowed in the target list

Pattern match operators (such as "*") can only be used in a qualification.

2121 line %0, Only character type domains are allowed in CONCAT operator

2123 line %0, '%1.all' not defined for replace

2125 line %0, Cannot use aggregates ("avg" or "avgu") on character values
2126 line %0, Cannot use aggregates ("sum" or "sumu") on character values
2127 line %0, Cannot use numerical functions (ATAN, COS, GAMMA, LOG, SIN, SQRT, EXP, ABS) on character values
2128 line %0, Cannot use unary operators ("+" or "−") on character values
2129 line %0, Numeric operations (+ − * /) not allowed on character values

Many functions and operators are meaningless when applied to character values.

2130 line %0, Too many result domains in target list

Maximum number of result domains is MAXDOM (currently 49).

2132 line %0, Too many aggregates in this query

Maximum number of aggregates allowed in a query is MAXAGG (currently 49).

2133 line %0, Type conflict on relational operator

It is not legal to compare a character type to a numeric type.

2134 line %0, '%1' is not a constant operator.
Only 'dba' or 'usercode' are allowed.

2135 line %0, You cannot duplicate the name of an existing relation(%1)

You have tried to create a relation which would redefine an existing relation. Choose another name.

2136 line %0, There is no such hour as %1, use a 24 hour clock system
2137 line %0, There is no such minute as %1, use a 24 hour clock system
2138 line %0, There is no such time as 24:%1, use a 24 hour clock system

Errors 2136-38 indicate that you have used a bad time in a *permit* statement. Legal times are from 0:00 to 24:00 inclusive.

2139 line %0, Your database does not support query modification

You have tried to issue a query modification statement (*define*), but the database was created with the −q flag. To use the facilities made available by query modification, you must say:

creatdb −e +q *dbname*

to the shell.

2500 syntax error on line %0
last symbol read was: %1

A 2500 error is reported by the parser if it cannot otherwise classify the error. One common way to obtain this error is to omit the required parentheses around the target list. The parser reports the last symbol which was obtained from the scanner. Sometimes, the last symbol is far ahead of the actual place where the error occurred. The string "EOF" is used for the last symbol when the parser has read past the query.

2502 line %0, The word '%1', cannot follow a RETRIEVE command, therefore the command was not executed.

2503 line %0, The word '%1', cannot follow an APPEND command, therefore the command was not executed.

2504 line %0, The word '%1', cannot follow a REPLACE command, therefore the command was not executed.

2505 line %0, The word '%1', cannot follow a DELETE command, therefore the command was not executed.

2506 line %0, The word '%1', cannot follow a DESTROY command, therefore the command was not executed.

2507 line %0, The word '%1', cannot follow a HELP command, therefore the command was not executed.

2508 line %0, The word '%1', cannot follow a MODIFY command, therefore the command was not executed.

2509 line %0, The word '%1', cannot follow a PRINT command, therefore the command was not executed.

2510 line %0, The word '%1', cannot follow a RETRIEVE UNIQUE command, therefore the command was not executed.

2511 line %0, The word '%1', cannot follow a DEFINE VIEW command, therefore the command was not executed.

2512 line %0, The word '%1', cannot follow a HELP VIEW, HELP INTEGRITY, or HELP PERMIT command, therefore the command was not executed.

2513 line %0, The word '%1', cannot follow a DEFINE PERMIT command, therefore the command was not executed.

2514 line %0, The word '%1', cannot follow a DEFINE INTEGRITY command, therefore the command was not executed.

2515 line %0, The word '%1', cannot follow a DESTROY INTEGRITY or DESTROY PERMIT command, therefore the command was not executed.

Errors 2502 through 2515 indicate that after an otherwise valid query, there was something which could not begin another command. The query was therefore aborted, since this could have been caused by misspelling **where** or something equally as dangerous.

2700 line %0, non-terminated string

You have omitted the required string terminator (").

2701 line %0, string too long

Somehow, you have had the persistence or misfortune to enter a character string constant longer than 255 characters.

2702 line %0, invalid operator

You have entered a character which is not alphanumeric, but which is not a defined operator, for example, "?".

2703 line %0, Name too long '%1'

In INGRES relation names and domain names are limited to 12 characters.

2704 line %0, Out of space in symbol table - Query too long

Your query is too big to process. Try breaking it up with more **\go** commands.

2705 line %0, non-terminated comment

You have left off the comment terminator symbol ("*/").

2707 line %0, bad floating constant: %1

Either your floating constant was incorrectly specified or it was too large or too small. Currently, overflow and underflow are not checked.

2708 line %0, control character passed in pre-converted string

In EQUEL a control character became embedded in a string and was not caught until the scanner was processing it.

2709 line %0, buffer overflow in converting a number

Numbers cannot exceed 256 characters in length. This shouldn't become a problem until number formats in INGRES are increased greatly.

NAME

Query Modification error message summary

SYNOPSIS

Error numbers 3000 – 3999.

DESCRIPTION

These error messages are generated by the Query Modification module. These errors include syntactic and semantic problems from view, integrity, and protection definition, as well as run time errors – such as inability to update a view, or a protection violation.

ERRORS

3310 %0 on view %1: cannot update some domain

You tried to perform operation %0 on a view; however, that update is not defined.

3320 %0 on view %1: domain occurs in qualification of view

It is not possible to update a domain in the qualification of a view, since this could cause the tuple to disappear from the view.

3330 %o on view %1: update would result in more than one query

You tried to perform some update on a view which would update two underlying relations.

3340 %0 on view %1: views do not have TID's

You tried to use the Tuple IDentifier field of a view, which is undefined.

3350 %0 on view %1: cannot update an aggregate value

You cannot update a value which is defined in the view definition as an aggregate.

3360 %0 on view %1: that update might be non-functional

There is a chance that the resulting update would be non-functional, that is, that it may have some unexpected side effects. INGRES takes the attitude that it is better to not try the update.

3490 INTEGRITY on %1: cannot handle aggregates yet

You cannot define integrity constraints which include aggregates.

3491 INTEGRITY on %1: cannot handle multivariable constraints

You cannot define integrity constraints on more than a single variable.

3492 INTEGRITY on %1: constraint does not initially hold

When you defined the constraint, there were already tuples in the relation which did not satisfy the constraint. You must fix the relation so that the constraint holds before you can declare the constraint.

3493 INTEGRITY on %1: is a view

You can not define integrity constraints on views.

3494 INTEGRITY on %1: You must own '%1'

You must own the relation when you declare integrity constraints.

3500 %0 on relation %1: protection violation

You have tried to perform an operation which is not permitted to you.

3590 PERMIT: bad terminal identifier "%2"

In a *permit* statement, the terminal identifier field was improper.

3591 PERMIT: bad user name "%2"

You have used a user name which is not defined on the system.

3592 PERMIT: Relation '%1' not owned by you

You must own the relation before issuing protection constraints.

3593 PERMIT: Relation '%1' must be a real relation (not a view)

You can not define permissions on views.

3594 PERMIT on %1: bad day-of-week '%2'

The day-of-week code was unrecognized.

3595 PERMIT on %1: only the DBA can use the PERMIT statement

Since only the DBA can have shared relations, only the DBA can issue *permit* statements.

3700 Tree buffer overflow in query modification
3701 Tree build stack overflow in query modification

Bad news. An internal buffer has overflowed. Some expression is too large. Try making your expressions smaller.

NAME

One Variable Query Processor error message summary

SYNOPSIS

Error numbers 4000 – 4499.

DESCRIPTION

These error messages can be generated at run time. The One Variable Query Processor actually references the data, processing the tree produced by the parser. Thus, these error messages are associated with type conflicts detected at run time.

ERRORS

4100 OVQP query list overflowed

This error is produced in the unlikely event that the internal form of your interaction requires more space in the one variable query processor than has been allocated for a query buffer. There is not much you can do except shorten your interaction or recompile OVQP with a larger query buffer.

4101 numeric operation using char and numeric domains not allowed

Occasionally, you will be notified by OVQP of such a type mismatch on arithmetic operations. This only happens if the parser has not recognized the problem.

4102 unary operators are not allowed on character values
4103 binary operators cannot accept combinations of char and numeric fields
4104 cannot use aggregate operator "sum" on character domains
4105 cannot use aggregate operator "avg" on character domains

These errors indicate type mismatches – such as trying to add a number to a character string.

4106 the interpreters stack overflowed -- query too long
4107 the buffer for ASCII and CONCAT commands overflowed

More buffer overflows.

4108 cannot use arithmetic operators on two character fields
4109 cannot use numeric values with CONCAT operator

You have tried to perform a numeric operation on character fields.

4110 floating point exception occurred.

If you have floating point hardware instead of the floating point software interpreter, you will get this error upon a floating point exception (underflow or overflow). Since the software interpreter ignores such exceptions, this error is only possible with floating point hardware.

4111 character value cannot be converted to numeric due to incorrect syntax.

When using int1, int2, int4, float4, or float8 to convert a character to value to a numeric value, the character value must have the proper syntax. This error will occur if the character value contained non-numeric characters.

4112 ovqp query vector overflowed

Similar to error 4100.

4199 you must convert your 6.0 secondary index before running this query!

The internal format of secondary indices was changed between versions 6.0 and 6.1 of INGRES. Before deciding to use a secondary index OVQP checks that it is not a 6.0 index. The solution is to destroy the secondary index and recreate it.

NAME

Decomposition error message summary

SYNOPSIS

Error numbers 4500 – 4999.

DESCRIPTION

These error messages are associated with the process of decomposing a multi-variable query into a sequence of one variable queries which can be executed by OVQP.

ERRORS

4602 query involves too many relations to create aggregate function intermediate result.

In the processing of aggregate functions it is usually necessary to create an intermediate relation for *each* aggregate function. However, no query may have more than ten variables. Since aggregate functions implicitly increase the number of variables in the query, you can exceed this limit. You must either break the interaction apart and process the aggregate functions separately or you must recompile INGRES to support more variables per query.

4610 Query too long for available buffer space (qbufsize).
4611 Query too long for available buffer space (varbufsiz)
4612 Query too long for available buffer space (sqsiz)
4613 Query too long for available buffer space (stacksiz)
4614 Query too long for available buffer space (agbufsiz).

These will happen if the internal form of the interaction processed by decomp is too long for the available buffer space. You must either shorten your interaction or recompile decomp. The name in parenthesis gives the internal name of which buffer was too small.

4615 Aggregate function is too wide or has too many domains.

The internal form of an aggregate function must not contain more than 49 domains or be more than 498 bytes wide. Try breaking the aggregate function into two or more parts.

4620 Target list for "retrieve unique" has more than 49 domains or is wider than 498 bytes.

NAME

Data Base Utility error message summary

SYNOPSIS

Error numbers 5000 – 5999

DESCRIPTION

The Data Base Utility functions perform almost all tasks which are not directly associated with processing queries. The error messages which they can generate result from some syntax checking and a considerable amount of semantic checking.

ERRORS

5001 PRINT: bad relation name %0

You are trying to print a relation which doesn't exist.

5002 PRINT: %0 is a view and can't be printed

The only way to print a view is by retrieving it.

5003 PRINT: Relation %0 is protected.

You are not authorized to access this relation.

5102 CREATE: duplicate relation name %0

You are trying to create a relation which already exists.

5103 CREATE: %0 is a system relation

You cannot create a relation with the same name as a system relation. The system depends on the fact that the system relations are unique.

5104 CREATE %0: invalid attribute name %1

This will happen if you try to create a relation with an attribute longer than 12 characters.

5105 CREATE %0: duplicate attribute name %1

Attribute names in a relation must be unique. You are trying to create one with a duplicated name.

5106 CREATE %0: invalid attribute format "%2" on attribute %1

The allowed formats for a domain are c1–c255, i1, i2, i4, f4 and f8. Any other format will generate this error.

5107 CREATE %0: excessive domain count on attribute %1

A relation cannot have more than 49 domains. The origin of this magic number is obscure. This is very difficult to change.

5108 CREATE %0: excessive relation width on attribute %1

The maximum number of bytes allowed in a tuple is 498. This results from the decision that a tuple must fit on one UNIX "page". Assorted pointers require the 14 bytes which separates 498 from 512. This "magic number" is very hard to change.

5201 DESTROY: %0 is a system relation

The system would immediately stop working if you were allowed to do this.

5202 DESTROY: %0 does not exist or is not owned by you

To destroy a relation, it must exist, and you must own it.

5203 DESTROY: %0 is an invalid integrity constraint identifier

Integers given do not identify integrity constraints on the specified relation. For example: If you were to type "destroy permit parts 1, 2, 3", and 1, 2, or 3 were not the numbers "help permit parts" prints out for permissions on parts, you would get this error.

5204 DESTROY: %0 is an invalid protection constraint identifier

Integers given do not identify protection constraints on the specified relation. Example as for error 5203.

5300 INDEX: cannot find primary relation

The relation does not exist – check your spelling.

5301 INDEX: more than maximum number of domains

A secondary index can be created on at most six domains.

5302 INDEX: invalid domain %0

You have tried to create an index on a domain which does not exist.

5303 INDEX: relation %0 not owned by you

You must own relations to put indicies on them.

5304 INDEX: relation %0 is already an index

INGRES does not permit tertiary indicies.

5305 INDEX: relation %0 is a system relation

Secondary indices cannot be created on system relations.

5306 INDEX: %0 is a view and an index can't be built on it

Since views are not physically stored in the database, you cannot build indicies on them.

5401 HELP: relation %0 does not exist

5402 HELP: cannot find manual section "%0"

Either the desired manual section does not exist, or your system does not have any on-line documentation.

5403 HELP: relation %0 is not a view

Did a "help view" (which prints view definition) on a nonview. For example: "help view overpaidv" prints out overpaidv's view definition.

5404 HELP: relation %0 has no permissions on it granted
5405 HELP: relation %0 has no integrity constraints on it

You have tried to print the permissions or integrity constraints on a relation which has none specified.

5410 HELP: tree buffer overflowed
5411 HELP: tree stack overflowed

Still more buffer overflows.

5500 MODIFY: relation %0 does not exist

5501 MODIFY: you do not own relation %0

You cannot modify the storage structure of a relation you do not own.

5502 MODIFY %0: you may not provide keys on a heap

By definition, heaps do not have keys.

5503 MODIFY %0: too many keys provided

You can only have 49 keys on any relation.

5504 MODIFY %0: cannot modify system relation

System relations can only be modified by using the *sysmod* command to the shell; for example

sysmod ***dbname***

5507 MODIFY %0: duplicate key "%1"

You may only specify a domain as a key once.

5508 MODIFY %0: key width (%1) too large for isam

When modifying a relation to isam, the sum of the width of the key fields cannot exceed 245 bytes.

5510 MODIFY %0: bad storage structure "%1"

The valid storage structure names are heap, cheap, isam, cisam, hash, and chash.

5511 MODIFY %0: bad attribute name "%1"

You have specified an attribute that does not exist in the relation.

5512 MODIFY %0: "%1" not allowed or specified more than once

You have specified a parameter which conflicts with another parameter, is inconsistant with the storage mode, or which has already been specified.

5513 MODIFY %0: fillfactor value %1 out of bounds

Fillfactor must be between 1 and 100 percent.

5514 MODIFY %0: minpages value %1 out of bounds

Minpages must be greater than zero.

5515 MODIFY %0: "%1" should be "fillfactor", "maxpages", or "minpages"

You have specified an unknown parameter to *modify*.

5516 MODIFY %0: maxpages value %1 out of bounds

5517 MODIFY %0: minpages value exceeds maxpages value

5518 MODIFY %0: invalid sequence specifier "%1" for domain %2.

Sequence specifier may be "ascending" (or "a") or "descending" (or "d") in a *modify*. For example:

```
modify parts to heapsort on
pnum:ascending,
pname:descending
```

5519 MODIFY: %0 is a view and can't be modified

Only physical relations can be modified.

5520 MODIFY: %0: sequence specifier "%1" on domain %2 is not allowed with the specified storage structure.

Sortorder may be supplied only when modifying to **heapsort** or **cheapsort**.

5600 SAVE: cannot save system relation "%0"

System relations have no save date and are guaranteed to stay for the lifetime of the data base.

5601 SAVE: bad month "%0"
5602 SAVE: bad day "%0"
5603 SAVE: bad year "%0"

This was a bad month, bad day, or maybe even a bad year for .

5604 SAVE: relation %0 does not exist or is not owned by you

5800 COPY: relation %0 doesn't exist

5801 COPY: attribute %0 in relation %1 doesn't exist or it has been listed twice

5803 COPY: too many attributes

Each dummy domain and real domain listed in the copy statement count as one attribute. The limit is 150 attributes.

5804 COPY: bad length for attribute %0. Length="%1"

5805 COPY: can't open file %0

On a copy "from", the file is not readable by the user.

5806 COPY: can't create file %0

On a copy "into", the file is not creatable by the user. This is usually caused by the user not having write permission in the specified directory.

5807 COPY: unrecognizable dummy domain "%0"

On a copy "into", a dummy domain name is used to insert certain characters into the unix file. The domain name given is not valid.

5808 COPY: domain %0 size too small for conversion.
There were %2 tuples successfully copied from %3 into %4

When doing any copy except character to character, copy checks that the field is large enough to hold the value being copied.

5809 COPY: bad input string for domain %0. Input was "%1". There were %2 tuples successfully copied from %3 into %4

This occurs when converting character strings to integers or floating point numbers. The character string contains something other than numeric characters (0-9,+,−,blank,etc.).

5810 COPY: unexpected end of file while filling domain %0.
There were %1 tuples successfully copied from %2 into %3

5811 COPY: bad type for attribute %0. Type="%1"

The only accepted types are i, f, c, and d.

5812 COPY: The relation "%0" has a secondary index. The index(es) must be destroyed before doing a copy "from"

Copy cannot update secondary indices. Therefore, a copy "from" cannot be done on an indexed relation.

5813 COPY: You are not allowed to update the relation %0

You cannot copy into a system relation or secondary index.

5814 COPY: You do not own the relation %0.

You cannot use copy to update a relation which you do not own. A copy "into" is allowed but a copy "from" is not.

5815 COPY: An unterminated "c0" field occurred while filling domain %0. There were %1 tuples successfully copied from %2 into %3

A string read on a copy "from" using the "c0" option cannot be longer than 1024 characters.

5816 COPY: The full pathname must be specified for the file %0

The file name for copy must start with a "/".

5817 COPY: The maximum width of the output file cannot exceed 1024 bytes per tuple

The amount of data to be output to the file for each tuple exceeds 1024. This usually happens only if a format was mistyped or a lot of large dummy domains were specified.

5818 COPY: %0 is a view and can't be copied

Only physical relations can be copied.

5819 COPY: Warning: %0 duplicate tuples were ignored.

On a copy "from", duplicate tuples were present in the relation.

5820 COPY: Warning: %0 domains had control characters which were converted to blanks.

5821 COPY: Warning: %0 c0 character domains were truncated.

Character domains in c0 format are of the same length as the domain length. You had a domain value greater than this length, and it was truncated.

5822 COPY: Relation %0 is protected.

You are not authorized to access this relation.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

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

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **/etc/termcap** database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Screen Package

Contents

1 Overview 1
- 1.1 Terminology (or, Words You Can Say to Sound Brilliant) 1
- 1.2 Compiling Things 1
- 1.3 Screen Updating 1
- 1.4 Naming Conventions 2

2 Variables 2

3 Usage 3
- 3.1 Starting up 3
- 3.2 The Nitty-Gritty 3
  - 3.2.1 Output 3
  - 3.2.2 Input 4
  - 3.2.3 Miscellaneous 4
- 3.3 Finishing up 4

4 Cursor Motion Optimization: Standing Alone 4
- 4.1 Terminal Information 4
- 4.2 Movement Optimizations, or, Getting Over Yonder 5

5 The Functions 6
- 5.1 Output Functions 6
- 5.2 Input Functions 9
- 5.3 Miscellaneous Functions 10
- 5.4 Details 13

Appendixes

Appendix A 14

1 Capabilities from termcap 14
- 1.1 Disclaimer 14
- 1.2 Overview 14
- 1.3 Variables Set By setterm() 14
- 1.4 Variables Set By gettmode() 15

Appendix B 16

1 The WINDOW structure 16

Appendix C 17

1 Examples 17

2 Screen Updating 17
- 2.1 Twinkle 17
- 2.2 Life 19

3 Motion optimization 22
- 3.1 Twinkle 22

1. Overview

In making available the generalized terminal descriptions in **/etc/termcap**, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*:

screen: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file **<curses.h>** needs to include **<sgtty.h>**, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [*flags*] file ... **−lcurses −ltermlib**

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the ter-

---

[1] The screen package also uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. It is redundant (but harmless) for the programmer to do it, too.

minal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal*. Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for **w**indow-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (e.g., *reg int i;*)
bool	boolean type, actually a "char" (e.g., *bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must ***always*** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *crmode()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order

to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *gettmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **eye** and **vi**[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing **cat**(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **/etc/termcap** database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For

[3] Eye actually uses these functions, **vi** does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as **rocket** and **gun**.

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by **gtty**(2) and **stty**(2) The second, *setterm()*, a larger task by reading in the descriptions from the **/etc/termcap** database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
      gettmode();
      if (sp=getenv("TERM"))
            setterm(sp);
}
else
      setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a **gtty**(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **/etc/termcap** associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see **termcap**(3)) to put out a string). It is these things which *endwin()* undoes.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the **termlib**(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS−1, LINES−1, 0)
```

---

[6] These names are identical to those variables used in the **/etc/termcap** database to describe each capability. See Appendix A for a complete list of those read, and **termcap**(5) for a full description.

[7] *isatty()* is defined in the default C library function routines. It does a **gtty**(2) on the descriptor and checks the return value.

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "**w**" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

addch(ch) †
char ch;

waddch(win, ch)
*WINDOW *win;*
char ch;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str) †
*char *str;*

waddstr(win, str)
*WINDOW *win;*
*char *str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
*WINDOW *win;*
char vert, hor;

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
*WINDOW *win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
WINDOW **scr;*
bool *boolf;*

Sets the clear flag for the screen *scr.* If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh(),* or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear(),* does not alter the contents of the screen. If *scr* is *curscr.* the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol() †

wclrtoeol(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)
WINDOW **win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
WINDOW **win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
WINDOW **win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y*, *x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
*WINDOW *win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
*char *str;*

wgetstr(win, str)
*WINDOW *win;*
*char *str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
*char *fmt;*

wscanw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

delwin(win)
*WINDOW *win;*

[9] UNIX is a trademark of Bell Laboratories.

delwin(win)
*WINDOW *win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

getyx(win, y, x) †
*WINDOW *win;*
int y, x;

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
*WINDOW *win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

leaveok(win, boolf) †
*WINDOW *win;*
bool boolf;

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
*char *termbuf, *name;*

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

mvwin(win, y, x)
WINDOW **win;*
int *y, x;*

Move the home position of the window *win* from its current starting coordinates to (*y*, *x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything.

WINDOW *
newwin(lines, cols, begin_y, begin_x)
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y*, *begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or (*COLS* − *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
WINDOW **win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW *
subwin(win, lines, cols, begin_y, begin_x)
WINDOW **win;*
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y*, *begin_x*) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y*, *begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or

(*COLS* − *begin_x*) respectively.

unctrl(ch) †
char ch;

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (*lasty*, *lastx*) to (*newy*, *newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

tstp()

If the new **tty(4)** driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n

variables set by *setterm()*

Type	Name	Pad	Description
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence[10]
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with *X* are reserved for severely nauseous glitches

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

[10] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm()*

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
# define              WINDOW struct _win_st

struct _win_st {
                short    _cury, _curx;
                short    _maxy, _maxx;
                short    _begy, _begx;
                short    _flags;
                bool     _clear;
                bool     _leave;
                bool     _scroll;
                char     **_y;
                short    *_firstch;
                short    *_lastch;
};

# define              _SUBWIN         01
# define              _ENDLINE        02
# define              _FULLWIN        04
# define              _SCROLLWIN      010
# define              _STANDOUT       0200
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury*, *_curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy*, *_begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

_y[i]

is a pointer to the *i*th line, and

_y[i][j]

is the *j*th character on the *i*th line.

_flags can have one or more values or'd into it. **_SUBWIN** means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. **_ENDLINE** says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_STANDOUT** says that all characters added to the screen are in standout mode.

---

[11] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

Appendix C

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
# include        <curses.h>
# include        <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define         NCOLS   80
# define         NLINES  24
# define         MAXPATTERNS    4

struct locs {
        char     y, x;
};

typedef struct locs          LOCS;

LOCS    Layout[NCOLS * NLINES];     /* current board layout */

int     Pattern,                    /* current pattern number */
        Numstars;                   /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                      /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);
```

```
	for (;;) {
		makeboard();			/* make the board setup */
		puton('*');			/* put on '*'s */
		puton(' ');			/* cover up with ' 's */
	}
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

	signal(SIGINT, SIG_IGN);
	mvcur(0, COLS-1, LINES-1, 0);
	endwin();
	exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard() {

	reg int		y, x;
	reg LOCS	*lp;

	Pattern = rand() % MAXPATTERNS;
	lp = Layout;
	for (y = 0; y < NLINES; y++)
		for (x = 0; x < NCOLS; x++)
			if (ison(y, x)) {
				lp->y = y;
				lp++->x = x;
			}
	Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int	y, x; {

	switch (Pattern) {
	  case 0:		/* alternating lines */
		return !(y & 01);
```

```
                case 1:                 /* box */
                        if (x >= LINES && y >= NCOLS)
                                return FALSE;
                        if (y < 3 || y >= NLINES - 3)
                                return TRUE;
                        return (x < 3 || x >= NCOLS - 3);
                case 2:                 /* holy pattern! */
                        return ((x + y) & 01);
                case 3:                 /* bar across center */
                        return (y >= 9 && y <= 15);
                }
                /* NOTREACHED */
}

puton(ch)
reg char                ch; {

                reg LOCS        *lp;
                reg int         r;
                reg LOCS        *end;
                LOCS            temp;

                end = &Layout[Numstars];
                for (lp = Layout; lp < end; lp++) {
                        r = rand() % Numstars;
                        temp = *lp;
                        *lp = Layout[r];
                        Layout[r] = temp;
                }

                for (lp = Layout; lp < end; lp++) {
                        mvaddch(lp->y, lp->x, ch);
                        refresh();
                }
}
```

2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include       <curses.h>
# include       <signal.h>

/*
 *              Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

struct lst_st {                                 /* linked list element */
```

```
		int			y, x;			/* (y, x) position of piece */
		struct lst_st		*next, *last;		/* doubly linked */
};

typedef struct lst_st			LIST;

LIST		*Head;					/* head of linked list */

main(ac, av)
int		ac;
char		*av[]; {

		int		die();

		evalargs(ac, av);			/* evaluate arguments */

		initscr();				/* initialize screen package */
		signal(SIGINT, die);			/* set to restore tty stats */
		crmode();				/* set for char-by-char */
		noecho();				/*		input */
		nonl();					/* for optimization */

		getstart();				/* get starting position */
		for (;;) {
			prboard();			/* print out current board */
			update();			/* update board position */
		}
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die() {

		signal(SIGINT, SIG_IGN);		/* ignore rubouts */
		mvcur(0, COLS-1, LINES-1, 0);		/* go to bottom of screen */
		endwin();				/* set terminal to initial state */
		exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart() {

		reg char		c;
		reg int			x, y;
```

```
        box(stdscr, '|', '_');                      /* box in the screen */
        move(1, 1);                                 /* move to upper left corner */

        do {
                refresh();                          /* print current position */
                if ((c=getch()) == 'q')
                        break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
                  case '.':
                        adjustyx(c);
                        break;
                  case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                  case 'x':
                        addch('X');
                        break;
                  case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                                       /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}

/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;
```

```
        erase();                                /* clear out last position */
        box(stdscr, '|', '_');                  /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

        reg char        *sp;
        char            *getenv();
        int             _putchar(), die();

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
             gettmode();
             if (sp=getenv("TERM"))
                        setterm(sp);
                signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}
```

```
/*
 * _putchar defined for tputs() (and _puts())
 */
_putchar(c)
reg char        c; {

        putchar(c);
}

puton(ch)
char    ch; {

        static int      lasty, lastx;
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
                        /* prevent scrolling */
                if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                        mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                        lasty = lp->y;
                        if ((lastx = lp->x + 1) >= NCOLS)
                                if (AM) {
                                        lastx = 0;
                                        lasty++;
                                }
                                else
                                        lastx = NCOLS - 1;
                }
}
```

NAME

curses - screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files **−lcurses −ltermcap** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold, stty(2), setenv(3), termcap(5)

AUTHOR

Ken Arnold

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*

raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	"change" all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

WRITING PAPERS WITH NROFF USING −ME

Eric P. Allman

Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes the text processing facilities available on the UNIX† operating system via NROFF† and the −me macro package. It is assumed that the reader already is generally familiar with the UNIX 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 UNIX 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. Finally, section 6 discusses things you will need to know if you want to typeset documents. 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 UNIX 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

†UNIX, NROFF, and TROFF are Trademarks of Bell Laboratories

.sp 4

spaces four lines. The number **4** is an *argument* to the **.sp** request which says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...

will be read, packed onto output lines, and justified to produce:

Now is the time for all good men to come to the aid of their party.
Four score and seven years ago,...

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph. To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe ("′") as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, 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. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as "mother-in-law"); NROFF is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1. 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 I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

> Now is the time for all good men
> to come to the aid of their party. Four score and seven years ago,...

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. 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 may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he ''%''
.fo 'Jane Jones''My Book'
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request **.ls 2**, as is done in this section. You can revert to single spaced mode by typing **.ls 1**.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values

you wish to use; bold characters represent characters 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 should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R. The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

```
Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent
          book containing translations of most of Confucius' most
          delightful sayings. A definite must for anyone interested in
          the early foundations of Chinese philosophy.
```

Text lines can be centered by using the **.ce** request. The line after the **.ce** is centered (horizontally) on the page. To center more than one line, use **.ce** *N* (where *N* is the number of lines to center), followed by the *N*

lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines to center
.ce 0
```

The **.ce 0** request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use **.br**.

2.5. Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the **.ce** request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. 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.

3.1. 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,...

3.2. 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
```

will produce:
Alternatives to avoid deadlock are:

> Lock in a specified order
> Detect deadlock and back out one process
> Lock all resources needed before proceeding

3.3. Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

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. When this is not appropriate, you can use the alternative, called *floating keeps*.

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3". A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear 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. The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

3.4. Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type **.(l F** (Throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This kind of display will be indented from both margins. For example, the input:

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1.  Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

```
.(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
```

will be 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 may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests **.(c** and **.)c**. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

first line of unfilled display
more lines

If the block requests (**.(b** and **.)b**) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. 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]

It is important that the footnote appears *inside* the quote, so that you can

[1]Like this.

[2]James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.

be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use ***#** 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.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type **.)x** "", which specifies an explicitly null page number.

The **.xp** request prints the index.

For example, the input:

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

Sealing wax 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

*Such as an asterisk.

two lines. 9

The **.(x** request may have a single character argument, specifying the "name" of the index; the normal index is **x**. Thus, several "indicies" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. 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.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using **.lp** instead of **.pp**, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the **.ip** request. A word specified on the same line as **.ip** is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to **.ip** appears in the margin.

We can continue text without starting a new indented paragraph by using the **.lp** request.

If you have spaces in the label of a **.ip** request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to **.ip**) is longer than the space allocated for the label, **.ip** will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlabel
This paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* **ii**. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to **.ip**. Refer to the reference manual for more information.

If **.ip** is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
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.

5.2. 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.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

```
.sh 3 "Another section" 7 3 4
```

will begin the section numbered **7.3.4**; all subsequent **.sh** requests will number relative to this number.

There are more complex features which will 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 0.5i
```

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.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The **.tp** request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(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 will produce, 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, which begins the preliminary part of the paper. After issuing this request, the **.+c** request will begin a preliminary

section of the paper. Most notably, this prints the page number restarted from one in lower case Roman numbers. **.+c** may be used repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request **.++ B** may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in figure 2. (In this figure, comments begin with the sequence \".)

```
.th                        \" set for thesis mode
.fo ''DRAFT''              \" define footer for each page
.tp                        \" begin title page
.(l C                      \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                        \" end centered part
.+c INTRODUCTION           \" begin chapter named "INTRODUCTION"
.(x t                      \" make an entry into index 't'
Introduction
.)x                        \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"         \" begin another chapter
.(x t                      \" enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B                      \" begin bibliographic information
.+c BIBLIOGRAPHY           \" begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P                      \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t                      \" print index 't' collected above
.+c PREFACE                \" begin another preliminary section
text of preface
```

Figure 2. Outline of a Sample Paper

5.4. Equations and Tables

Two special UNIX programs exist to format special types of material. **Eqn** and **neqn** set equations for the phototypesetter and NROFF respectively. **Tbl** arranges to print extremely pretty tables in a variety of formats. This document will only describe the embellishments to the standard features; consult the reference manuals for those processors for a description of their use.

The **eqn** and **neqn** programs are described fully in the document *Typesetting Mathematics – Users' Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept on one page. They are introduced by the **.EQ** request and terminated by the **.EN** request.

The **.EQ** request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The **C** on the **.EN** request specifies that the equation will be continued.

The **tbl** program produces tables. It is fully described (including numerous examples) in the document *Tbl – A Program to Format Tables* by M. E. Lesk. Tables begin with the **.TS** request and end with the **.TE** request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request **.TS H** and put the request **.TH** after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request **.2c**. This causes everything after it to be output in two-column form. The request **.bc** will start 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**.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line **.de** *xx* (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line *.xx* is the same as stating

all the other lines. For example, to define a macro that spaces 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 may 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**.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index **f** (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string **\!** at the beginning of all the lines dealing with the index. In other words, you should use:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with **.(b** and **.)b**) as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting

program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the **.ul** request is set in italics in TROFF.

There are ways of switching between fonts. The requests **.r**, **.i**, and **.b** switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must 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 you do want a quote mark to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

"Master Control"

in italics, you must type:

```
.i """Master Control\|"""
```

The **\|** produces a very narrow space so that the "l" does not overlap the quote sign in TROFF, like this:

"Master Control"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

underlined
bold italics
|words in a box|

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line TROFF would produce ***some bold italics*** and |words in a box|, which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

.b bold face

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

boldface.

To set the two words **bold** and **face** both in **bold face**, type:

.b "bold face"

You can mix fonts in a word by using the special sequence **\c** at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space inbetween them. For example, the input:

.u under \c
.i italics

generates <u>under</u>*italics*, but if we had typed:

.u under
.i italics

the result would have been <u>under</u> *italics* as two words.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the 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.

6.3. 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, you may 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"

will generate "quoted text". Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the −me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on June 17, 1983 and applies to version 1.1 of the −me macros.

An introduction to the C shell

William Joy

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

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

November 8, 1980

†UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

William Joy

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

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX programmer's manual. The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics.* These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *builtin* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or *words* interpreted as a *command name* followed by *arguments*. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a ↑D which sent an end-of-file to the mail program. (Here and throughout this document, the notation "↑*x*" is to be read "control-*x*" and represents the striking of the *x* key while the control key is held down.) The mail program then echoed the characters 'EOT' and transmitted our message. The characters '% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '% ' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ↑D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal — the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the backspace (↑H) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

```
tset −e
```

which tells the program *tset* to set the erase character, and its default setting for this character is a backspace.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '−' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current *working directory*. The option *−s* is the size option, and

```
ls −s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a ↑H, as we demonstrated in section 1.1 how this could be set up.

sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

```
date > #now
```

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail*, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% '.

1.5. Input from files; pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its *standard input.* Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ↑D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline.* For instance the command

```
ls −s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

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

```
ls −s | sort −n
```

specifies that the output of the *ls* command run with the option *−s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *−r* reverse sort option and the *head* command in combination with the previous command doing

```
ls −s | sort −n −r | head −5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory*, which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

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

```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

```
echo prog.*
```

will echo the names

prog.c prog.errs prog.o prog.output

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

echo ? ?? ???

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

prog.[co]

will match

prog.c prog.o

in the example above. We can also place two characters around a '−' in this notation to denote a range. Thus

chap.[1−5]

might match files

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

if they existed. This is shorthand for

chap.[12345]

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list)* contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

No match.

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another users' login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp.* If I give the command

cp thatfile ~

the shell will expand this command to

cp thatfile /usr/bill

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

echo *

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

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '−' in an argument word to a command is to enclose it with single quotation characters ''', i.e.

echo '*'

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within ''' characters. It and the character ''' itself can be preceded by a single '\' to prevent their special meaning. Thus

echo \'\!

prints

'!

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

echo \''*'

which prints

'*

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

1.8. Terminating commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

cat /etc/passwd

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing the DEL or RUBOUT key on your terminal.* Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT

*Many users use *stty*(1) to change the interrupt character to ↑C.

again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ↑D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ↑D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

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

```
mail bill < prepared.text
```

the mail command will terminate without our typing a ↑D. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a ↑Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
↑Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1]  + Stopped              mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ↑Z. When the shell noticed

that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a ↑D which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The ↑Z should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in section 2.6.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing a ↑\. This will usually provoke the shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file 'core' has been created containing information about the program 'a.out's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See section 2.6 for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

```
cat /etc/passwd
```

command will, you can use the command

```
more /etc/passwd
```

The *more* program pauses after each complete screenful and types '--More--' at which point you can hit a space to get another screenful, a return to get another line, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

```
cat /etc/passwd | more
```

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ↑S key to stop the typeout. The typeout will resume when you hit ↑Q or any other key, but ↑Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ↑S and ↑Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ↑O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ↑O is a toggle, so flushing can be turned off by typing ↑O again while output is being flushed.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it**

takes effect at next login. You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset -s -m dialup:c100rv4pna -m plugboard:?hp2621nl *`';
ts; stty intr ↑C kill ↑U crt
set time=15 history=10
msgs -f
if (-e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ↑D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set;* this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ↑C and the line kill character to ↑U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '−f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In

any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

2.2. Shell variables

The shell maintains a set of *variables.* We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings.* Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path.* This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

```
% set
argv      ()
cwd       /usr/bill
home      /usr/bill
path      (. /usr/ucb /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
term      c100rv4pna
user      bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

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

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on

each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

    set ignoreeof

and to unset it do

    unset ignoreeof

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

    > filename

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

    set noclobber

in your *.login* file. Then trying to do

    date > now

would cause a diagnostic if 'now' existed already. You could type

    date >! now

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.†

2.3. The shell's history list

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

†The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

```
% cat bug.c
main()

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

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '−o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls −l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

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

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmers Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '−s'. We can do

```
alias ls ls −s
```

or even

```
alias dir ls −s
```

creating a new command syntax 'dir' which does an 'ls −s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls −s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the

arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

    alias cd 'cd \!* ; ls '

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in '' characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

    alias whois 'grep \!↑ /etc/passwd'

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

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

    command >& file

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

    command |& lpr

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.#

Finally, it is possible to use the form

    command >> file

to place output at the end of an existing file.†

#A command form

    command >&! file

exists, and is used when *noclobber* is set and *file* already exists.

†If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

    command >>! file

makes it not be an error for file to not exist when *noclobber* is set.

2.6. Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls −s | sort −n | head −5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without out waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] − Done                du > usage
%
```

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

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are re-used.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls −s | sort −n > usage &
[2] 2034 2035
%
```

runs the 'ls' program with the '−s' options, pipes this output into the 'sort' program with the '−n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediates prompts for a new command, leaving the job running simultaneously.

As mentioned in section 1.8, foreground jobs become *suspended* by typing ↑Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
↑Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)        sort usage
%
```

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

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

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job — identified by a '−' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%−' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some

unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '−l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls −s | sort −n > myfile &
[2] 3405
% mail bill
↑Z
Stopped
% jobs
[1] − Running          du > usage
[2]   Running          ls −s | sort −n > myfile
[3] + Stopped          mail bill
% fg %ls
ls −s | sort −n > myfile
% more myfile
```

The *fg* command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

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

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

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

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

```
ed bigfile
w
120000
q
%
```

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

The command

```
stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)     wc hugefile
% fg wc
wc hugefile
  13371  30123  302577
% stty −tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the *jobs* command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The *ps* can be used in this case to find out about background jobs not started in the current shell.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir*, creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory *newpaper*. where, for example, he might place a group of related files.

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

```
cd
```

with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

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

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

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *csh* manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs −l
/mnt/bill
% cd myproject
% dirs
˜/myproject
% ed prog.c
1143
↑Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ˜/myproject)
```

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

```
% fg
ed prog.c (wd: ˜/myproject)
 . . . after some editing
q
(wd now: ˜)
%
```

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

2.8. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

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

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*.

By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

    set prompt='\! % '

Note that the '!' character had to be *escaped* here even within '' characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

cputime	unlimited
filesize	unlimited
datasize	5616 kbytes
stacksize	512 kbytes
coredumpsize	unlimited

Limits can be set, e.g.:

    limit coredumpsize 128k

Most reasonable units abbreviations will work; see the *csh* manual page for more details.

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

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

    repeat 5 cat one >> five

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

    setenv TERM adm3a

will set the value of the environment variable TERM to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

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

    source .cshrc

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
    52   178   1347 /etc/rc
    52   178   1347 /usr/bill/rc
   104   356   2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

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

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile.* This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set.* It is the fundamental mechanism used

for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

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

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

```
$argv[1]
```

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

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

would give 'c', and

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

would give 'a b'. Other notations useful in shell scripts are

$*n*

where *n* is an integer as a shorthand for

$argv[*n*]

the *n th* parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the

variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (↑D) was typed.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if *n* is not in the range '1−$#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n−'; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm−n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '==' and '!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

−? filename

where '?' is replace by a number of single characters. For instance the expression primitive

−e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

      if ($i !~ *.c) continue  # not a .c file so do nothing

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

      cmp -s $i ~/backup/$i:t # to set $status

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

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

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

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

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

†The following two formats are not currently acceptable to the shell:

```
if ( expression )          # Won't work!
then
        command
        ...
endif
```

and

```
if ( expression ) then command endif          # Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else−if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier ':r' here to extract a root of a filename or ':e' to extract the *extension*. Thus if the variable *i* has the value '/mnt/foo.bar' then

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

shows how the ':r' modifier strips off the trailing '.bar' and the the ':e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the programmers manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.# Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using ''' or '\' to place it in an argument word.

#It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

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

does not do what one would expect.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

 ...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in ''' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

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

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as ''' does.

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep −c csh$ /etc/passwd
27
% grep −c nsh$ /etc/passwd
128
% grep −c −v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '−v sh$')
? grep −c $i /etc/passwd
? end
27
128
430
%
```

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

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

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

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

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

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

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

4.3. Command substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

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

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

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

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

Appendix — Special characters

The following table lists the special characters of *csh* and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic metacharacters

<table>
<tr><td>;</td><td>2.4</td><td>separates commands to be executed sequentially</td></tr>
<tr><td>|</td><td>1.5</td><td>separates commands in a pipeline</td></tr>
<tr><td>( )</td><td>2.2,3.6</td><td>brackets expressions and variable values</td></tr>
<tr><td>&</td><td>2.5</td><td>follows commands to be executed without waiting for completion</td></tr>
</table>

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.5	indicates redirected input
>	1.3	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	1.3,3.6	begins scratch file names; indicates shell comments
—	1.2	prefixes option (flag) arguments to commands
%	2.6	prefixes job name specifications

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX programmer's manual in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

. — Your current directory has the name '.' as well as the name printed by the command *pwd;* see also *dirs.* The current directory '.' is usually the first *component* of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating *components* of filenames (1.6). The character '.' at the beginning of a *component* of a *pathname* is treated specially and not matched by the *filename expansion* metacharacters '?', '*', and '[' ']' pairs (1.6).

.. — Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir*, i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (2.7).

a.out — Compilers which create executable images create them, by default, in the file *a.out.* for historical reasons (2.3).

absolute pathname — A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system — called the *root* directory. *Pathnames* which are not *absolute* are called *relative* (see definition of *relative pathname*) (1.6).

alias — An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (2.4).

argument — Commands in UNIX receive a list of *argument* words. Thus the command

```
echo a b c
```

consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (1.1).

argv — The list of arguments to a command written in the shell language (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 (3.4).

background — Commands started without waiting for them to complete are called *background* commands (2.6).

base — A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* — the part after the '.'. See *filename* and *extension* (1.6)

bg
: The *bg* command causes a *suspended* job to continue execution in the *background* (2.6).

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' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

break
: *Break* is a builtin command used to exit from loops within the control structure of the shell (3.7).

breaksw
: The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (3.7).

builtin
: A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

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. Details are given in the shell documentation 'csh(1)' (3.7).

cat
: The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).

cd
: The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory (2.4, 2.7).

chdir
: The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

chsh
: The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing

    chsh your-login-name /bin/csh

  Thus I would do

    chsh bill /bin/csh

  It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).

cmp
: *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.

command
: A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory within the UNIX system, is called a *command* (1.1).

command name
: When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution
: The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (4.3).

component
: A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable which has multiple strings as value is said to have several *components*; each string is a *component* of the variable.

continue
: A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

control-
: Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control*-c is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an up-arrow (↑) followed by the corresponding letter when you type a *control* character (e.g. '↑C' for *control*-c (1.8).

core dump
: When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form

    Illegal instruction (core dumped)

  (where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

cp
: The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).

csh
: The name of the shell program that this document describes.

.cshrc
: The file *.cshrc* in your *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 *alias* parameters which are to take effect globally (2.1).

cwd
: The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).

date
: The *date* command prints the current date and time (1.3).

debugging
: *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).

default:
: The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).

DELETE
: The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ↑C.

detached
: A command that continues running in the *background* after you logout is said to be *detached*.

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*, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).

directory — A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).

directory stack — The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left (2.7).

dirs — The *dirs* command prints the shell's *directory stack* (2.7).

du — The *du* command is a program (described in 'du(1)') which prints the number of disk blocks is all directories below and including your current *working directory* (2.6).

echo — The *echo* command prints its arguments (1.6, 3.6).

else — The *else* command is part of the 'if-then-else-endif' control command construct (3.6).

endif — If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true (3.6).

EOF — An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an *end-of-file* when the command sending them input completes. Most commands terminate when they receive an *end-of-file*. The shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

escape — A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended*. Most systems use control-s to stop the output and control-q to start it.

/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 ':' characters (1.8). You can look at this file by saying

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See 'finger(1)', 'passwd(5)', and 'grep(1)' for more details.

exit — The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).

exit status — A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script

to give a non-zero *exit status* (3.6).

expansion
: The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion.* Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions* (1.6, 3.4, 4.2).

expressions
: *Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (3.5).

extension
: Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '−me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).

fg
: The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (1.8, 2.6).

filename
: Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension* (1.6).

filename expansion
: *Filename expansion* uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (1.6, 4.2).

flag
: Many UNIX commands accept arguments which 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 character '−' (1.2). Thus the *ls* (list files) command has an option '−s' to list the sizes of files. This is specified

```
ls −s
```

foreach
: The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

foreground
: When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground.* This is as opposed to *background.* *Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard (1.8, 2.6).

goto
: The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

grep
: The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file */etc/passwd* which contains the string 'bill'.

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' (2.4).

head — The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).
Head is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).

history — The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (2.3).

home directory — Each user has a *home directory*, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~' (1.6).

if — A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).

ignoreeof — Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).

input — Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in *pipelines* will read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).

interrupt — An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key (although users can and often do change the interrupt character, usually to ↑C). It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (1.8, 3.9).

job — One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended* (2.6).

job control — The builtin functions that control the execution of jobs are called *job control* commands. These are *bg, fg, stop, kill* (2.6).

job number — When each job is started it is assigned a small number called a *job number* which is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job (2.6).

jobs — The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (2.6).

kill — A command which sends a signal to a job causing it to terminate (2.6).

.login — The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (2.1).

login shell — The shell that is started on your terminal when you login is called your *login shell.* It is different from other shells which you may run (e.g. on shell scripts) in that it reads the *.login* file before reading commands from the terminal and it reads the *.logout* file after you logout (2.1).

logout — The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file,* but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.8).

.logout — When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.

lpr — The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).

ls — The *ls* (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail — The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.1).

make — The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).

makefile — The file containing commands for *make* is called *makefile* (3.2).

manual — The *manual* often referred to is the 'UNIX programmer's manual'. It contains a number of sections and a description of each UNIX program. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

metacharacter

Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters.* If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted.* 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 (1.4). The appendix to this user's manual lists the *metacharacters* in groups by their function.

mkdir — The *mkdir* command is used to create a new directory.

modifier — Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

more — The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

noclobber — The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).

noglob — The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (3.6).

notify — The *notify* command tells the shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the shell to always report the termination of *background* jobs exactly when they occur (2.6).

onintr — The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (3.9).

output — Many commands in UNIX result in some lines of text which are called their *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 (1.3). 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 (1.5). Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (2.3). 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, but it is possible to direct error diagnostics along with *standard output* using a special metanotation (2.5).

pushd — The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).

path — The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell 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 (. /usr/ucb /bin /usr/bin)

the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

chmod 755 script

was executed to turn this execute permission on (3.3). If you add new commands to a directory in the *path*, you should issue the command *rehash* (2.2).

pathname
: A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* may name a directory, but usually names a file.

pipeline
: A group of commands which are connected together, 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 shell metacharacter '|' (1.5, 2.3).

popd
: The *popd* command changes the shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so (2.7).

port
: The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr
: The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

printenv
: The *printenv* command is used to print the current setting of variables in the environment (2.8).

process
: An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started — called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.

program
: Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.

programmer's manual
: Also referred to as the *manual*. See the glossary entry for 'manual'.

prompt
: Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex(1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).

ps — The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), 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 (2.6). Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

pwd — The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.

quit — The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).

quotation — The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\', is referred to as *quotation* (1.7).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

rehash — The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).

relative pathname

A *pathname* which does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the *working directory*, and subsequent *components* between '/' characters refer to directories below the *working directory*. *Pathnames* that are not *relative* are called *absolute pathnames* (1.6).

repeat — The *repeat* command iterates another command a specified number of times.

root — The directory that is at the top of the entire directory structure is called the *root* directory since it is the 'root' of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is '/'. *Pathnames* starting with '/' are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation (1.6).

RUBOUT — The RUBOUT or DELETE key sends an interrupt to the current job. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell. Users often change interrupt to be generated by ↑C rather than DELETE by using the *stty* command.

scratch file — Files whose names begin with a '#' are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).

script — Sequences of shell 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 in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).

set — The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

setenv — Variables in the environment 'environ(5)' can be changed by using the *setenv* builtin command (2.8). 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 to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular *shell*, called *csh*.

shell script — See *script* (3.3, 3.10).

signal — A *signal* in UNIX is a short message that is sent to a running program which causes something to happen to that process. *Signals* are sent either by typing special *control* characters on the keyboard or by using the *kill* or *stop* commands (1.8, 2.6).

sort — The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (1.5).

source — The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.8).

special character — See *metacharacters* and the appendix to this manual.

standard — We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).

status — A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (3.6).

stop — The *stop* command causes a *background* job to become *suspended* (2.6).

string — A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters (2.2).

stty — The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty(1)' for a complete description (2.6).

substitution — The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (3.4).

suspended — A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control*-z at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command (2.6).

switch — The *switch* command of the shell 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 language C (3.7).

termination — When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end-of-file* from their *standard input*. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified jobs (2.6).

then — The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time — The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command (2.1, 2.8).

tset — The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).

tty — The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.

unalias — The *unalias* command removes aliases (2.8).

UNIX — UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.

unset — The *unset* command removes the definitions of shell variables (2.2, 2.8).

variable expansion
: See *variables* and *expansion* (2.2, 3.4).

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 shell programs (shell command scripts) (2.2).

verbose — The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's *−v* command line option (3.10).

wc — The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).

while — The *while* builtin control construct is used in shell command scripts (3.7).

word — A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '−', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with ''' characters except for the characters ''' and '!' which require special treatment (1.1). This process of placing special characters in *words* without their special meaning is called *quoting*.

working directory
: At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.

write — The *write* command is used to communicate with other users who are logged in to UNIX.

A Guide to the Dungeons of Doom

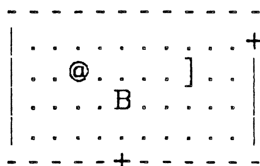
Michael C. Toy
Kenneth C. R. C. Arnold

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Rogue is a visual CRT based fantasy game which runs under the UNIX† timesharing system. This paper describes how to play rogue, and gives a few hints for those who might otherwise get lost in the Dungeons of Doom.

†UNIX is a trademark of Bell Laboratories



Level: 1 Gold: 0   Hp: 12(12) Str: 16(16) Ac: 6 Exp: 1/0

Figure 1

Hp   Your current and maximum hit points. Hit points indicate how much damage you can take before you die. The more you get hit in a fight, the lower they get. You can regain hit points by resting. The number in parentheses is the maximum number your hit points can reach.

Str   Your current strength and maximum ever strength. This can be any integer less than or equal to 31, or greater than or equal to three. The higher the number, the stronger you are. The number in the parentheses is the maximum strength you have attained so far this game.

Ac   Your current armor class. This number indicates how effective your armor is in stopping blows from unfriendly creatures. The lower this number is, the more effective the armor.

Exp   These two numbers give your current experience level and experience points. As you do things, you gain experience points. At certain experience point totals, you gain an experience level. The more experienced you are, the better you are able to fight and to withstand magical attacks.

3.2. The top line

The top line of the screen is reserved for printing messages that describe things that are impossible to represent visually. If you see a "--More--" on the top line, this means that rogue wants to print another message on the screen, but it wants to make certain that you have read the one that is there first. To read the next message, just type a space.

3.3. The rest of the screen

The rest of the screen is the map of the level as you have explored it so far. Each symbol on the screen represents something. Here is a list of what the various symbols mean:

@   This symbol represents you, the adventurer.

-| These symbols represent the walls of rooms.

+   A door to/from a room.

.   The floor of a room.

The floor of a passage between rooms.

*   A pile or pot of gold.

) A weapon of some sort.

] A piece of armor.

! A flask containing a magic potion.

? A piece of paper, usually a magic scroll.

= A ring with magic properties

/ A magical staff or wand

^ A trap, watch out for these.

% A staircase to other levels

: A piece of food.

A-Z The uppercase letters represent the various inhabitants of the Dungeons of Doom. Watch out, they can be nasty and vicious.

4. Commands

Commands are given to rogue by typing one or two characters. Most commands can be preceded by a count to repeat them (e.g. typing "10s" will do ten searches). Commands for which counts make no sense have the count ignored. To cancel a count or a prefix, type <ESCAPE>. The list of commands is rather long, but it can be read at any time during the game with the "?" command. Here it is for reference, with a short explanation of each command.

? The help command. Asks for a character to give help on. If you type a "*", it will list all the commands, otherwise it will explain what the character you typed does.

/ This is the "What is that on the screen?" command. A "/" followed by any character that you see on the level, will tell you what that character is. For instance, typing "/@" will tell you that the "@" symbol represents you, the player.

h, H Move left. You move one space to the left. If you use upper case "h", you will continue to move left until you run into something. This works for all movement commands (e.g. "L" means run in direction "l")

j Move down.

k Move up.

l Move right.

y Move diagonally up and left.

u Move diagonally up and right.

b Move diagonally down and left.

n Move diagonally down and right.

t Throw an object. This is a prefix command. When followed with a direction it throws an object in the specified direction. (e.g. type "th" to throw something to the left.)

f Find prefix. When followed by a direction it means to continue moving in the specified direction until you pass something interesting or run into a wall. You should experiment with this, since it is a very useful command, but very difficult to describe.

z Zap prefix. Point a staff or wand in a given direction and fire it. Even non-directional staves must be pointed in some direction to be used.

^ Identify trap command. If a trap is on your map and you can't remember what type it is, you can get rogue to remind you by getting next to it and

typing "^" followed by the direction that would move you on top of it.

s Search for traps and secret doors. Examine each space immediately adjacent to you for the existence of a trap or secret door. There is a large chance that even if there is something there, you won't find it, so you might have to search a while before you find something.

> Climb down a staircase to the next level. Not surprisingly, this can only be done if you are standing on staircase.

< Climb up a staircase to the level above. This can't be done without the Amulet of Yendor in your posession.

. Rest. This is the "do nothing" command. This is good for waiting and healing.

i Inventory. List what you are carrying in your pack.

I Selective inventory. Tells you what a single item in your pack is.

q Quaff one of the potions you are carrying.

r Read one of the scrolls in your pack.

e Eat food from your pack.

w Wield a weapon. Take a weapon out of your pack and carry it for use in combat, replacing the one you are currently using (if any).

W Wear armor. You can only wear one suit of armor at a time. This takes extra time.

T Take armor off. You can't remove armor that is cursed. This takes extra time.

P Put on a ring. You can wear only two rings at a time (one on each hand). If you aren't wearing any rings, this command will ask you which hand you want to wear it on, otherwise, it will place it on the unused hand. The program assumes that you wield your sword in your right hand.

R Remove a ring. If you are only wearing one ring, this command takes it off. If you are wearing two, it will ask you which one you wish to remove,

d Drop an object. Take something out of your pack and leave it lying on the floor. Only one object can occupy each space. You cannot drop a cursed object at all if you are wielding or wearing it.

c Call an object something. If you have a type of object in your pack which you wish to remember something about, you can use the call command to give a name to that type of object. This is usually used when you figure out what a potion, scroll, ring, or staff is after you pick it up. (See the "askme" option below.)

D Print out which things you've discovered something about. This command will ask you what type of thing you are interested in. If you type the character for a given type of object (*e.g.* "!" for potion) it will tell you which kinds of that type of object you've discovered (*i.e.*, figured out what they are). This command works for potions, scrolls, rings, and staves and wands.

o Examine and set options. This command is further explained in the section on options.

^L Redraws the screen. Useful if spurious messages or transmission errors have messed up the display.

^R Repeat last message. Useful when a message disappears before you can read it. This only repeats the last message that was not a mistyped command so that you don't loose anything by accidentally typing the wrong

character instead of ^R.

<ESCAPE>
Cancel a command, prefix, or count.

! Escape to a shell for some commands.

Q Quit. Leave the game.

S Save the current game in a file. It will ask you whether you wish to use the default save file. *Caveat*: Rogue won't let you start up a copy of a saved game, and it removes the save file as soon as you start up a restored game. This is to prevent people from saving a game just before a dangerous position and then restarting it if they die. To restore a saved game, give the file name as an argument to rogue. As in

% rogue *save_file*

To restart from the default save file (see below), run

% rogue −r

v Prints the program version number.

5. Rooms

Rooms in the dungeons are either lit or dark. If you walk into a lit room, the entire room will be drawn on the screen as soon as you enter. If you walk into a dark room, it will only be displayed as you explore it. Upon leaving a room, all objects inside the room which might move or be removed are erased from the screen. In the darkness you can only see one space in all directions around you. A corridor is always dark.

6. Fighting

If you see a monster and you wish to fight it, just attempt to run into it. Many times a monster you find will mind its own business unless you attack it. It is often the case that discretion is the better part of valor.

7. Objects you can find

When you find something in the dungeon, it is common to want to pick the object up. This is accomplished in rogue by walking over the object. If you are carrying too many things, the program will tell you and it won't pick up the object, otherwise it will add it to your pack and tell you what you just picked up.

Many of the commands that operate on objects must prompt you to find out which object you want to use. If you change your mind and don't want to do that command after all, just type an <ESCAPE> and the command will be aborted.

Some objects, like armor and weapons, are easily differentiated. Others, like scrolls and potions, are given labels which vary according to type. During a game, any two of the same kind of object with the same label are the same type. However, the labels will vary from game to game.

When you use one of these labeled objects, if its effect is obvious, rogue will remember what it is for you. If it's effect isn't extremely obvious, you can use the "call" command (see above) or the "askme" option (see below) to scribble down something about it so you will recognize it later.

7.1. Weapons

Some weapons, like arrows, come in bunches, but most come one at a time. In order to use a weapon, you must wield it. To fire an arrow out of a bow, you must first wield the bow, then throw the arrow. You can only wield one weapon at a time, but you can't change weapons if the one you are currently wielding is

cursed.

7.2. Armor

There are various sorts of armor lying around in the dungeon. Some of it is enchanted, some is cursed, and some is just normal. Different armor types have different armor classes. The lower the armor class, the more protection the armor affords against the blows of monsters. Here is a list of the various armor types and their normal armor class:

Type	Class
None	10
Leather armor	8
Studded leather / Ring mail	7
Scale mail	6
Chain mail	5
Banded mail / Splint mail	4
Plate mail	3

If a piece of armor is enchanted, its armor class will be lower than normal. If a suit of armor is cursed, its armor class will be higher, and you will not be able to remove it. However, not all armor with a class that is higher than normal is cursed.

7.3. Scrolls

Scrolls come with titles in an unknown tongue. After you read a scroll, it disappears from your pack.

7.4. Potions

Potions are labeled by the color of the liquid inside the flask. They disappear after being quaffed.

7.5. Staves and Wands

Staves and wands do the same kinds of things. Staves are identified by a type of wood; wands by a type of metal or bone. They are generally things you want to do to something over a long distance, so you must point them at what you wish to affect to use them. Some staves are not affected by the direction they are pointed, though. Staves come with multiple magic charges, the number being random, and when they are used up, the staff is just a piece of wood or metal.

7.6. Rings

Rings are very useful items, since they are relatively permanent magic, unlike the usually fleeting effects of potions, scrolls, and staves. Of course, the bad rings are also more powerful. Most rings also cause you to use up food more rapidly, the rate varying with the type of ring. Rings are differentiated by their stone settings.

8. Options

Due to variations in personal tastes and conceptions of the way rogue should do things, there are a set of options you can set that cause rogue to behave in various different ways.

8.1. Setting the options

There are two ways to set the options. The first is with the "o" command of rogue; the second is with the "ROGUEOPTS" environment variable[2].

8.1.1. Using the 'o' command

When you type "o" in rogue, it clears the screen and displays the current settings for all the options. It then places the cursor by the value of the first option and waits for you to type. You can type a <RETURN> which means to go to the next option, a "–" which means to go to the previous option, an <ESCAPE> which means to return to the game, or you can give the option a value. For boolean options this merely involves typing "t" for true or "f" for false. For string options, type the new value followed by a <RETURN>.

8.1.2. Using the ROGUEOPTS variable

The ROGUEOPTS variable is a string containing a comma separated list of initial values for the various options. Boolean variables can be turned on by listing their name or turned off by putting a "no" in front of the name. Thus to set up an environment variable so that **jump** is on, **terse** is off, and the **name** is set to "Blue Meanie", use the command

% setenv ROGUEOPTS "jump,noterse,name=Blue Meanie"[3]

8.2. Option list

Here is a list of the options and an explanation of what each one is for. The default value for each is enclosed in square brackets. For character string options, input over fifty characters will be ignored.

terse [*noterse*]
: Useful for those who are tired of the sometimes lengthy messages of rogue. This is a useful option for playing on slow terminals, so this option defaults to **terse** if your are on a slow (1200 baud or under) terminal.

jump [*nojump*]
: If this option is set, running moves will not be displayed until you reach the end of the move. This saves considerable cpu and display time. This option defaults to **jump** if you are using a slow terminal.

step [*nostep*]
: When **step** is set, lists of things, like inventories or "*" responses to "Which item do you wish to . . .? " questions, are displayed one item at a time on the top of the screen, rather than clearing the screen, displaying the list, then re-displaying the dungeon level.

flush [*noflush*]
: All typeahead is thrown away after each round of battle. This is useful for those who type far ahead and then watch in dismay as a Kobold kills them.

askme [*noaskme*]
: Upon reading a scroll or quaffing a potion which does not automatically identify itself upon use, rogue will ask you what to name it so you can recognize it if you encounter it again.

---

[2] On Version 6 systems, there is no equivalent of the ROGUEOPTS feature.

[3] For those of you who use the bourne shell, the commands would be

$ ROGUEOPTS="jump,noterse,name=Blue Meanie"
$ export ROGUEOPTS

passgo [*nopassgo*]
Follow turnings in passageways. If you run in a passage and you run into stone or a wall, rogue will see if it can turn to the right or left. If it can only turn one way, it will turn that way. If it can turn either or neither, it will stop. This is followed strictly, which can sometimes lead to slightly confusing occurrences (which is why it defaults to being off). The "f" prefix still works.

name [account name]
This is the name of your character. It is used if you get on the top ten scorer's list.

fruit [*slime-mold*]
This should hold the name of a fruit that you enjoy eating. It is basically a whimsey that the program uses in a couple of places.

file [*~/rogue.save*]
The default file name for saving the game. If your phone is hung up by accident, rogue will automatically save the game in this file. The file name may contain the special character "~" which expands to be your home directory.

9. Scoring

Rogue usually maintains a list of the top ten scoring people on your machine. Each account on the machine can post only one non-winning score on this list. If you score higher than someone else on this list, or better your previous score on the list, you will be inserted in the proper place under your current name.

If you quit the game, you get out with all of your gold intact. If, however, you get killed in the Dungeons of Doom, your body is forwarded to your next-of-kin, along with 90% of your gold; ten percent of your gold is kept by the Dungeons' wizard as a fee. This should make you consider whether you want to take one last hit at that monster and possibly live, or quit and thus stop with whatever you have. If you quit, you do get all your gold, but if you swing and live, you might find more.

If you just want to see what the current top ten list is, you can type

```
% rogue -s
```

10. Acknowledgements

Rogue was originally conceived of by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added jillions of new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance, and also the teeming multitudes who graciously ignored work, school, and social life to play rogue and send us bugs, complaints, suggestions, and just plain flames. And also Mom.

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

ABSTRACT

Vi (visual) is a display oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file which you are editing. Changes which you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like **d** for delete and **c** for change, are combined with the motion commands to form operations such as delete word or change paragraph, in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and to use.

Vi will work on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, the editor will function quite well on dumb terminals over slow phone lines. The editor makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and "glass tty's" using a one line editing window; thus *vi's* command set is available on all terminals. The full command set of the more traditional, line oriented editor *ex* is available within *vi;* it is quite simple to switch between the two modes of editing.

September 16, 1980

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

1. Getting started

This document provides a quick introduction to *vi*. (Pronounced *vee-eye*.) You should be running *vi* on a file you are familiar with while you are reading this. The first part of this document (sections 1 through 5) describes the basics of using *vi*. Some topics of special interest are presented in section 6, and some nitty-gritty details of how the editor functions are saved for section 7 to avoid cluttering the presentation here.

There is also a short appendix here, which gives for each character the special meanings which this character has in *vi*. Attached to this document should be a quick reference card. This card summarizes the commands of *vi* in a very compact format. You should have the card handy while you are learning *vi*.

1.1. Specifying terminal type

Before you can start *vi* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent

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

t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system the type of your terminal:

% **setenv TERM** 2621

This command works with the shell *csh* on both version 6 and 7 systems. If you are using the standard version 7 shell then you should give the commands

$ TERM=2621
$ export TERM

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use csh) would be

**setenv TERM `tset** – –d mime`

or for your *.profile* file (if you use sh)

**TERM=`tset** – –d mime`

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

1.2. Editing a file

After telling the system which kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file, giving the command

% **vi** *name*

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen. If something else happens refer to the footnote.‡

1.3. The editor's copy: the buffer

The editor does not directly modify the file which you are editing. Rather, the editor makes a copy of this file, in a place called the *buffer,* and remembers the file's name. You do not affect the contents of the file unless and until you write the changes you make back into the original file.

---

‡ If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

Another thing which can go wrong is that you typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again this time spelling the file name correctly.

If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the **:q** command again followed by a carriage return.

1.4. Notational conventions

In our examples, input which must be typed as is will be presented in **bold face**. Text which should be replaced with appropriate input will be given in *italics*. We will represent special characters in SMALL CAPITALS.

1.5. Arrow keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labelled with arrows on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use will leave the cursor positioned incorrectly.)

1.6. Special characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in a quiescent state.‡ Partially formed commands are cancelled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '/' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.* From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."**

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then 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. When this is happening you can stop the editor by sending an interrupt.

1.7. Getting out of the editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command which ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish

---

* As we will see later, *h* moves back to the left (like control-h which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

‡ On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.

* Backspacing over the '/' will also cancel the search.

** On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

† All commands which read from the last display line can also be terminated with a ESC as well as an CR.

only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. Moving around in the file

2.1. Scrolling and paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labelled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.‡

As you know now if you tried hitting **^D**, this command scrolls down in the file. The **D** thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is **^U**. Many dumb terminals can't scroll up at all, in which case hitting **^U** clears the screen and refreshes it with a line which is farther back in the file at the top.

If you want to see more of the file below where you are, you can hit **^E** to expose one more line at the bottom of the screen, leaving the cursor where it is. ‡‡ The command **^Y** (which is hopelessly non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen.

There are other ways to move around in the file; the keys **^F** and **^B** ‡ move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than **^D** and **^U** if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **^F** to move forward a page will leave you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

2.2. Searching, goto, and previous context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character **/** followed by a string of characters terminated by CR. The editor will position the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** will search backwards from where you are, and is otherwise like **/**.†

If the search string you give the editor is not present in the file the editor will print a diagnostic on the last line of the screen, and the cursor will be returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a **$**. Thus **/↑search**CR will search for the word 'search' at the beginning of a line, and **/last$**CR searches for the word 'last' at the end of a line.*

---

‡ If you don't have a '^' key on your terminal then there is probably a key labelled '↑'; in any case these characters are one and the same.

‡‡ Version 3 only.

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

† These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

*Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex*(1) and *ed*(1). If you don't wish to learn about this yet, you can disable this more general facility by doing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, then it moves to the end of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

You can find out the state of the file you are editing by typing a **^G**. The editor will show you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer which you are. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you were.

You can also get back to a previous position by using the command `` (two back quotes). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with **/** or **?** and then a `` to get back to where you were. If you accidentally hit **n** or any command which moves you far away from a context of interest, you can quickly get back by hitting ``.

2.3. Moving around on the screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals using v2 editors, they won't.) If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

Hit the **+** key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The **−** key is like **+** but goes the other way.

These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen will scroll down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the **+** key.

Vi also has commands to take you to the top, middle and bottom of the screen. **H** will take you to the top (home) line on the screen. Try preceding it with a number as in **3H**. This will take you to the third line on the screen. Many *vi* commands take preceding numbers and do interesting things with them. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last line on the screen. **L** also takes counts, thus **5L** will take you to the fifth line from the bottom.

2.4. Moving within a line

Now try picking a word on some line on the screen, not the first word on the line. move the cursor using RETURN and **−** to be on the line where the word is. Try hitting the **w** key. This will advance the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Also try the **e** key which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BS (backspace or **^H**) key which moves left one character. The key **h** works as **^H** does and is useful if you don't have a BS key. (Also, as noted just above, **l** will move to the right.)

If the line had punctuation in it you may have noticed that that the **w** and **b** keys stopped at each group of punctuation. You can also go back and forwards words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w**.

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom (v3)
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
−	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of **/** or **?** pattern
w	word after this word

2.6. View ‡

If you want to use the editor to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This will set the *readonly* option which will prevent you from accidently overwriting the file.

3. Making simple changes

3.1. Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you hit ESC.

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then 'sESC' to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

Try inserting and appending a few times to make sure you understand how this works; **i** placing text to the left of the cursor, **a** to the right.

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, text you type up to an ESC

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

is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay which would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, **^X**, or **^U**) to erase the input you have typed on the current line.† The character **^W** will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you hit ESC; if you want to get rid of them immediately, hit an ESC and then **a** again.

Notice also that you can't erase characters which you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

3.2. Making small corrections

You can make small corrections in existing text quite easily. Find a single character which is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key or **^H** or even just **h**) or SPACE (using the space bar) until the cursor is on the character which is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you **x** out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by giving the command **r***c*, where *c* is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.3. More corrections: operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** a few times. Notice that this repeats the effect of the **dw**. The command **.** repeats the last command which made a change. You can remember it by analogy with an ellipsis '**...**'.

† In fact, the character **^H** (backspace) always works to erase the last input character here, regardless of what your erase character is.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Another very useful operator is **c** or change. The command **cw** thus changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word which you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

3.4. Operating on lines

It is often the case that you want to operate on lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.†

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.* Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change you make affects text which you cannot see.

3.5. Undoing

Now suppose that the last change which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change which you made. Try this a few times, and give it twice in a row to notice that an **u** also undoes a **u.**

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see the section on recovering lost text below.

3.6. Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text

† The command S is a convenient synonym for for **cc**, by analogy with **s**. Think of S as a substitute on lines, while **s** is a substitute on characters.

* One subtle point here involves using the / search after a **d**. This will normally delete characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. Moving about; rearranging and duplicating text

4.1. Low level character motions

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a ;, which finds the next instance of the same character. By using the **f** command and then a sequence of ;'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also a **F** command, which is like **f**, but searches backward. The ; command repeats **F** also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for to, i.e. delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, an ↑ moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** will append new text at the end of the current line.

Your file may have tab (**^I**) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.* When the cursor is at a tab, it sits on the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a **^V** before the control character. The **^V** quotes the following character, causing it to be inserted directly into the file.

4.2. Higher level text objects

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations **(** and **)** move to the beginning of the previous and next sentences respectively. Thus the command **d)** will delete the rest of the current sentence; likewise **d(** will delete the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence is defined to end at a '.', '!' or '?' which is followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces or end of line.

The operations { and } move over paragraphs and the operations **[[** and **]]** move over sections.†

---

* This is settable by a command of the form **:se ts=***x*CR, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

† The **[[** and **]]** operations require the operation character to be doubled because they can move the cursor far

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the *−ms* and *−mm* macro packages, i.e. the '.IP', '.LP', '.PP' and '.QP', '.P' and '.LI' macros.‡ Each paragraph boundary is also a sentence boundary. 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, normally '.NH', '.SH', '.H' and '.HU', and each line with a formfeed **^L** in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

4.3. Rearranging and duplicating text

The editor has a single unnamed buffer where the last deleted or changed away text is saved, and a set of named buffers **a−z** which you can use to save copies of text and to move text around in your file and between files.

The operator **y** yanks a copy of the object which follows into the unnamed buffer. If preceded by a buffer name, "*x***y**, where *x* here is replaced by a letter **a−z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P; p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text which you yank forms a part of a line, or is an object such as a sentence which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like a **o** or **O** command.

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in "**a5dd** deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of the these lines and do a "**ap** or "**aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form **:e** *name*CR where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you should use an unnamed buffer.

---

from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally.

‡ You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

4.4. Summary.

↑	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5. High level commands

5.1. Writing, quitting, editing new files

So far we have seen how to enter *vi* and to write out our file using either **ZZ** or **:w**CR. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

If you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, then you can give the command **:q!**CR to quit from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command **:e!**CR. These commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command **:e** *name*CR. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command **:w**CR to save your work and then the **:e** *name*CR command again, or carefully give the command **:e!** *name*CR, which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT, and use **:n** instead of **:e**.

5.2. Escaping to a shell

You can get to a shell to execute a single command by giving a *vi* command of the form **:!***cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command **:sh**CR. This will give you a new shell, and when you finish with the shell, ending it by typing a **^D**, the editor will clear the screen and continue.

On systems which support it, **^Z** will suspend the editor and return to the (top level) shell. When the editor is resumed, the screen will be redrawn.

5.3. Marking and returning

The command `` returned to the previous place after a motion of the cursor by a command such as **/**, **?** or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and hit **`a**. The cursor will return to the place which you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case you can use the form *'x* rather than *`x*. Used without an operator, *'x* will move to the first non-white character of the marked line; similarly *''* moves to the first non-white character of the line containing the previous context mark ``.

5.4. Adjusting the screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a **^L**, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to cause the editor to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a **.** if you want it at the center, or a **−** if you want it at the bottom. (**z.**, **z-**, and **z+** are not available on all v2 editors.)

6. Special topics

6.1. Editing on slow terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command :se **slow**CR. If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by **:se noslow**CR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command **:se redraw**CR. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command
:se noredrawCR.

The editor also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window sizes as argument to the commands which cause large screen motions:

: / ? [[ ]] ` '

Thus if you are searching for a particular instance of a common string in a file you can precede

the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string which it locates.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a **z** command, after the **z** and before the following RETURN, **.** or **−**. Thus the command **z5.** redraws the screen with the current line in the center of a five line window.†

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a DEL or RUB as usual. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a **^L**; or move or search again, ignoring the current state of the display.

See section 7.8 on *open* mode for another way to use the *vi* command set on slow terminals.

6.2. Options, set, and editor startup files

The editor has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

Name	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before **:n**, **:ta**, ^↑, !
ignorecase	noic	Ignore case in searching
lisp	nolisp	({) } commands deal with S-expressions
list	nolist	Tabs print as ^I; end of lines marked with $
magic	nomagic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names which start new sections
shiftwidth	sw=8	Shift distance for <, > and input **^D** and **^T**
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

set *opt*= *val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment, or given while you are running *vi* by preceding them with a : and following them with a CR.

You can get a list of all options which you have changed by the command **:set**CR, or the value of a single option by the command **:set** *opt***?**CR. A list of all possible options and their values is generated by **:set all**CR. Set can be abbreviated **se**. Multiple options can be placed on one line, e.g. **:se ai aw nu**CR.

Options set by the **set** command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A

† Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

† All commands which start with : are *ex* commands.

typical list includes a **set** command, and possibly a few **map** commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

set ai aw terse|**map** @ dd|**map** # x

which sets the options *autoindent*, *autowrite*, *terse*, (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command, which only works in version 3.) This string should be placed in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

setenv EXINIT '**set** ai aw terse|**map** @ dd|**map** # x'

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

EXINIT='**set** ai aw terse|**map** @ dd|**map** # x'
export EXINIT

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

set ai aw terse|**map** @ dd|**map** # x

Of course, the particulars of the line would depend on which options you wanted to set.

6.3. Recovering lost lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1−9. You can get the *n*'th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

"1pu.u.u.

will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the **u** commands here to gather up all this text in the buffer, or stop after any **.** command to keep just the then recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

6.4. Recovering lost files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login giving you the name of the file which has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

% **vi −r** *name*

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.†

† In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be among the last few which you changed. You can either choose to discard the changes which you made (if they are easy to remake) or to replace the few lost lines by hand.

You can get a listing of the files which are saved for you by giving the command:

% **vi −r**

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation "*vi -r*" will not always list all saved files, but they can be recovered even if they are not listed.

6.5. Continuous text input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command **:se wm=10**CR. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.*

If the editor breaks an input line and you wish to put it back together you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined as in **3J** to join 3 lines. The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

6.6. Features for editing programs

The editor has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has a *autoindent* facility for helping you generate correctly indented programs.

To enable this facility you can give the command **:se ai**CR. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D** you back up one position, normally to an 8 column boundary. This amount is settable; the editor has an option called *shiftwidth* which you can set to change this value. Try giving the command **:se sw=4**CR and then experimenting with autoindent again.

For shifting lines in the program left and right, there are operators < and >. These shift the lines you specify right or left by one *shiftwidth*. Try << and >> which shift one line left or right, and **<L** and **>L** shifting the rest of the display left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit **%**. This will show you the matching parenthesis. This works also for braces { and }, and brackets [and].

If you are editing C programs, you can use the **[[** and **]]** keys to advance or retreat to a line starting with a {, i.e. a function declaration at a time. When **]]** is used with an operator it stops after a line which starts with }; this is sometimes useful with **y]]**.

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

6.7. Filtering portions of the buffer

You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command **!}sort**CR. This says to sort the next paragraph of material, and the blank line ends a paragraph.

6.8. Commands for editing LISP†

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the **(** and **)** commands to move backward and forward over s-expressions. The **{** and **}** commands are like **(** and **)** but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying 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.

There is another option which is useful for typing in LISP, the *showmatch* option. Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the **=** operator. Try the command **=%** at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP,, the **[[** and **]]** advance and retreat to lines beginning with a **(**, and are useful for dealing with entire function definitions.

6.9. Macros‡

Vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a) Ones where you put the macro body in a buffer register, say *x*. You can then type **@x** to invoke the macro. The **@** may be followed by another **@** to repeat the last macro.

b) You can use the *map* command from *vi* (typically in your *EXINIT*) with a command of the form:

:map *lhs rhs*CR

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a **^V**. (It may be necessary to double the **^V** if the map command is given inside *vi*, rather than in *ex.)* Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the **q** key write and exit the editor, you can give the command

:map q :wq**^V^V**CR CR

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A **^V**'s is needed because without it the CR would end the : command, rather than

---

† The LISP features are not available on some v2 editors due to memory constraints.

‡ The macro feature is available only in version 3 editors.

becoming part of the *map* definition. There are two **^V**'s because from within *vi*, two **^V**'s must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

unmap lhs

If the *lhs* of a macro is "#0" through "#9", this maps the particular function key instead of the 2 character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" will mean function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way:

:map **^V^V^I** #

to use tab, for example. (This won't affect the *map* command, which still uses #, but just the invocation from visual mode.

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing a '!' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for **^T** to be the same as 4 spaces in input mode, you can type:

:map **^T ^V**bbbb

where **b** is a blank. The **^V** is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

7. Word Abbreviations ‡‡

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

:ab eecs Electrical Engineering and Computer Sciences

causes the word 'eecs' to always be changed into the phrase 'Electrical Engineering and Computer Sciences'. Word abbreviation is different from macros in that only whole words are affected. If 'eecs' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

7.1. Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

8. Nitty-gritty details

8.1. Line representation in the display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80**| on a line which is more than 80 columns long.†

The editor only puts 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, placing only an @ on the line as a

‡‡ Version 3 only.

† You can make long lines very easily by using **J** to join together short lines.

place holder. When you delete lines on a dumb terminal, the editor will often just clear the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the **^R** command.

If you wish, you can have the editor place line numbers before each line on the display. Give the command **:se nu**CR to enable this, and the command **:se nonu**CR to turn it off. You can have tabs represented as **^I** and the ends of lines indicated with '$' by giving the command **:se list**CR; **:se nolist**CR turns this off.

Finally, lines consisting of only the character '~' are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of file.

8.2. Counts

Most *vi* commands will use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

new window size	**: / ? [[ ]] ` '**
scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On terminals which run at speeds greater than 1200 baud the editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dialup lines are in this group) the editor uses 8 lines as the default window size. At 1200 baud the default is 16 lines.

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a **−** or similar command or off the bottom with a command such as RETURN or **^D**. The window will revert to the last specified size the next time it is cleared and refilled.†

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+−−−−**ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as **^R**), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

8.3. More file manipulation commands

The following table lists the file manipulation commands which you can use when you are in *vi*. All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with one

† But not by a **^L** which just redraws the screen as it is.

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *name*	edit file *name*
:e!	reedit, discarding changes
:e + *name*	edit, starting at end
:e +*n*	edit, starting at line *n*
:e #	edit alternate 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!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

with a **:w** and start editing a new file by giving a **:e** command, or set *autowrite* and use **:n** <file>.

If you make changes to the editor's copy of a file, but do not wish to write them back, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The **:e** command can be given a **+** argument to start at the end of the file, or a **+***n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usefully a scan like **+/***pat* or **+?***pat*. In forming new names to the **e** command, you can use the character **%** which is replaced by the current file name, or the character **#** which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e #** command to redo the previous **:e**.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using **^G**, and giving these numbers after the **:** and before the **w**, separated by **,**'s. You can also mark these lines with **m** and then use an address of the form **'***x***,'***y* on the **w** command here.

You can read another file into the buffer after the current line by using the **:r** command. You can similarly read in the output from a command, just use **!***cmd* instead of a file name.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to respecify the list of files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it on the initial *vi* command.

If you are editing large programs, you will find the **:ta** command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, to quickly find a function whose name you give. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

When you are searching for strings in the file with **/** and **?**, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c** or **y**, then you may well wish to affect lines up to the line before the line containing the pattern.

You can give a search of the form */pat/−n* to refer to the *n*'th line before the next line containing *pat*, or you can use **+** instead of **−** to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will affect characters up to the match place, rather than whole lines; thus use "+0" to affect to the line which matches.

You can have the editor ignore the case of words in the searches it does by giving the command **:se ic**CR. The command **:se noic**CR turns this off.

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

set nomagic

in your EXINIT. In this case, only the characters ↑ and **$** are special in patterns. The character \ is also then special (as it is most everywhere in the system), and may be used to get at the an extended pattern matching facility. It is also necessary to use a \ before a / in a forward scan or a **?** in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

↑	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	matches any character
\<	matches the beginning of a word
\>	matches the end of a word
[*str*]	matches any single character in *str*
[↑*str*]	matches any single character not in *str*
[*x*−*y*]	matches any character between *x* and *y*
*	matches any number of the preceding pattern

If you use **nomagic** mode, then the **. [** and ***** primitives are given with a preceding \.

8.5. More about input mode

There are a number of characters which you can use to make corrections during input mode. These are summarized in the following table.

^H	deletes the last input character
^W	deletes the last input word, defined as by **b**
erase	your erase character, same as **^H**
kill	your kill character, deletes the input on this line
\	escapes a following **^H** and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*
0^D	kills all the *autoindent*
↑^D	same as **0^D**, but restores indent next line
^V	quotes the next non-printing character into the file

The most usual way of making corrections to input is by typing **^H** to correct a single character, or by typing one or more **^W**'s to back over incorrect words. If you use # as your erase character in the normal system, it will work like **^H**.

Your system kill character, normally @, **^X** or **^U**, will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which you did not insert with this insertion command. To make corrections on the previous line after a new line has been started you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue.

The command **A** which appends at the end of the current line is often useful for continuing.

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a **^V**. The **^V** echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.*

If you are using *autoindent* you can backtab over the indent which it supplies by typing a **^D**. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent.*

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then **^D**. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a **^D** if you wish to kill all the indent and not have it come back on the next line.

8.6. Upper case only terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters which you normally type are converted to lower case, and you can type upper case letters by preceding them with a \. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\↑ \) \! \'. These characters are represented on the display in the same way they are typed.‡ ‡

8.7. Vi and ex

Vi is actually one mode of editing within the editor *ex.* When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command **Q**. All of the : commands which were introduced above are available in *ex.* Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the : and follow them with a CR.

In rare instances, an internal error may occur in *vi.* In this case you will get a diagnostic and be left in the command mode of *ex.* You can then save your work and quit if you wish by giving a command **x** after the : which *ex* prompts you with, or you can reenter *vi* by giving *ex* a *vi* command.

There are a number of things which you can do more easily in *ex* than in *vi.* Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out a lot more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

8.8. Open mode: vi on hardcopy terminals and "glass tty's" ‡

If you are on a hardcopy terminal or a terminal which does not have a cursor which can move off the bottom line, you can still use the command set of *vi,* but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex,* which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is

---

* This is not quite true. The implementation of the editor does not allow the NULL (**^@**) character to appear in files. Also the LF (linefeed or **^J**) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type **^S** or **^Q**, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

‡ The \ character you give will not echo until you type another key.

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

displayed.

In *open* mode the editor uses a single line window into the file, and moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open:* **z** and **^R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the **^R** command will retype the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters which are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

Acknowledgements

Bruce Englar encouraged the early development of this display editor. Peter Kessler helped bring sanity to version 2's command layout. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Appendix: character functions

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 lower case characters.

For each character we tell a 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. Section numbers in parentheses indicate where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

^@	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 (7.5f).
^A	Unused.
^B	Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^C	Unused.
^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 **^D** and **^U** commands (2.1, 7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 7.5); this white space cannot be backspaced over.
^E	Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)
^F	Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^G	Equivalent to **:f**CR, printing the current file, 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.
^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 (3.1, 7.5).
^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 (4.1, 6.6).
^J (LF)	Same as **down arrow** (see **j**).
^K	Unused.
^L	The ASCII formfeed character, this causes the screen to be cleared and redrawn. 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 (5.4, 7.2f).
^M (CR)	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 (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).
^N	Same as **down arrow** (see **j**).
^O	Unused.

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

^Q Not a command character. In input mode, **^Q** quotes the next character, the same as **^V**, except that some teletype drivers will eat the **^Q** so that the editor never sees it.

^R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).

^S Unused. Some teletype drivers use **^S** to suspend output until **^Q**is

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

^U Scrolls the screen up, inverting **^D** which scrolls down. Counts work as they do for **^D**, and the previous scroll amount is common to both. On a dumb terminal, **^U** will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).

^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 (4.2, 7.5).

^W Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see **^H**) (7.5).

^X Unused.

^Y Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to **^U** which scrolls up a bunch.) (Version 3 only.)

^Z If supported by the Unix system, stops the editor, exiting to the top level shell. Same as **:stop**CR. Otherwise, unused.

^[(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. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).

^ Unused.

^] Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).

^↑ Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (7.3). (You have to do a **:w** before **^↑** will work in this case. If you do not wish to write the file you should do **:e! #**CR instead.)

^_ Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

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

! An operator, which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. 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!}***fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%***grind*CR,* given at the

*Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.

beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). To read a file or the output of a command into the buffer use **:r** (7.3). To simply execute a command use **:!** (7.3).

" Precedes a named buffer specification. There are named buffers **1−9** used for saving deleted text and named buffers **a−z** into which you can place text (4.3, 6.3)

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys (6.9). In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.

$** Moves to the end of the current line. If you **:se list**CR, then the end of each line will be shown by printing a **$ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.

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

& A synonym for **:&**CR, by analogy with the *ex* **&** command.

′ When followed by a ′ returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a−z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. (2.2, 5.3). When used with an operator such as **d**, the operation takes place over complete lines; if you use `, the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **. !** or **?** which is followed by either the end of a line or by two spaces. Any number of closing **)]** " and ′ characters may appear after the **. !** or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see **{** and **[[** below). A count advances that many sentences (4.2, 6.8).

) Advances to the beginning of a sentence. A count repeats the effect. See **(** above for the definition of a sentence (4.2, 6.8).

* Unused.

+ Same as CR 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 (2.3).

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit **.** to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words (3.3, 6.3, 7.2, 7.4).

/ 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 on the bottom line; an returns to command state without ever searching. The search begins when you hit CR to terminate 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, returning the cursor 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 you can force whole lines to be affected. To do this give a pattern with a closing a closing / and then an offset $+n$ or $-n$.

To include the character / in the search string, you must 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, see section 7.4; unless you set **nomagic** in your *.exrc* file you will have to preceed the characters **. [*** and ~ in the search pattern with a \ to get them to work as you would naively expect (1.5, 2,2, 6.1, 7.2, 7.4).

0 Moves to the first character on the current line. Also used, in forming numbers, after an initial **1−9**.

1−9 Used to form numeric arguments to commands (2.3, 7.2).

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit : accidentally (see primarily 6.2 and 7.3).

; Repeats the last single character find which used **f F t** or **T**. A count iterates the basic scan (4.1).

< An operator which 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 (6.6, 7.2).

= Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).

> An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object (6.6, 7.2).

? Scans backwards, the opposite of /. See the / description above for details on scanning (2.2, 6.1, 7.4).

@ A macro character (6.9). If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 7.5).

A Appends at the end of line, a synonym for **$a** (7.2).

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 (2.4).

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 (4.1).

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 (7.2).

H **Home arrow**. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected (2.3, 3.2).

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 on line is **)**. A count causes that many lines to be joined rather than the default two (6.5, 7.1f).

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L** (2.3).

M Moves the cursor to the middle line on the screen, at the first non-white position on the line (2.3).

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 up to an 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 (3.1).

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted 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 (6.3).

Q Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the **:** commands; the editor supplies the **:** as a prompt (7.7).

R Replaces characters on the screen with characters you type (overlay fashion). Terminates with an 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 repeats the effect. Most useful with operators such as **d** (4.1).

U Restores the current line to its state before you started changing it (3.5).

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 repeats the effect (2.4).

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**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (7.4).

ZZ Exits the editor. (Same as **:x**CR.) 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 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. (4.2, 6.1, 6.6, 7.2).

**** Unused.

]] Forward to a section boundary, see **[[** for a definition (4.2, 6.1, 6.6, 7.2).

↑ Moves to the first non-white position on the current line (4.4).

_ Unused.

`** When followed by a ` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a−**z**, returns to the position which was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ′, the operation takes place over complete lines (2.2, 5.3).

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 an ESC (3.1, 7.2).

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (2.4).

c An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences (7.4).

d An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 7.4).

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect (2.4, 3.1).

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find (4.1).

g Unused.

Arrow keys **h**, **j**, **k**, **l**, and **H**.

h — **Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**^H**) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 7.5).

i — Inserts text before the cursor, otherwise like **a** (7.2).

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 ´ (5.3).

n — Repeats the last **/** or **?** scanning commands (2.2).

o — Opens new lines below the current line; otherwise like **O** (3.1).

p — Puts text after/below the cursor; otherwise like **P** (6.3).

q — Unused.

r — Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r** (3.2).

s — Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c** (3.2).

t — Advances the cursor upto 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. You can use **.** to delete more if this doesn't delete enough the first time (4.1).

u — Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers (3.5).

v — Unused.

w — Advances to the beginning of the next word, as defined by **b** (2.4).

x — Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (6.5).

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** (7.4).

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. (5.4)

{ Retreats to the beginning of 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 at each section boundary (see [[above) (4.2, 6.8, 7.6).

| Places the cursor on the character in the column specified by the count (7.1, 7.2).

} Advances to the beginning of the next paragraph. See { for the definition of paragraph (4.2, 6.8, 7.6).

~ Unused.

^? (DEL) Interrupts the editor, returning it to command accepting state (1.5, 7.5)

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

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

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

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

August 31, 1980

†UNIX is a trademark of Bell Laboratories.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

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

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

program — A set of instructions given to the computer, describing the sequence of steps which the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.

UNIX — UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.

edit — *edit* is the name of the UNIX text editor which you will be learning to use, a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor named *ex.*

file — Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file it is kept until you instruct the system to remove it. You may create a file during one UNIX session, log out, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, and another might contain a very long document or program. The only way to save information from one session to the next is to write it to a file, storing it for later use.

filename — Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.

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

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

Session 1: Creating a File of Text

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

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

:login:

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

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

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

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

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

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

Login incorrect.
:login:

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

Asking for *edit*

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

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

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

"text" No such file or directory

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

The "Command not found" message

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

% **editor**
editor: Command not found.
%

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

A summary

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

:login: **susan**
Password:

... A Message of General Interest ...

% **edit text**
"text" No such file or directory
:

Entering text

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

% **edit text**
:**append**

Messages from *edit*

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

"append" or "a", you will receive this message:

> :**add**
> add: Not an editor command
> :

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

Text input mode

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

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

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

> .

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

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

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

Writing text to disk

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

: **write**

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

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

No current filename

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

: **write text**

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

Logging off

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

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

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

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

Session 2

Login with UNIX as in the first session:

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

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

> % **edit text**
> "text" 3 lines, 90 characters
> :

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

Adding more text to the file

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

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

Interrupt

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

> Interrupt
> :

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

Making corrections

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

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

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

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

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

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

This is yukky tex@This is great text.

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

Listing what's in the buffer

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

:**1,$p**

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

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

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

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

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

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

Finding things in the buffer

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

:**/thiss/**

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

And thiss is some more text.

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

The current line

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

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

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

:**.=**
2

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

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

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

Numbering lines (nu)

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

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

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

Substitute command (s)

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

2s/thiss/this/

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

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

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

Substitute pattern match failed

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

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

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

:**s/thiss/this/**

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

For another illustration of substitution we may choose the line:

Text editing is strange, but nice.

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

Text editing is nice.

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

:**/strange/s/strange, but //**

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

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

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

Text editing is nice.

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

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

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

:**1z**

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

:**z**

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

Saving the modified text

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

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

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

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

and then,

:**q**
% logout

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

Session 3

Bringing text into the buffer (e)

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

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

% **edit text**

or simply

% **edit**

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

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

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

Moving text in the buffer (m)

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

:**2,4m$**

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

:**1,6m20**

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

Let's move some text using the command:

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

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

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

We can restore the original order by typing:

:**4,$m1**

or, combining context searching and the move command:

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

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

Copying lines (copy)

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

:**12,15copy $**

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

Deleting lines (d)

Suppose you want to delete the line

This is text added in Session 2.

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

:**4d**
It doesn't mean much here, but

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

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

:**/added in Session 2./**
This is text added in Session 2.
:**d**
It doesn't mean much here, but

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

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

To delete both lines 2 and 3:

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

you type

:2,3d

which specifies the range of lines from 2 to 3, and the operation on those lines — "d" for delete.

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

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

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

A word or two of caution:

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

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

Undo (u) to the rescue

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

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

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

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

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

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

1. to exit from append mode we type dot (and only a dot) on a line and press carriage return;

2. to refer to the line we are at in the buffer.

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

:**.=**

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

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

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

:**.,$d**

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

Moving around in the buffer (+ and −)

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

−3p

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

+2p

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

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

:**−1,+2copy$**

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

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

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

At end-of-file

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

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

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

Changing lines (c)

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

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

to read

This text was created with the UNIX text editor.

To do so, you can type:

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

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

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

Session 4

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

Making commands global (g)

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

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

> :**g/text/p**

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

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

> :**g/text/s/text/material/g**

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

> :**14s/text/material/g**

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

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

> :**g/text/s/text/material/gp**

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

> :**g/ /d**
> 72 less lines in file after global

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

More about searching and substituting

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

: **/noun/s/noun/nouns/**

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

: **/noun/s//nouns/**

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

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

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

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

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

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

: **s/noun/nouns/**

we could use the command

: **/nouns/&**

or simply

: **//&**

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

Special characters

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

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

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

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

: **s/^/1. /**

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

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

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

> :**s/\$/dollar/**

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

Issuing UNIX commands from the editor

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

> :**!rm junk**
> !
> :

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

Filenames and file manipulation

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

> :**w chapter3**
> "chapter3" 283 lines, 8698 characters

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

The file (f) command

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

> :**f**
> "text" [Modified] line 3 of 4 --75%--

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

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

Reading additional files (r)

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

: **$r bibliography**
"bibliography" 18 lines, 473 characters

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

Writing parts of the buffer

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

: **45,$w ending**

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

Recovering files

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

: **recover chap6**

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

Other recovery techniques

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

: **preserve**

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

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

Further reading and other information

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

Using *ex*

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

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

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

Index

addressing, *see* line numbers
append mode, 4
backslash (\), 18
buffer, 2
command mode, 4
"Command not found" (message), 3
context search, 7-8, 9, 13, 17
control characters ("^" notation), 7
control-D, 5
current filename, 18, 19
current line (.), 8, 14
diagnostic messages, 3-4
disk, 1
documentation, 20
edit (to begin editing session), 2, 6
editing commands:
    append (a), 3, 4, 6
    change (c), 15
    copy (co), 12
    delete (d), 12-13
    edit (e), 11
    file (f), 18
    global (g), 16-17
    move (m), 11-12
    number (nu), 8
    preserve (pre), 19
    print (p), 7
    quit (q), 5, 10
    quit! (q!), 10
    read (r), 19
    recover (rec), 19
    substitute (s), 8-9, 16, 17-18
    undo (u), 13, 16
    write (w), 4-5, 10, 19
    z, 10
    ! (shell escape), 18
    $= , 14
    +, 14
    −, 14
    //, 7-8, 17
    ??, 17
    ., 8, 14
    .=, 8, 14
erasing
    characters (#), 6
    lines (@), 7
ex (text editor), 20
Ex Reference Manual, 20
file, 1
file recovery, 19
filename, 1
Interrupt (message), 6
line numbers, *see also* current line
    dollar sign ($), 7, 14
    dot (.), 8, 14
    relative (+ and −), 14
logging out, 5
login procedure, 2
non-printing characters, 7
program, 1
recovery *see* file recovery
shell, 18
shell escape (!), 18
special characters (^, $, \), 17-18
text input mode, 4
UNIX, 1

MAIL REFERENCE MANUAL

Kurt Shoens

Version 2.10

June 15, 1983

1. Introduction

Mail provides a simple and friendly environment for sending and receiving mail. It divides incoming mail into its constituent messages and allows the user to deal with them in any order. In addition, it provides a set of *ed*-like commands for manipulating messages and sending mail. *Mail* offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to define and send to names which address groups of users. Finally, *Mail* is able to send and receive messages across such networks as the ARPANET, UUCP, and Berkeley network.

This document describes how to use the *Mail* program to send and receive messages. The reader is not assumed to be familiar with other message handling systems, but should be familiar with the UNIX[1] shell, the text editor, and some of the common UNIX commands. "The UNIX Programmer's Manual," "An Introduction to Csh," and "Text Editing with Ex and Vi" can be consulted for more information on these topics.

Here is how messages are handled: the mail system accepts incoming *messages* for you from other people and collects them in a file, called your *system mailbox*. When you login, the system notifies you if there are any messages waiting in your system mailbox. If you are a *csh* user, you will be notified when new mail arrives if you inform the shell of the location of your mailbox. On version 7 systems, your system mailbox is located in the directory /usr/spool/mail in a file with your login name. If your login name is "sam," then you can make *csh* notify you of new mail by including the following line in your .cshrc file:

```
set mail=/usr/spool/mail/sam
```

When you read your mail using *Mail*, it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author and the date they sent it.

---

[1] UNIX is a trademark of Bell Laboratories.

2. Common usage

The *Mail* command has two distinct usages, according to whether one wants to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, "root," use the shell command:

% Mail root

then type your message. When you reach the end of the message, type an EOT (control–d) at the beginning of a line, which will cause *Mail* to echo "EOT" and return you to the Shell. When the user you sent mail to next logs in, he will receive the message:

You have mail.

to alert him to the existence of your message.

If, while you are composing the message you decide that you do not wish to send it after all, you can abort the letter with a RUBOUT. Typing a single RUBOUT causes *Mail* to print

(Interrupt -- one more to kill letter)

Typing a second RUBOUT causes *Mail* to save your partial letter on the file "dead.letter" in your home directory and abort the letter. Once you have sent mail to someone, there is no way to undo the act, so be careful.

The message your recipient reads will consist of the message you typed, preceded by a line telling who sent the message (your login name) and the date and time it was sent.

If you want to send the same message to several other people, you can list their login names on the command line. Thus,

```
% Mail sam bob john
Tuition fees are due next Friday. Don't forget!!
<Control–d>
EOT
%
```

will send the reminder to sam, bob, and john.

If, when you log in, you see the message,

You have mail.

you can read the mail by typing simply:

% Mail

Mail will respond by typing its version number and date and then listing the messages you have waiting. Then it will type a prompt and await your command. The messages are assigned numbers starting with 1 – you refer to the messages with these numbers. *Mail* keeps tack of which messages are *new* (have been sent since you last read your mail) and *read* (have been read by you). New messages have an **N** next to them in the header listing and old, but unread messages have a **U** next to them. *Mail* keeps track of new/old and read/unread messages by putting a header field called "Status" into your messages.

To look at a specific message, use the **type** command, which may be abbreviated to simply **t**. For example, if you had the following messages:

```
N 1 root   Wed Sep 21 09:21  "Tuition fees"
N 2 sam    Tue Sep 20 22:55
```

you could examine the first message by giving the command:

type 1

which might cause *Mail* to respond with, for example:

```
Message 1:
From root  Wed Sep 21 09:21:45 1978
Subject: Tuition fees
Status: R

Tuition fees are due next Wednesday.  Don't forget!!
```

Many *Mail* commands that operate on messages take a message number as an argument like the **type** command. For these commands, there is a notion of a current message. When you enter the *Mail* program, the current message is initially the first one. Thus, you can often omit the message number and use, for example,

```
t
```

to type the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

```
1
```

would type the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *Mail* by simply typing a newline. As a special case, you can type a newline as your first command to *Mail* to type the first message.

If, after typing a message, you wish to immediately send a reply, you can do so with the **reply** command. **Reply**, like **type**, takes a message number as an argument. *Mail* then begins a message addressed to the user who sent you the message. You may then type in your letter in reply, followed by a <control-d> at the beginning of a line, as before. *Mail* will type EOT, then type the ampersand prompt to indicate its readiness to accept another command. In our example, if, after typing the first message, you wished to reply to it, you might give the command:

```
reply
```

Mail responds by typing:

```
To: root
Subject: Re: Tuition fees
```

and waiting for you to enter your letter. You are now in the message collection mode described at the beginning of this section and *Mail* will gather up your message up to a control−d. Note that it copies the subject header from the original message. This is useful in that correspondence about a particular matter will tend to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found will also be used. For example, if the letter had a "To:" header listing several recipients, *Mail* would arrange to send your replay to the same people as well. Similarly, if the original message contained a "Cc:" (carbon copies to) field, *Mail* would send your reply to *those* users, too. *Mail* is careful, though, not too send the message to *you*, even if you appear in the "To:" or "Cc:" field, unless you ask to be included explicitly. See section 4 for more details.

After typing in your letter, the dialog with *Mail* might look like the following:

```
reply
To: root
Subject: Tuition fees

Thanks for the reminder
```

EOT
&

The **reply** command is especially useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation. The **reply** command can be abbreviated to **r**.

Sometimes you will receive a message that has been sent to several people and wish to reply *only* to the person who sent it. **Reply** with a capital **R** replies to a message, but sends a copy to the sender only.

If you wish, while reading your mail, to send a message to someone, but not as a reply to one of your messages, you can send the message directly with the **mail** command, which takes as arguments the names of the recipients you wish to send to. For example, to send a message to "frank," you would do:

```
mail frank
This is to confirm our meeting next Friday at 4.
EOT
&
```

The **mail** command can be abbreviated to **m**.

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave *Mail*. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox* you can delete it using the **delete** command. In our example,

```
delete 1
```

will prevent *Mail* from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, *Mail* will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The **delete** command can be abbreviated to simply **d**.

Many features of *Mail* can be tailored to your liking with the **set** command. The **set** command has two forms, depending on whether you are setting a *binary* option or a *valued* option. Binary options are either on or off. For example, the "ask" option informs *Mail* that each time you send a message, you want it to prompt you for a subject header, to be included in the message. To set the "ask" option, you would type

```
set ask
```

Another useful *Mail* option is "hold." Unless told otherwise, *Mail* moves the messages from your system mailbox to the file *mbox* in your home directory when you leave *Mail*. If you want *Mail* to keep your letters in the system mailbox instead, you can set the "hold" option.

Valued options are values which *Mail* uses to adapt to your tastes. For example, the "SHELL" option tells *Mail* which shell you like to use, and is specified by

```
set SHELL=/bin/csh
```

for example. Note that no spaces are allowed in "SHELL=/bin/csh." A complete list of the *Mail* options appears in section 5.

Another important valued option is "crt." If you use a fast video terminal, you will find that when you print long messages, they fly by too quickly for you to read them. With the "crt" option, you can make *Mail* print any message larger than a given number of lines by sending it through the paging program *more*. For example, most CRT users should do:

set crt=24

to paginate messages that will not fit on their screens. *More* prints a screenful of information, then types --MORE--. Type a space to see the next screenful.

Another adaptation to user needs that *Mail* provides is that of *aliases*. An alias is simply a name which stands for one or more real user names. *Mail* sent to an alias is really sent to the list of real users associated with it. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The **alias** command in *Mail* defines an alias. Suppose that the users in a project are named Sam, Sally, Steve, and Susan. To define an alias called "project" for them, you would use the *Mail* command:

alias project sam sally steve susan

The **alias** command can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named "Bob Anderson" had the login name "anderson,"" you might want to use:

alias bob anderson

so that you could send mail to the shorter name, "bob."

While the **alias** and **set** commands allow you to customize *Mail*, they have the drawback that they must be retyped each time you enter *Mail*. To make them more convenient to use, *Mail* always looks for two files when it is invoked. It first reads a system wide file "/usr/lib/Mail.rc," then a user specific file, ".mailrc," which is found in the user's home directory. The system wide file is maintained by the system administrator and contains **set** commands that are applicable to all users of the system. The ".mailrc" file is usually used by each user to set options the way he likes and define individual aliases. For example, my .mailrc file looks like this:

set ask nosave SHELL=/bin/csh

As you can see, it is possible to set many options in the same **set** command. The "nosave" option is described in section 5.

Mail aliasing is implemented at the system-wide level by the mail delivery system *delivermail*. These aliases are stored in the file /usr/lib/aliases and are accessible to all users of the system. The lines in /usr/lib/aliases are of the form:

alias: $name_1$, $name_2$, $name_3$

where *alias* is the mailing list name and the $name_i$ are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing /usr/lib/aliases since the delivery system uses an indexed file created by *newaliases*.

We have seen that *Mail* can be invoked with command line arguments which are people to send the message to, or with no arguments to read mail. Specifying the **−f** flag on the command line causes *Mail* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file "letters" you can use *Mail* to read them with:

% Mail −f letters

You can use all the *Mail* commands described in this document to examine, modify, or delete messages from your "letters" file, which will be rewritten when you leave *Mail* with the **quit** command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read *mbox* in your home directory by using simply

% Mail −f

Normally, messages that you examine using the **type** command are saved in the file "mbox" in your home directory if you leave *Mail* with the **quit** command described below. If you wish to retain a message in your system mailbox you can use the **preserve** command to tell *Mail* to leave it there. The **preserve** command accepts a list of message numbers, just like **type** and may be abbreviated to **pre**.

Messages in your system mailbox that you do not examine are normally retained in your system mailbox automatically. If you wish to have such a message saved in *mbox* without reading it, you may use the **mbox** command to have them so saved. For example,

mbox 2

in our example would cause the second message (from sam) to be saved in *mbox* when the **quit** command is executed. **Mbox** is also the way to direct messages to your *mbox* file if you have set the "hold" option described above. **Mbox** can be abbreviated to **mb**.

When you have perused all the messages of interest, you can leave *Mail* with the **quit** command, which saves the messages you have typed but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

% Mail

The **quit** command can be abbreviated to simply **q**.

If you wish for some reason to leave *Mail* quickly without altering either your system mailbox or *mbox*, you can type the **x** command (short for **exit**), which will immediately return you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving *Mail*, you can type the command preceded by an exclamation point, just as in the text editor. Thus, for instance:

!date

will print the current date without leaving *Mail*.

Finally, the **help** command is available to print out a brief summary of the *Mail* commands, using only the single character command abbreviations.

3. Maintaining folders

Mail includes a simple facility for maintaining groups of messages together in folders. This section describes this facility.

To use the folder facility, you must tell *Mail* where you wish to keep your folders. Each folder of messages will be a single file. For convenience, all of your folders are kept in a single directory of your choosing. To tell *Mail* where your folder directory is, put a line of the form

```
set folder=letters
```

in your *.mailrc* file. If, as in the example above, your folder directory does not begin with a '/,' *Mail* will assume that your folder directory is to be found starting from your home directory. Thus, if your home directory is **/usr/person** the above example told *Mail* to find your folder directory in **/usr/person/letters**.

Anywhere a file name is expected, you can use a folder name, preceded with '+.' For example, to put a message into a folder with the **save** command, you can use:

```
save +classwork
```

to save the current message in the *classwork* folder. If the *classwork* folder does not yet exist, it will be created. Note that messages which are saved with the **save** command are automatically removed from your system mailbox.

In order to make a copy of a message in a folder without causing that message to be removed from your system mailbox, use the **copy** command, which is identical in all other respects to the **save** command. For example,

```
copy +classwork
```

copies the current message into the *classwork* folder and leaves a copy in your system mailbox.

The **folder** command can be used to direct *Mail* to the contents of a different folder. For example,

```
folder +classwork
```

directs *Mail* to read the contents of the *classwork* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including **type**, **delete**, and **reply**. To inquire which folder you are currently editing, use simply:

```
folder
```

To list your current set of folders, use the **folders** command.

To start *Mail* reading one of your folders, you can use the **−f** option described in section 2. For example:

```
% Mail −f +classwork
```

will cause *Mail* to read your *classwork* folder without looking at your system mailbox.

4. More about sending mail

4.1. Tilde escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. *Mail* provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

~p

which will print a line of dashes, the recipients of your message, and the text of the message so far. Since *Mail* requires two consecutive RUBOUT's to abort a letter, you can use a single RUBOUT to abort the output of ~p or any other ~ escape without killing your letter.

If you are dissatisfied with the message as it stands, you can invoke the text editor on it using the escape

~e

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. *Mail* will respond by typing

(continue)

after which you may continue typing text which will be appended to your message, or type <control-d> to end the message. A standard text editor is provided by *Mail*. You can override this default by setting the valued option "EDITOR" to something else. For example, you might prefer:

set EDITOR=/usr/ucb/ex

Many systems offer a screen editor as an alternative to the standard text editor, such as the *vi* editor from UC Berkeley. To use the screen, or *visual* editor, on your current message, you can use the escape,

~v

~v works like ~e, except that the screen editor is invoked instead. A default screen editor is defined by *Mail*. If it does not suit you, you can set the valued option "VISUAL" to the path name of a different editor.

It is often useful to be able to include the contents of some file in your message; the escape

~r filename

is provided for this purpose, and causes the named file to be appended to your current message. *Mail* complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text. The filename may contain shell metacharacters like * and ? which are expanded according to the conventions of your shell.

As a special case of ~r, the escape

~d

reads in the file "dead.letter" in your home directory. This is often useful since *Mail* copies the text of your message there when you abort a message with RUBOUT.

To save the current text of your message on a file you may use the

~w filename

escape. *Mail* will print out the number of lines and characters written to the file, after which you may continue appending text to your message. Shell metacharacters may be used in the filename, as in ~r and are expanded with the conventions of your shell.

If you are sending mail from within *Mail's* command mode you can read a message sent to you into the message you are constructing with the escape:

~m 4

which will read message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. Messages can also be forwarded without shifting by a tab stop with ~f. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

~t name1 name2 ...

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; you cannot remove someone from the recipient list with ~t.

If you wish, you can associate a subject with your message by using the escape

~s Arbitrary string of text

which replaces any previous subject with "Arbitrary string of text." The subject, if given, is sent near the top of the message prefixed with "Subject:" You can see what the message will look like by using ~p.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

~c name1 name2 ...

adds the named people to the "Cc:" list, similar to ~t. Again, you can execute ~p to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subject:" field, and the carbon copies the "Cc:" field. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape

~h

which prints "To:" followed by the current list of recipients and leaves the cursor (or printhead) at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard # and @ symbols,

~h
To: root kurt####bill

would change the initial recipients "root kurt" to "root bill." When you type a newline, *Mail* advances to the "Subject:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use ~p to print the current text of the header fields and the body of the message.

To effect a temporary escape to the shell, the escape

~!command

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

~|command

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, *Mail* assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt*, designed to format outgoing mail.

To effect a temporary escape to *Mail* command mode instead, you can use the

~:*Mail command*

escape. This is especially useful for retyping the message you are replying to, using, for example:

~:t

It is also useful for setting options and modifying aliases.

If you wish (for some reason) to send a message that contains a line beginning with a tilde, you must double it. Thus, for example,

~~This line begins with a tilde.

sends the line

~This line begins with a tilde.

Finally, the escape

~?

prints out a brief summary of the available tilde escapes.

On some terminals (particularly ones with no lower case) tilde's are difficult to type. *Mail* allows you to change the escape character with the "escape" option. For example, I set

set escape=]

and use a right bracket instead of a tilde. If I ever need to send a line beginning with right bracket, I double it, just as for ~. Changing the escape character removes the special meaning of ~.

4.2. Network access

This section describes how to send mail to people on other machines. Recall that sending to a plain login name sends mail to that person on your machine. If your machine is directly (or sometimes, even, indirectly) connected to the Arpanet, you can send messages to people on the Arpanet using a name of the form

name@host

where *name* is the login name of the person you're trying to reach and *host* is the name of the machine where he logs in on the Arpanet.

If your recipient logs in on a machine connected to yours by UUCP (the Bell Laboratories supplied network that communicates over telephone lines), sending mail to him is a bit more complicated. You must know the list of machines through which your message must travel to arrive at his site. So, if his machine is directly connected to yours, you can send mail to him using the syntax:

host!name

where, again, *host* is the name of his machine and *name* is his login name. If your message must go through an intermediate machine first, you must use the syntax:

intermediate!host!name

and so on. It is actually a feature of UUCP that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Talk to your system administrator about the machines connected to your site.

If you want to send a message to a recipient on the Berkeley network (Berknet), you use the syntax:

host:name

where *host* is his machine name and *name* is his login name. Unlike UUCP, you need not know the names of the intermediate machines.

When you use the **reply** command to respond to a letter, there is a problem of figuring out the names of the users in the "To:" and "Cc:" lists *relative to the current machine*. If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. So, when you **reply** to remote mail, the names in the "To:" and "Cc:" lists may change somewhat.

4.3. Special recipients

As described previously, you can send mail to either user names or **alias** names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a '/' in it or begins with a '+', it is assumed to be the path name of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (ie, one for which a '/' would not usually be needed) you can precede the name with './' So, to send mail to the file "memo" in the current directory, you can give the command:

% Mail ./memo

If the name begins with a '+,' it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the **alias** command for the group. Using our previous **alias** example, you might give the command:

alias project sam sally steve susan /usr/project/mail_record

Then, all mail sent to "project" would be saved on the file "/usr/project/mail_record" as well as being sent to the members of the project. This file can be examined using *Mail −f*.

It is sometimes useful to send mail directly to a program, for example one might write a project billboard program and want to access it using *Mail*. To send messages to the billboard program, one can send mail to the special name '|billboard' for example. *Mail* treats recipient names that begin with a '|' as a program to send the mail to. An **alias** can be set up to reference a '|' prefaced name if desired. *Caveats*: the shell treats '|' specially, so it must be quoted on the command line. Also, the '| program' must be presented as a single argument to mail. The safest course is to surround the entire name with double quotes.

This also applies to usage in the **alias** command. For example, if we wanted to alias 'rmsgs' to 'rmsgs −s' we would need to say:

alias rmsgs "| rmsgs -s"

5. Additional features

This section describes some additional commands of use for reading your mail, setting options, and handling lists of messages.

5.1. Message lists

Several *Mail* commands accept a list of messages as an argument. Along with **type** and **delete**, described in section 2, there is the **from** command, which prints the message headers associated with the message list passed to it. The **from** command is particularly useful in conjunction with some of the message list features described below.

A *message list* consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters "↑" "." or "$" to specify the first relevant, current, or last relevant message, respectively. *Relevant* here means, for most commands "not deleted" and "deleted" for the **undelete** command.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

    type 1−4

and to print all the messages from the current message to the last message, use

    type .−$

A *name* is a user name. The user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users that is *relevant* (in the sense described earlier) is selected. Thus, to print every message sent to you by "root," do

    type root

As a shorthand notation, you can specify simply "*" to get every *relevant* (same sense) message. Thus,

    type *

prints all undeleted messages,

    delete *

deletes all undeleted messages, and

    undelete *

undeletes all deleted messages.

You can search for the presence of a word in subject lines with **/**. For example, to print the headers of all messages that contain the word "PASCAL," do:

    from /pascal

Note that subject searching ignores upper/lower case differences.

5.2. List of commands

This section describes all the *Mail* commands available when receiving mail.

! Used to preface a command to be executed by the shell.

– The – command goes to the previous message and prints it. The – command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

Print
Like **print**, but also print out ignored header fields. See also **print** and **ignore**.

Reply
Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command. The **Reply** command is especially useful for replying to messages that were sent to enormous distribution groups when you really just want to send a message to the originator. Use it often.

Type
Identical to the **Print** command.

alias
Define a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example

alias project john sue willie kathryn

creates an alias *project* which expands to the four people John, Sue, Willie, and Kathryn.

alternates
If you have accounts on several machines, you may find it convenient to use the /usr/lib/aliases on all the machines except one to direct your mail to a single account. The **alternates** command is used to inform *Mail* that each of these other addresses is really *you*. *Alternates* takes a list of user names and remembers that they are all actually you. When you **reply** to messages that were sent to one of these alternate names, *Mail* will not bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism). If *alternates* is given no argument, it lists the current set of alternate names. **Alternates** is usually used in the .mailrc file.

chdir
The **chdir** command allows you to change your current directory. **Chdir** takes a single argument, which is taken to be the pathname of the directory to change to. If no argument is given, **chdir** changes to your home directory.

copyThe **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete
Deletes a list of messages. Deleted messages can be reclaimed with the **undelete** command.

dt The **dt** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail.

edit To edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of messages as described under the **type** command and processes each by writing it into the file Message*x* where *x* is the message number being edited and executing the text editor on it. When you have edited the message to your satisfaction, write the message out and quit, upon which *Mail* will read the message back and remove the file. **Edit** may be abbreviated to **e**.

else Marks the end of the then-part of an **if** statement and the beginning of the part to take effect if the condition of the **if** statement is false.

endif
Marks the end of an **if** statement.

exit Leave *Mail* without updating the system mailbox or the file your were reading. Thus, if you accidentally delete several messages, you can use **exit** to avoid scrambling your mailbox.

file The same as **folder**.

folders
List the names of the folders in your folder directory.

folder
The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read the new file. Some special conventions are recognized for the name:

Name	Meaning
#	Previous file read
%	Your system mailbox
%name	*Name*'s system mailbox
&	Your ~/mbox file
+folder	A file in your folder directory

from
The **from** command takes a list of messages and prints out the header lines for each one; hence

from joe

is the easy way to display all the message headers from "joe."

headers
When you start up *Mail* to read your mail, it lists the message headers that you have. These headers tell you who each message is from, when they were sent, how many lines and characters each message is, and the "Subject:" header field of each message, if present. In addition, *Mail* tags the message header of each message that has been the object of the **preserve** command with a "P." Messages that have been **saved** or **written** are flagged with a "*." Finally, **deleted** messages are not printed at all. If you wish to reprint the current list of message headers, you can do so with the **headers** command. The **headers** command (and thus the initial header listing) only lists the first so many message headers. The number of headers listed depends on the speed of your terminal. This can be overridden by specifying the number of headers you want with the *window* option. *Mail* maintains a notion of the current "window" into your messages for the purposes of printing headers. Use the **z** command to move forward and back a window. You can move *Mail's* notion of the current window directly to a

particular message by using, for example,

headers 40

to move *Mail's* attention to the messages around message 40. The **headers** command can be abbreviated to **h**.

help Print a brief and usually out of date help message about the commands in *Mail*. Refer to this manual instead.

hold Arrange to hold a list of messages in the system mailbox, instead of moving them to the file *mbox* in your home directory. If you set the binary option *hold*, this will happen by default.

if Commands in your ".mailrc" file can be executed conditionally depending on whether you are sending or receiving mail with the **if** command. For example, you can do:

```
if receive
        commands...
endif
```

An **else** form is also available:

```
if send
        commands...
else
        commands...
endif
```

Note that the only allowed conditions are **receive** and **send**.

ignore
Add the list of header fields named to the *ignore list*. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as *Via* which are not usually of interest. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

mail Send mail to one or more people. If you have the *ask* option set, *Mail* will prompt you for a subject to your message. Then you can type in your message, using tilde escapes as described in section 4 to edit, print, or modify your message. To signal your satisfaction with the message and send it, type control-d at the beginning of a line, or a . alone on a line if you set the option *dot*. To abort the message, type two interrupt characters (RUBOUT by default) in a row or use the **~q** escape.

mbox
Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next The **next** command goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

next root

goes to the next message sent by "root" and types it. The **next** command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters "↑" "." or "$". Thus,

prints the current message and

4

prints message 4, as described previously.

preserve
Same as **hold**. Cause a list of messages to be held in your system mailbox when you quit.

quit Leave *Mail* and update the file, folder, or system mailbox your were reading. Messages that you have examined are marked as "read" and messages that existed when you started are marked as "old." If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and you did *not* have *hold* set, all messages which have not been deleted, saved, or preserved will be moved to the file *mbox* in your home directory.

reply
Frame a reply to a single message. The reply will be sent to the person who sent you the message to which you are replying, plus all the people who received the original message, except you. You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command.

save It is often useful to be able to save messages on related topics in a file. The **save** command gives you ability to do this. The **save** command takes as argument a lit of message numbers, followed by the name of the file on which to save the messages. The messages are appended to the named file, thus allowing one to keep several messages in the file, stored in the order they were put there. The **save** command can be abbreviated to **s**. An example of the **save** command relative to our running example is:

s 1 2 tuitionmail

Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, unless explicitly specified.

set Set an option or give an option a value. Used to customize *Mail*. Section 5.3 contains a list of the options. Options can be *binary*, in which case they are *on* or *off*, or *valued*. To set a binary option *option on*, do

set option

To give the valued option *option* the value *value*, do

set option=value

Several options can be specified in a single **set** command.

shell
The **shell** command allows you to escape to the shell. **Shell** invokes an interactive shell and allows you to type commands to it. When you leave the shell, you will return to *Mail*. The shell used is a default assumed by *Mail*; you can override this default by setting the valued option "SHELL," eg:

set SHELL=/bin/csh

source
The **source** command reads *Mail* commands from a file. It is useful when

you are trying to fix your ".mailrc" file and you need to re-read it.

top The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**. If you wish, you can change the number of lines that **top** prints out by setting the valued option "toplines." On a CRT terminal,

set toplines=10

might be preferred.

type Print a list of messages on your terminal. If you have set the option *crt* to a number and the total number of lines in the messages you are printing exceed that specified by *crt*, the messages will be printed by a terminal paging program such as *more*.

undelete
The **undelete** command causes a message that had been deleted previously to regain its initial status. Only messages that have been deleted may be undeleted. This command may be abbreviated to **u**.

unset
Reverse the action of setting a binary or valued option.

visual
It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

Both the **edit** and **visual** commands assume some default text editors. These default editors can be overridden by the valued options "EDITOR" and "VISUAL" for the standard and screen editors. You might want to do:

set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi

write
The **save** command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the **write** command. The **write** command has the same syntax as the **save** command, and can be abbreviated to simply **w**. Thus, we could write the second message by doing:

w 2 file.c

As suggested by this example, the **write** command is useful for such tasks as sending and receiving source program text over the message system.

z *Mail* presents message headers in windowfuls as described under the **headers** command. You can move *Mail's* attention forward to the next window by giving the

z+

command. Analogously, you can move to the previous window with:

z–

5.3. Custom options

Throughout this manual, we have seen examples of binary and valued options. This section describes each of the options in alphabetical order, including some that you have not seen yet. To avoid confusion, please note that the options are either all lower case letters or all upper case letters. When I start a sentence such as: "Ask" causes *Mail* to prompt you for a subject header, I am only capitalizing "ask" as a courtesy to English.

EDITOR
The valued option "EDITOR" defines the pathname of the text editor to be used in the **edit** command and ~e. If not defined, a standard editor is used.

SHELL
The valued option "SHELL" gives the path name of your shell. This shell is used for the **!** command and ~! escape. In addition, this shell expands file names with shell metacharacters like * and ? in them.

VISUAL
The valued option "VISUAL" defines the pathname of your screen editor for use in the **visual** command and ~v escape. A standard screen editor is used if you do not define one.

append
The "append" option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, *Mail*will *mbox* in the same order that the system puts messages in your system mailbox. By setting "append," you are requesting that *mbox* be appended to regardless. It is in any event quicker to append.

ask "Ask" is a binary option which causes *Mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

askcc
"Askcc" is a binary option which causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline shows your satisfaction with the current list.

autoprint
"Autoprint" is a binary option which causes the **delete** command to behave like **dp** – thus, after deleting a message, the next one will be typed automatically. This is useful to quickly scanning and deleting messages in your mailbox.

dot "Dot" is a binary option which, if set, causes *Mail* to interpret a period alone on a line as the terminator of a message you are sending.

escape
To allow you to change the escape character used when sending mail, you can set the valued option "escape." Only the first character of the "escape" option is used, and it must be doubled if it is to appear as the first character of a line of your message. If you change your escape character, then ~ loses all its special meaning, and need no longer be doubled at the beginning of a line.

folder
The name of the directory to use for storing folders of messages. If this name begins with a '/' *Mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

ignore
The binary option "ignore" causes RUBOUT characters from your terminal to be ignored and echoed as @'s while you are sending mail. RUBOUT characters retain their original meaning in *Mail* command mode. Setting the "ignore" option is equivalent to supplying the **–i** flag on the command line as described in section 2.

ignoreeof
An option related to "dot" is "ignoreeof" which makes *Mail* refuse to accept a control–d as the end of a message. "Ignoreeof" also applies to

Mail command mode.

keep
The "keep" option causes *Mail* to truncate your system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you would do with the shell command:

chmod 600 /usr/spool/mail/yourname

where *yourname* is your login name. If you do not do this, anyone can probably read your mail, although people usually don't.

keepsave
When you **save** a message, *Mail* usually discards it when you **quit**. To retain all saved messages, set the "keepsave" option.

metoo
When sending mail to an alias, *Mail* makes sure that if you are included in the alias, that mail will not be sent to you. This is useful if a single alias is being used by all members of the group. If however, you wish to receive a copy of all the messages you send to the alias, you can set the binary option "metoo."

nosave
Normally, when you abort a message with two RUBOUTs, *Mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option "nosave" prevents this.

quiet
The binary option "quiet" suppresses the printing of the version when *Mail* is first invoked, as well as printing the for example "Message 4:" from the **type** command.

record
If you love to keep records, then the valued option "record" can be set to the name of a file to save your outgoing mail. Each new message you send is appended to the end of the file.

screen
When *Mail* initially prints the message headers, it determines the number to print by looking at the speed of your terminal. The faster your terminal, the more it prints. The valued option "screen" overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the **z** command.

sendmail
To alternate delivery system, set the "sendmail" option to the full pathname of the program to use. Note: this is not for everyone! Most people should use the default delivery system.

toplines
The valued option "toplines" defines the number of lines that the "top" command will print out instead of the default five lines.

6. Command line options

This section describes command line options for *Mail* and what they are used for.

−N Suppress the initial printing of headers.

−d Turn on debugging information. Not of general interest.

−f file
Show the messages in *file* instead of your system mailbox. If *file* is omitted, *Mail* reads *mbox* in your home directory.

−i Ignore tty interrupt signals. Useful on noisy phone lines, which generate spurious RUBOUT or DELETE characters. It's usually more effective to change your interrupt character to control−c, for which see the *stty* shell command.

−n Inhibit reading of /usr/lib/Mail.rc. Not generally useful, since /usr/lib/Mail.rc is usually empty.

−s string
Used for sending mail. *String* is used as the subject of the message being composed. If *string* contains blanks, you must surround it with quote marks.

−u name
Read *names's* mail instead of your own. Unwitting others often neglect to protect their mailboxes, but discretion is advised.

The following command line flags are also recognized, but are intended for use by programs invoking *Mail* and not for people.

−T file
Arrange to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. **−T** is for the *readnews* program and should NOT be used for reading your mail.

−h number
Pass on hop count information. *Mail* will take the number, increment it, and pass it with **−h** to the mail delivery system. **−h** only has effect when sending mail and is used for network mail forwarding.

−r name
Used for network mail forwarding: interpret *name* as the sender of the message. The *name* and **−r** are simply sent along to the mail delivery system. Also, *Mail* will wait for the message to be sent and return the exit status. Also restricts formatting of message.

Note that **−h** and **−r**, which are for network mail forwarding, are not used in practice since mail forwarding is now handled separately. They may disappear soon.

7. Format of messages

This section describes the format of messages. Messages begin with a *from* line, which consists of the word "From" followed by a user name, followed by anything, followed by a date in the format returned by the *ctime* library routine described in section 3 of the Unix Programmer's Manual. A possible *ctime* format date is:

Tue Dec 1 10:58:23 1981

The *ctime* date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as PDT.

Following the *from* line are zero or more *header field* lines. Each header field line is of the form:

name: information

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id*, *bcc*, *cc*, *from*, *reply-to*, *sender*, *subject*, and *to*. Other header fields are also significant to other systems; see, for example, the current Arpanet message standard for much more on this topic. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a system which encodes six bits into a printable character. For example, one could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Then, one can send a 16-bit binary number as three characters. These characters should be packed into lines, preferably lines about 70 characters long as long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

It should be noted that some network transport protocols enforce limits to the lengths of messages.

8. Glossary

This section contains the definitions of a few phrases peculiar to *Mail*.

alias
: An alternative name for a person or list of people.

flag
: An option, given on the command line of *Mail*, prefaced with a –. For example, **–f** is a flag.

header field
: At the beginning of a message, a line which contains information that is part of the structure of the message. Popular header fields include *to*, *cc*, and *subject*.

mail
: A collection of messages. Often used in the phrase, "Have you read your mail?"

mailbox
: The place where your mail is stored, typically in the directory /usr/spool/mail.

message
: A single letter from someone, initially stored in your *mailbox*.

message list
: A string used in *Mail* command mode to describe a sequence of messages.

option
: A piece of special purpose information used to tailor *Mail* to your taste. Options are specified with the **set** command.

9. Summary of commands, options, and escapes

This section gives a quick summary of the *Mail* commands, binary and valued options, and tilde escapes.

The following table describes the commands:

Command	*Description*
!	Single command escape to shell
-	Back up to previous message
Print	Type message with ignored fields
Reply	Reply to author of message only
Type	Type message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
chdir	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see **if**
edit	Edit a list of messages
else	Start of else part of conditional; see **if**
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as **file**
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of *Mail* commands
hold	Same as **preserve**
if	Conditional execution of *Mail* commands
ignore	Set/examine list of ignored header fields
mail	Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
quit	Leave *Mail*; update system mailbox, *mbox* as appropriate
reply	Compose a reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive shell
top	Print first so many (5 by default) lines of list of messages
type	Print messages
undelete	Undelete list of messages
unset	Undo the operation of a **set**
visual	Invoke visual editor on a list of messages
write	Append messages to a file, don't include headers
z	Scroll to next/previous screenful of headers

The following table describes the options. Each option is shown as being either a binary or valued option.

Option	*Type*	*Description*
EDITOR	*valued*	Pathname of editor for ~e and **edit**
SHELL	*valued*	Pathname of shell for **shell**, ~! and **!**
VISUAL	*valued*	Pathname of screen editor for ~v, **visual**
append	*binary*	Always append messages to end of *mbox*
ask	*binary*	Prompt user for Subject: field when sending
askcc	*binary*	Prompt user for additional Cc's at end of message
autoprint	*binary*	Print next message after **delete**
crt	*valued*	Minimum number of lines before using *more*
dot	*binary*	Accept . alone on line to terminate message input
escape	*valued*	Escape character to be used instead of ~
folder	*valued*	Directory to store folders in
hold	*binary*	Hold messages in system mailbox by default
ignore	*binary*	Ignore RUBOUT while sending mail
ignoreeof	*binary*	Don't terminate letters/command input with **↑D**
keep	*binary*	Don't unlink system mailbox when empty
keepsave	*binary*	Don't delete **save**d messages by default
metoo	*binary*	Include sending user in aliases
nosave	*binary*	Don't save partial letter in *dead.letter*
quiet	*binary*	Suppress printing of *Mail* version
record	*valued*	File to save all outgoing mail in
screen	*valued*	Size of window of message headers for **z**, etc.
sendmail	*valued*	Choose alternate mail delivery system
toplines	*valued*	Number of lines to print in **top**

The following table summarizes the tilde escapes available while sending mail.

Escape	*Arguments*	*Description*
~!	*command*	Execute shell command
~c	*name ...*	Add names to Cc: field
~d		Read *dead.letter* into message
~e		Invoke text editor on partial message
~f	*messages*	Read named messages
~h		Edit the header fields
~m	*messages*	Read named messages, right shift by tab
~p		Print message entered so far
~q		Abort entry of letter; like RUBOUT
~r	*filename*	Read file into message
~s	*string*	Set Subject: field to *string*
~t	*name ...*	Add names to To: field
~v		Invoke screen editor on message
~w	*filename*	Write message on file
~|	*command*	Pipe message through *command*
~~	*string*	Quote a ~ in front of *string*

The following table shows the command line flags that *Mail* accepts:

Flag	Description
−N	Suppress the initial printing of headers
−T *file*	Article-id's of read/deleted messages to *file*
−d	Turn on debugging
−f *file*	Show messages in *file* or *~/mbox*
−h *number*	Pass on hop count for mail forwarding
−i	Ignore tty interrupt signals
−n	Inhibit reading of /usr/lib/Mail.rc
−r *name*	Pass on *name* for mail forwarding
−s *string*	Use *string* as subject in outgoing mail
−u *name*	Read *name's* mail instead of your own

Notes: **−T**, **−d**, **−h**, and **−r** are not for human use.

10. Conclusion

Mail is an attempt to provide a simple user interface to a variety of underlying message systems. Thanks are due to the many users who contributed ideas and testing to *Mail*.

An Introduction to the Berkeley Network

Eric Schmidt

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

May 1979
(revised March 1980)

ABSTRACT

This document describes the use of a network between a number of UNIX† machines on the Berkeley campus. This network can execute commands on other machines, including file transfers, sending and receiving mail, remote printing, and shell-scripts.

The network operates in a batch-request mode. Network requests are queued up at the source and sent in shortest-first order to the destination machine. To do this, the requests are forwarded through a network of interconnected machines until they arrive at their destination where they are executed. The time this requires depends on system load, inter-machine transfer speed, and quantity of data being sent.

The network enforces normal UNIX security and demands a remote account with a password for most commands. Information can be returned to the user in files, for later processing, or on the terminal for immediate viewing.

Introduction

A network between a number of UNIX machines on the Berkeley campus has been implemented. This document is a brief introduction to the use of this network. Information which is specific to the local network has been gathered into Appendix A. The new user should read both this introduction and Appendix A in order to learn to use the network effectively.

This document is subdivided into the following sections:

†UNIX is a Trademark of Bell Laboratories.

Use of the Network
1) Copying Files over the Network
2) Listing Requests in the Network Queue
3) Removing Requests from the Network Queue
4) Sending Mail over the Network
5) Reading Mail over the Network
6) Using the Lineprinter over the Network
7) Net Prototype Command
Setting Defaults
How to Specify Remote Passwords
Appendix A: The Network at Berkeley
Appendix B: Getting Started — An Example

This manual is written in terms of three mythical machines, named X, Y, and Z. Specific names at Berkeley are in Appendix A, along with more local information.

Use of the Network

The network provides facilities for issuing a command on one machine (the *local* machine) which is to be executed on another (the *remote* machine). Network commands are available to transfer files from one machine to another, to send mail to a user on a remote machine, to retrieve one's mail from a remote account, or to print a file on a remote line-printer. These commands are described below, as is the more general *net* command which allows users to specify the name of some command or shell script to be executed on a remote machine. Network requests are queued on the local machine and sent to the remote machine, forwarded through intermediate machines if necessary.

Most of the network commands require that you have an account on the remote machine. If a remote account is not needed for a particular command, it will be noted in the following discussion. The first example introduces procedures and responses which are applicable to all network commands.

1. Copying Files over the Network

Suppose that you have accounts on both the X and Y machines and that you are presently logged into the X machine. If you want to copy a file named 'file1' from your current directory on machine X to machine Y (the *remote* machine), use the command:

```
% netcp file1 Y:file1
```

The net will make a copy of 'file1' in your login directory on the Y machine. (The 'Y:' will not be part of the filename on the Y machine.) In order to verify your permission to write into the Y account, the *netcp* command will prompt you with:

```
Name (Y:your-name):
```

You should respond with your login name on the Y machine, followed by a carriage-return. If you have the same login name on both machines, just type a carriage-return. Next a password will be requested:

```
Password (Y:remote-name):
```

Now type in your password followed by a carriage-return (you must type it even if your passwords are the same on both machines). The *netcp* command will make a copy of your 'file1' in a queue destined for the Y machine, and will then return you to the shell.

Likewise if you wanted to transfer a file named 'scan.p' from Y to X,

```
% netcp Y:scan.p scan.p
```

would place that file in your current directory on X.

The network will "write" you when it has executed your request (if you are still logged in), or will "mail" you a message (if you are not). You may use the −n option (described later) to disallow the interruption and thus force mail to be sent. A typical message might look like this:

> Message from Y:your-name at time ...
> (command: netcp file1 Y:file1, R: 0, sent April 1 18:03, took 10 min 3 sec)
> ———

The message includes the current time, the time you sent the command on machine X, and the exit code of the command (zero normally means success).

The network response will tell you if it was unable to execute the remote command successfully by returning an error message some time later. If, for example, you type the wrong password, you will get the response

> Message from Y:your-name at time ...
> (command: netcp file1 Y:file1, sent April 1 18:03, took 10 min 3 sec)
> Error: bad login/password your-name
> ———

The *netcp* command is actually a generalization of the UNIX *cp* command, similar to *uucp*†. Its syntax is:

> **netcp** [**−l** *login*] [**−p** *password*] [**−n**] [**−q**] [**−f**] *fromfile tofile*

where *fromfile* and *tofile* can be local or remote files. A filename which is not a full pathname is either from the current directory on the local machine or your login directory on the remote machine. The −l and −p options may be used to specify your remote login name and password on the command line. If the password contains shell meta-characters, it must be in quotes. (These options are useful in shell scripts, but be sure to make the shell script readable only by yourself if you've got passwords in it!) The −n option forces any responses from the remote machine to be mailed rather than written to you. The −f option forces prompting for a remote user name and password, even if they are set by other options or are in the ".netrc" file (see "Setting Defaults" below). Finally, the −q option prevents any confirmation messages from being sent back to you, if there were no errors, the exit code of the command is zero, and the command had no output.

Transferred files may or may not have the correct file protection mode; use the *chmod* (I) command to reset it. When files are to be brought from a remote machine, they are created zero-length at the time the command is issued; when they arrive, they assume their true length. Unlike *cp*, *netcp* does not allow the *tofile* to be simplified to a directory, if the files have the same name.

Examples:

% netcp file1 Y:file1	copy 'file1' from the current directory to Y
% netcp Y:file1 file1	copy 'file1' from Y to the current directory
% netcp Z:file1 Z:file2	*cp* command on remote machine
% netcp X:lex.c Y:lex.c	copy from X to Y
% netcp Y:subdir/file1 file1	copy from a sub-directory
% netcp file1 file2	an error— use the *cp* command

† See the UNIX Programmers Manual (Version 7 only).

2. Listing Requests in the Network Queue

To see where your command is in the queue, type

% netq

A typical output of which looks like:

From	To	Len	Code	Time	Command
X:your-name	Y:remote-name	100	b99999	Mar 23 18:05	netcp file1 Y:file1

The format is similar to that of the *lpq* command. The files are sent one at a time, in the order listed. If *netq* tells you the queue is empty, your request has been sent already. The queues for different destinations are totally separate.

% netq Y

will list just the queue destined for the Y machine. *Netq* summarizes requests from other users. The command

% netq —a

will print the requests from all users.

3. Removing Requests from the Network Queue

If you want to cancel your net request, and "b99999" (see the *netq* example above) is your "Code," use the command

% netrm b99999

which will remove the request (if it hasn't already been sent). Furthermore,

% netrm —

will remove all your net requests in the queues on the local machine (you must have made the request in order to remove it).

4. Sending Mail over the Network

To send mail to remote machines, use the *mail* command with the remote account prefixed by the destination machine's name and a ':'. "Y:schmidt", for example, refers to an account "schmidt" on the Y machine. The full sequence is illustrated below:

% mail Y:schmidt
{your message to user "schmidt" }
{control-d}

This will send to user "schmidt" on the Y machine the text you type in. As with intra-machine mail, the message is terminated by a control-d.

You do not need an account on a remote machine to send mail to a user there.

5. Reading Mail over the Network

It is also possible to read your mail on remote machines. From the X machine, the command

% netmail Y

sends a command to the Y machine to take any mail you may have and mail it back to you. As a precaution, the mail on the remote machine (Y in this example) is appended to the file

"mbox". Netmail has −l, −p, −n and −f options just like *netcp*. If a machine is not specified, the default machine† is used. If the −q option is specified (like *netcp)* no message is sent back if there is no mail.

Netmail also takes a −c option:

% netmail −c Y:username

which turns the command into a "mail check" command. A message is sent back telling the user whether the specified username has mail. No password is required. As above, the −q option suppresses the message if there is no mail. This command was designed to be put in C shell ".login" files.

6. Using the Lineprinter over the Network

Remote lineprinters can be used with the *netlpr* command:

netlpr [−m *machine*] [−c *command*] *file1 file2 ... filen*

which sends the files its arguments represent to the lineprinter on *machine*. It will prompt you for an account and password. The −l, −p, −n and −f options may be supplied, as in the *netcp* command. If the −c option is specified, a different printing *command* (default is "lpr") can be specified; see Appendix A for the list of printers allowed. Copies of the files are not made in the remote account.

7. Net Prototype Command

The above commands all use internally one more general command—the *net* command which has the following form:

net [−m *machine*] [−l *login*] [−p *password*] [−r *file*] [−] [−n] [−q] [−f] *command*

Net sends the given command to a remote machine. The machine may be specified either with the −m option or in the ".netrc" file (for the specific names, see Appendix A). If not specified, a default is used. −l, −p, −n, −q and −f are as explained above for the *netcp* command. The −r option indicates the local *file* which will receive the output (the standard output and standard error files) of *command* when it is executed on the remote machine. By default this output is written or mailed to you. Thus, for example, to find out who is on the Y machine when you are logged in on the X machine, execute the following command:

% net −m Y "who"

which will run the *who* command on the Y machine; the response will be written or mailed to you. Similarly,

% net −m Y −r resp "who"

will take the output (result) and return it to you in file 'resp' on the local machine. If instead you want the result of the *who* command to remain on the Y machine the command

% net −m Y "who >resp"

will create a file 'resp' in your login directory on the Y machine. It is a good idea to put the command in quotes, and it *must* be in quotes if I/O redirection (<, >, or other syntax special to the shell) is used.

If you do not specify the remote machine explicitly (or in the ".netrc" file, explained below), the default machine will be used (see Appendix A).

† (see "Setting Defaults" below)

The − option indicates that standard input from the local machine is to be supplied to the command executing remotely as standard input, thus if defaults for the login name and password are set up correctly as described below,

```
% net −m Y − "mail ripper"
   { message to ripper }
{control-d}
```

is equivalent to

```
% mail Y:ripper
   { message to ripper }
{control-d}
```

The net command also has other options not documented here. See the UNIX Programmer's Manual sections for more details.

Setting Defaults

Instead of repeatedly typing frequently-needed options for every invocation of the various network commands, the user may supply in his login directory a file ".netrc", which contains the repeated information. The ".netrc" file is typically used to specify login names on remote machines, as well as other options. An example of such a file is given below:

```
default Y
machine Y, login dracula
machine Z login dracula, quiet yes
```

This example sets the default machine to Y so that for net commands where a remote machine is not explicitly specified, the command will then be executed on the Y machine. The second and third lines indicate for the Y and Z machines a login name of "dracula" should be used to network commands, and to assume the "quiet" option on all commands destined for the Z machine. The complete list of options that may follow the machine indication is:

.netrc options for each machine			
Option	Parameter	Default	Comment
login	name	localname	login name for remote machine
password	password	(none)	password for remote login name
command	command	(none)	default command to be executed
write	yes/no	yes	if possible, write to user
force	yes/no	no	always prompt for name and password
quiet	yes/no	no	like the −q option

In setting up the ".netrc" file, if the "default" option is present, it must be the first line of the file. The information for each machine starts with the word "machine" and the machine name and continues one or more lines up to another machine indication (or the end of the file). Input is free-format. Multiple spaces, tabs, newlines, and commas serve as separators between words. Double quotes (") must surround passwords with blanks or special characters in them.

How to Specify Remote Passwords

For the commands which require the password for the account on the remote machine, there are a number of ways to specify the password:

1) letting the command ask you, as in the *netcp* example in Section 1,
2) specifying it with an alias (if using the C shell),
3) putting it into the current environment if the local machine is running UNIX Version 7,
4) specifying it on the command line with the −p option,
5) storing it in the ".netrc" file, described in the previous section.

These can be ranked in order of security, from 1 = greatest security to 5 = lowest security, from the point of view of security of passwords from unauthorized use by other users and possibly an illicit super-user. Each is described in turn:

1) If you make no effort to specify the remote password elsewhere, the network commands will prompt you with:

   Password(mach:username):

   Type your password, followed by a carriage return. This is the most secure mode of specifying passwords. If the net command is executed in the background (i.e. with "&") then the command can't read the password from your terminal and one of options 2-5 below must be used.

2) The alias feature of the C shell can be used to specify the remote password. The command

   % alias netcp netcp −l godzilla −p $pass

   in the ".cshrc" file, followed by

   % set path=your-passwd

   right before using the network will set for subsequent *netcp* commands the login name "godzilla" and password "passwd". This alias command must be given everytime you login (see the UNIX Programmers Manual section for the C shell (csh (1)) for more information about *alias*. Do *not* put this alias command in your ".login" file.

3) If running on a Version 7 UNIX system, the password can be put in the current environment. The command (to the C shell)

   % setenv MACH*mch* \`netlogin −m *mch*\`

   or (to the default Version 7 "Bourne" shell)

   % MACH*mch*=\`netlogin −m *mch*\`
   % export MACH*mch*

   will prompt you for a login name and password for the remote machine *mch* and put an encrypted copy of the password in your environment. (Note the back-quotes to the shell.) Subsequent network commands will find it in your environment and not prompt you for it. These encrypted passwords are invalidated after the user logs out. Type "man netlogin" for more information on the *netlogin* command.

4) Each net command takes a −p option on the command line to specify the password. These are usually put in shell command scripts. These shell script files should have file mode 0600 — use the chmod(I) command to set the mode.

5) The remote password can be specified in the user's ".netrc" file. If passwords are present, the ".netrc" file must have mode 0600 (as in #4 above).

The system managers recommend options 1-3 and warn against 4 and 5. Should someone break into your account on one machine, and you use option 4 or 5, you will have to change your passwords on all net machines for which your passwords have been stored in shell script files or in the ".netrc" file.

Log File

The file "/usr/spool/berknet/logfile" has a record of the most recent requests and responses, each line of which is dated. Lines indicating "sent" show the file name sent; lines indicating "rcv" show commands executed on the local machine (C:), their return code (R:), and their originator. For example, on the Y machine, the logfile:

```
Feb 28 10:29: rcv X: neil (neil) R: 0 C: netcp design Y:design
Feb 28 10:43: sent tuck to Z (z00466, 136 bytes, wait 2 min 3 sec)
Feb 28 11:05: rcv X: bill (bill) R: 0 C: netcp structures Y:structures
```

shows three entries. In this example, there are two *netcp* commands sending files from the X machine to Y, each from a different user. The second command sent was originated here by "tuck" and is 136 bytes long; the command that was sent is not shown. The command

% netlog

will print the last few lines of this file. Its prototype is

netlog −*num*

where *num* is an integer will print the last *num* lines from the file.

Acknowledgements

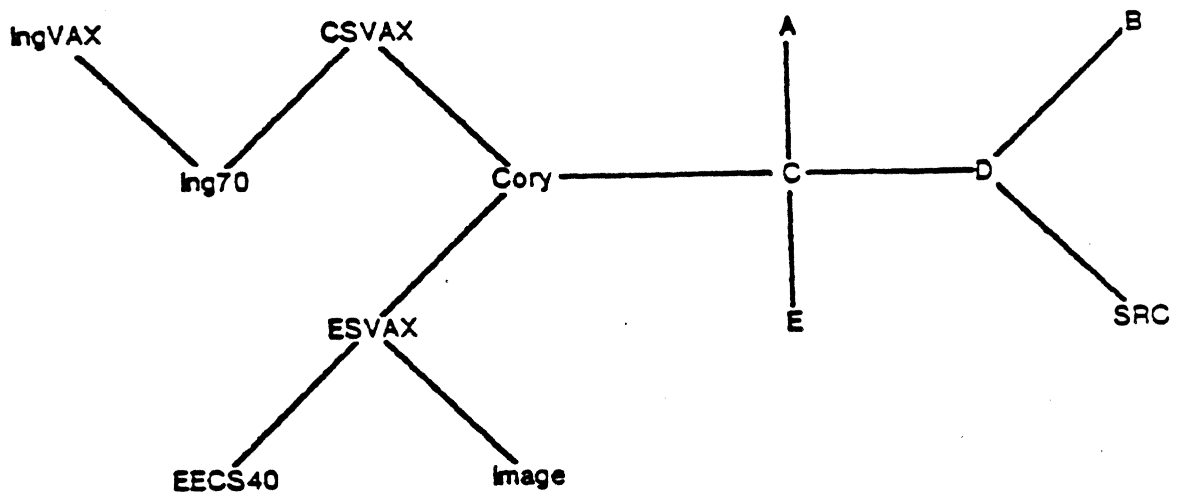
Special thanks go to Bob Fabry, Bill Joy, Vance Vaughan, Ed Gould, Robyn Allsman, Bob Kridle, Jeff Schriebman, Kirk Thege and Ricki Blau of Berkeley, and Dave Boggs of XEROX PARC for their help in making this network possible.

Appendix A

The Network at Berkeley

1. The Configuration (March 1, 1980)

The Current State of the Berkeley UNIX Network				
Machine Name	Internal Name	Run By	Default Machine	Other Name(s)
A.	A	Computer Center	C	
B	B	Computer Center	D	
C	C	Computer Center	A	
D	D	Computer Center	C	
E	E	Computer Center	C	
Ing70	I	INGRES Group	IngVAX	Ingres
IngVAX	J	INGRES Group	Ing70	
Image	M	Sakrison	ESVAX	
ESVAX	O	EE-CE Research	CSVAX	
SRC	S	Survey Res. Cent.	D	
CSVAX	V	CS Research	Cory	
Cory	Y	EECS Dept.	CSVAX	
EECS40	Z	EECS Dept.	ESVAX	



If a path exists from the local machine to the requested remote machine, the network will forward the request to the correct machine. Thus Cory users may communicate with all the other machines on the network as well as C and CSVAX (with a degradation in speed because of the intermediate machine(s)). The links between Ing70−IngVAX, Ing70−CSVAX, A−C, C−D, C−E, and B−D run at 9600 Baud, the other links run at 1200 Baud.

2. Documentation

The network commands *(net, netq, netrm, netlog, netcp, netmail, netlpr, netlogin)* are all documented in the UNIX Programmers Manual. For example,

% man netq

will print the *netq* manual section.

There are two more documents available:

Network System Manual
Berkeley Network Retrospective

The Manual is intended for the systems staff who will maintain the network. The Retrospective is my Master's report and details the history of the project, discusses the design, and lists future plans.

There is an up-to-date news file:

% news net

or

% help net

or

% cat /usr/net/news   (if those fail)

which prints news about the network, dated and with the most recent news first.

The UNIX Programmer's Manual, section I, has information on the *chmod, cp, mail, who,* and *write* commands mentioned in the text. Also, the *help* command has information about file protections:

% news access   (on the Cory machine)

or

% help permissions   (on the CC machines)

3. Features at Berkeley

a) There is a built-in character limit of 100,000 characters per single transmission, which cannot be overridden. The limit is 500,000 characters between the INGRES machines. Longer files must be split into smaller ones in order to be sent.

b) The 1200 Baud links between machines seldom transmit any faster than 50 characters per second (for 9600 Baud links, 350 characters a second), and can slow to a fraction of that in peak system loading periods. This is due to an expansion of the data packets to accomodate a seven-bit data path, wakeup time on the machines, and the packet sent in acknowledgement. Heavy file transfer is faster by magnetic tape.

c) On the CSVAX, IngVAX, and ESVAX the net commands are all in '/usr/ucb'. Your search path on these VAX's should be set to include the directory '/usr/ucb'; otherwise you will have to prefix all net commands by '/usr/ucb', as in '/usr/ucb/netcp'.

d) Limited Free Commands

Users who do not have accounts on remote machines may still execute certain commands by giving a remote login name of "network", and no remote password. The commands currently allowed are:

bpq	netlog	rcs	vpq	whom
epq	netq	rcslog	w	write
finger	ps	rcsq	where	yank
lpq	pstat	trq	who	

The *lpr* command is allowed on the INGRES machine. Also, *mail* to remote machines and *netlpr* between Computer Center machines do not require a remote account. The EECS40

machine allows no free commands (but allows the sending of mail).

For example, to execute an *lpq* command on the A machine, the user would type:

% net −l network −m A "who"

e) If no machine name specification is in the front of a full path name, the first four characters are checked and the machine is inferred from that if possible. In the command

% netcp file1 /ca/schmidt/file1

the second file name is equivalent to "C:file1", if you are "schmidt" on the C machine.

f) The network can only send files in one direction at a time. Thus confirmations can slow down heavy file transfer. If you regularly use a shell script to transfer a set of files, the −q option to *netcp* will improve transfer time.

g) The network creates a heavy load on the system and thus is expensive to run. If general user throughput is adversely affected, a charge will be implemented on the Computer Center machines.

h) When transferring files, quota overflow will result in a partial copy, so you should check the space requirements of the file being sent.

i) The Computer Center "A" machine's phototypesetter is usable from other network machines. If on one of the B-E machines, you do not need an account on the A machine. You simply type

% troff −Q other-options file(s)

instead of the normal

% troff other-options file(s)

The troff command is executed on the local machine and the phototypesetter instructions are sent to the A machine. You will be sent mail both when the file is queued and when it is finally typeset. To see your place in the *troff* queue, type:

% trq

on any Computer Center machine. There is a command

% trrm code

(where *code* is the code from the *trq* command) to remove queue files before they have been typeset. *Trrm* must be executed on the same machine from which the job was submitted.

If you are on a non-Computer Center machine, you may use the *nettroff* command:

% nettroff options file(s)

which is similar to the "troff −Q" command earlier. You will need an account on the A machine and the *trrm* command doesn't work from a non-Computer Center machine.

If using *nettroff*, no more than 15 pages may be sent to the typesetter. If using *troff* more than 15 pages may be sent only if the −s option is specified (see troff(1) for more information). The network will not transfer any file longer than 100,000 characters to the A machine. (It is best to aim for files of 25,000 characters or less)†. For more information, type

† Characters from *troff*'s output, not the user's source files. It is our general experience that *troff* outputs roughly twice as many characters as are in the source file (before any *eqn* or *tbl* preprocessing.)

man troff {on the Computer Center machines}

or

man nettroff {on the other machines}

The *nettroff* command is not supported by the Computer Center.

j) The *netlpr* command allows "epr", "bpr", and "vpr" as alternate lineprinters (using the −c option).

4. **Bugs in systems at Berkeley (As of March 1, 1980)**

a) If you are on the Computer Center machines using obsolete shells (/usr/pascal/sh, /usr/pascal/nsh) and have a ".profile" or ".shrc" file to change your shell prompt, you must make sure that you don't turn on "prompting" for non-interactive shells. This will interfere with the net commands. You should use this shell command to change your prompt:

```
${prompt?prompt=P}
```

where "P" is the prompt desired. This will avoid the problem.

If you set the variable *time* in the C shell, extraneous time stamps may appear in response messages. The correct way to set the variable *time* in the C shell is

```
if ($?prompt) then
    set time=num
endif
```

where *num* is the time interval in seconds.

b) The file mode should be preserved by *netcp* and it should be possible to default the second file name to a directory as in cp(I).

c) Various response messages are lost. This includes "fetching" files when the file being retrieved never arrives. I suspect this has something to do with unreliable delivery of error messages, but this is not reliably reproducible.

d) The network makes no provision for errors in transit on intermediate machines, such as "No more processes" or "File System Overflow". While these occur only rarely, when they do, no message or notification is sent to anyone.

e) The network commands are too slow on heavily-loaded instructional machines. The *net* command has to read the password file, ".netrc" file and the "/etc/utmp" file.

f) The queue files are normally sent shortest-job first. Unfortunately, under heavy loading the queue-search becomes too expensive and the network will choose the next file to send from the first 35 queue entries it finds in the queue directory, so the user should not depend on the requests being sent shortest-first.

g) Comments and bug discoveries are encouraged and can be sent by local or remote mail to "csvax:schmidt".

Appendix B

Getting Started -- An Example

The best way to start out is to follow this example. Suppose you're a Cory user, and you have accounts on the A and CSVAX machines.

1) Add a file ".netrc" (mode 600) to your login directory, as in the following example:

   default CSVAX
   machine A login *yourNameOnA*
   machine CSVAX login *yourNameOnCSVax*

   (If "default" occurs, it must be the first line of the file.)

2) Make sure that

   a) if you are on one of the VAX's, you have in your search path the directory '/usr/ucb'. Otherwise, on those machines you will have to prefix all commands by '/usr/ucb/' (e.g. '/usr/ucb/netcp').

   b) on the Computer Center machines, if you choose to set your shell prompt, you have done so correctly (details in Appendix A).

3) Then type

   % net "who"
   % netq

   which will send a *w* command to the CSVAX; some undetermined time later you will have written (or mailed) to you the output from the command executed on the CSVAX machine.

   The adventuresome may try:

   % net -m A "who"

   with the effect of being routed more slowly through an intermediate link in the net.

Berkeley Network Quick Reference

May 1979
(updated March 1980)

Command Summary	Example	Explanation
netcp *fromfile tofile*	% netcp defs.h C:defs.h	copy defs.h to C machine
	% netcp A:fig1 fig1	copy from A to here
	% netcp Cory:q1 D:q1	copy from Cory to D
mail *mach:user*	% mail A:alan	send mail to "alan" on A
	% mail A:alan C:george	multiple recipients
netmail [−c] [*mach*]	% netmail Cory	read mail on Cory machine
	% netmail −c Cory:usern	check mail on Cory machine
netlpr [−m *mach*] *file*	% netlpr −m D p1.p	print p1.p on D lineprinter
netq [−a] [*mach*]	% netq	list network queue
	% netq −a Cory	... all files to Cory
netrm [−] [*code*]	% netrm −	remove the user's requests
net [−m *mach*] [−] *command*	% net −m A who	send who to A machine
	% net − −mA lpr <f1	*lpr* file "f1" on A

The *netcp, netmail, netlpr,* and *net* commands need remote login names and passwords. They may be provided when prompted-for on the terminal, on the command line, or in a ".netrc" file in the user's login directory.

Other Options

−l *login*	login name on remote machine
−p *password*	password on remote machine
−f	prompt for login name and password
−n	mail back rather than write back
−q	silent mode− no messages sent back
−m *mach*	send to *mach* (only for *net, netlpr*)
−	input from standard input (only for *net*)

Example .netrc file:

```
default Z
machine Y login dracula
machine Z login schmidt, quiet yes
```

.netrc options for each machine			
Option	Parameter	Default	Comment
login	name	localname	login name for remote machine
password	password	(none)	password for remote login name
command	command	(none)	default command to be executed
write	yes/no	yes	if possible, write to user
force	yes/no	no	always prompt for name and password
quiet	yes/no	no	like the −q option

A TUTORIAL ON INGRES

by
Robert Epstein

Memorandum No. ERL - M77-25
December 15, 1977
(Revised)

Electronics Research Laboratory
College of Engineering
University of California, Berkeley
94720

A Tutorial on INGRES

This tutorial describes how to use the INGRES data base management system. You should be able to follow the the examples given here and observe the same results.

The data manipulation language supported by the INGRES system is called QUEL (QUEry Language). Complete information on QUEL and INGRES appears in the INGRES reference manual. This tutorial does not attempt to cover every detail of INGRES.

Begin by logging onto UNIX, the time sharing system under which INGRES runs. If at all possible, use a terminal that has both upper and lower case letters; otherwise life is going to be miserable for you. If you are on an upper case only terminal, type "\\" everywhere "\" appears in the tutorial.

There should currently be a "%" printed on your terminal. To start using INGRES type the command:

```
% ingres demo
```

This requests "UNIX" to invoke INGRES using the data base called "demo". After a few seconds, the following will appear:

```
INGRES version 6.1/0 login
Tue Aug 30 14:52:23 1977

COPYRIGHT
The Regents of the University of California
1977

This program material is the property of the
Regents of the University of California and
may not be reproduced or disclosed without
the prior written permission of the owner.

go
*
```

The first two lines include the INGRES version number (in this case version 6.1) and the current date. Following that is the "dayfile", which includes messages related to the INGRES system. The "go" indicates that INGRES is ready for your interactions.

The INGRES monitor prints an asterisk ("*") at the beginning of each line to remind you that INGRES is waiting for input.

Type the command:

```
* print parts
* \g
Executing . . .
```

The line "print parts" requests a printout of some data stored in the data base. The "\g" means "go". The message "Executing . . ." indicates that INGRES is processing your query. The following then appears:

parts relation

pnum	pname	color	weight	qoh
1	central processor	pink	10	1
2	memory	gray	20	32
3	disk drive	black	685	2
4	tape drive	black	450	4
5	tapes	gray	1	250
6	line printer	yellow	578	3
7	l-p paper	white	15	95
8	terminals	blue	19	15
13	paper tape reader	black	107	0
14	paper tape punch	black	147	0
9	terminal paper	white	2	350
10	byte-soap	clear	0	143
11	card reader	gray	327	0
12	card punch	gray	427	0

continue
*

What is printed on your terminal is the "parts relation". Intuitively, a relation is nothing more that a table with rows and columns.

In this case the relation name is "parts". There are five columns (we call them domains) named pnum (part number), pname (part name), color, weight, qoh (quantity on hand). Each row of the relation (called a tuple) represents one entry, which in this case represents one part in a computer installation. A relation can have up to 49 domains and a virtually unlimited number of tuples.

Notice that after the query is executed, INGRES prints "continue", while when we first entered INGRES it printed "go". As you enter a query INGRES saves what you type in a "workspace". If you ever mistype a query, typing "\r" will "reset" (ie. erase) your workspace. (Later on we will learn ways to edit mistakes so we don't have to retype the entire query.)

At any time you can see what is in the workspace by typing "\p". Try typing "\p":

```
* \p
print parts
*
```

The current contents of the workspace is printed. Now try typing "\r":

```
* \r
go
*
```

The workspace is now empty. Whenever INGRES types "continue" the workspace is non-empty; whenever INGRES types "go" the workspace is empty.

After a query is executed, INGRES typically types "continue". If you then type a new query, INGRES automatically erases the previous query, so you don't have to type "\r" after every query. This will be further explained as we proceed.

Using the "retrieve" command we can write specific queries about relations. As an example, let's have INGRES print only the "pname" domain of the parts relation. Type the command:

```
* range of p is parts
* retrieve (p.pname)
* \g
Executing . . .
```

pname
central processor
memory
disk drive
tape drive
tapes
line printer
l-p paper
terminals
paper tape reader
paper tape punch
terminal paper
byte-soap
card reader
card punch

continue

*

The output is just the pname domain from the parts relation. What we did required two steps. First we declared what is called a "tuple variable" and assigned it to range over the parts relation.

range of p is parts

What this means in English is that the letter "p" represents the parts relation. It may be thought of as a marker which moves down the "parts" relation to keep our place. INGRES remembers the association so that once p is declared to range over parts, we don't have to repeat the range declaration. This is useful when we are working with more than one relation, as will be seen later on.

Next we used the retrieve command. Its form is

retrieve (list here what you want retrieved)

"p" by itself refers to the parts relation. "p.pname" refers to the pname domain of the parts relation, so saying:

retrieve (p.pname)

means retrieve the pname domain of the parts relation.

Try the query to retrieve pname and color:

```
* retrieve p.pname, p.color
* \g
Executing . . .

2500: syntax error on line 1
last symbol read was: .

continue
*
```

Unfortunately we've made an error. INGRES tells us that it found a syntax error on the first line of the query. "Syntax error" means that we have typed something which INGRES cannot recognize. The error occured on line 1. INGRES makes a sometimes helpful and sometimes feeble attempt at diagnosing the problem. Whenever possible, INGRES tells us the last thing it read before it got confused.

In this case, the error is that the list of things to be retrieved (called the target list) must be enclosed in parenthesis. The correct query is:

```
* retrieve (p.pname, p.color)
* \g
Executing . . .


|pname                    |color   |
|----------------------------------|
|central processor        |pink    |
|memory                   |gray    |
|disk drive               |black   |
|tape drive               |black   |
|tapes                    |gray    |
|line printer             |yellow  |
|l-p paper                |white   |
|terminals                |blue    |
|paper tape reader        |black   |
|paper tape punch         |black   |
|terminal paper           |white   |
|byte-soap                |clear   |
|card reader              |gray    |
|card punch               |gray    |
|----------------------------------|

continue
*
```

You can restrict which tuples are printed by adding a "qualification" to the query. For example to get the name and color of only those parts which are gray, type:

```
* retrieve (p.pname, p.color)
* where p.color = "gray"
* \g
Executing . . .


|pname                    |color   |
|----------------------------------|
|memory                   |gray    |
|tapes                    |gray    |
|card reader              |gray    |
|card punch               |gray    |
|----------------------------------|

continue
*
```

Notice that INGRES prints only those parts where p.color is gray. Notice also that gray must be in quotes ("gray"). This is necessary. The only way INGRES will recognize character strings (e.g. words) is to enclose them in quotes.

What if we wanted part names for gray or pink parts? We only need to append to the previous query the phrase:

```
or p.color = "pink"
```

Remember, however, that if the next line typed begins a new query, INGRES will automatically reset the workspace. The workspace will be saved only if the next line begins with a command such as "\p" or "\g". (There are others which we will come to later.) If such a command is typed, the previous query is saved and anything further will be appended to that query.

Thus, by typing:

```
* \p
retrieve (p.pname, p.color)
where p.color = "gray"
*
```

you can see the previous query. Now type:

```
* or p.color = "pink"
*
```

INGRES appends that last line to the end of the query. You can verify this yourself by printing the workspace:

```
* \p
retrieve (p.pname, p.color)
where p.color = "gray"
or p.color = "pink"
*
```

Now run the query:

```
* \g
Executing . . .


|pname                   |color   |
|---------------------------------|
|central processor       |pink    |
|memory                  |gray    |
|tapes                   |gray    |
|card reader             |gray    |
|card punch              |gray    |
|---------------------------------|

continue
*
```

The rules about when the workspace is reset may be very confusing

at first. In general, INGRES will do exactly what you want without you having to think about it.

We have seen qualifications which used "or" and "=". In general one can use:

```
and
or
not
=       (equal)
!=      (not equal)
>       (greater than)
>=      (greater than or equal)
<       (less than)
<=      (less than or equal)
```

Evaluation occurs in the order the qualification was typed (ie. left to right). Parenthesis can be used to group things in any arbitrary order.

INGRES can do computations on the data stored in a relation. For example, the parts relation has quantity on hand and weight for each item. We might like to know the total weight for each group of parts (i.e. weight multiplied by qoh).

To get the name, part number and total weight for each part type the query:

```
* retrieve (p.pname, p.pnum, p.qoh * p.weight)
* \g
Executing . . .

2500: syntax error on line 1
last symbol read was: *

continue
*
```

Another error. The problem is that when a computation is done, INGRES does not know how to title the domain on the printout. For a simple domain, INGRES uses the domain name as a title. For anything else, you must create a new domain title by specifying:

```
tot = p.qoh * p.weight
```

More generally the form is:

```
title = expression
```

For example:

```
name = p.pname
```

```
computation = p.weight / 2000 * (p.qoh + 2)
```

Let's fix the error by retyping the query. As long as the first line after a query does not begin with a "\p" or "\g" then INGRES will automatically reset the workspace, erasing the previous query for us.

```
* retrieve (p.pname, p.pnum, tot=p.qoh * p.weight)
* \g
Executing . . .
```

pname	pnum	tot
central processor	1	10
memory	2	640
disk drive	3	1370
tape drive	4	1800
tapes	5	250
line printer	6	1734
l-p paper	7	1425
terminals	8	285
paper tape reader	13	0
paper tape punch	14	0
terminal paper	9	700
byte-soap	10	0
card reader	11	0
card punch	12	0

```
continue
*
```

In addition to multiplication, INGRES supports:

```
+     addition
-     subtraction (and unary negation)
/     division
*     multiplication
**    exponentiation (e.g. 3**10)
abs   absolute value (e.g. abs(p.qoh - 50) )
mod   modulo division
```

and many others. Please refer to the INGRES reference manual for a brief but complete description of what is supported.

If all we wanted were part numbers 2 or 10, then we could add the qualification:

```
where p.pnum = 2 or p.pnum = 10
```

CAUTION: if we just started typing "where p.pnum .... " INGRES

would understand this as the beginning of a new query and would reset the workspace. To avoid this you could type "\p" and force INGRES to print the workspace, or you can type "\a" (append). The append command guarantees that whatever else is typed will be appended to what is already in the workspace. This command is only needed immediately after a query is executed. Any other time data will be appended automatically. Try the following:

```
* \a
* where p.pnum = 2 or p.pnum = 10
* \g
Executing . . .


|pname                   |pnum  |tot   |
|--------------------------------------|
|memory                  |     2|   640|
|byte-soap               |    10|     0|
|--------------------------------------|

continue
*
```

To include all part numbers greater than 2 and less than or equal to 10:

```
* retrieve (p.pname, p.pnum, tot=p.qch * p.weight)
* where p.pnum > 2 and p.pnum <= 10
* \g
Executing . . .


|pname                   |pnum  |tot   |
|--------------------------------------|
|disk drive              |     3|  1370|
|tape drive              |     4|  1800|
|tapes                   |     5|   250|
|line printer            |     6|  1734|
|l-p paper               |     7|  1425|
|terminals               |     8|   285|
|terminal paper          |     9|   700|
|byte-soap               |    10|     0|
|--------------------------------------|

continue
*
```

Now, suppose we want to change the previous query to give results for part numbers between 5 and 10 instead of 2 and 10. You are probably annoyed at having to retype the entire query in order to change one character. Consequently, INGRES lets you use the UNIX text editor to make corrections and/or additions to your

workspace. At any time you can type "\e" and the INGRES monitor will write your workspace to a file and call the UNIX "ed" program. For example:

```
* \e
>>ed
83
```

The ">>ed" message tells you that you are now using the editor. The number 83 is the number of characters in your workspace.

We can now edit the query by changing the 2 to a 5. Included in the UNIX documentation is a tutorial on using the text editor. Rather than duplicating that tutorial, we will just use a few of the editor commands to illustrate how to do editing:

```
1p
retrieve (p.pname,p.pnum,tot = p.qoh * p.weight)
2p
where p.pnum > 2 and p.pnum <= 10
s/2/5/p
where p.pnum > 5 and p.pnum <= 10
w
83
q
<<monitor
*
```

Very briefly, this is what happens. "1p" and "2p" printed lines 1 and 2. "s/2/5/p" substitutes a 5 for a 2 on the current line (line 2), and then prints that line. "w" writes the query back to the INGRES workspace.

Inside the editor you can use any "ed" command except "e" (since e changes the file name). When you quit the editor (q command), the INGRES monitor will print "<<monitor" to remind you that you are back in INGRES. Notice that you MUST precede the "q" command with a "w" command to pass the corrected workspace back to INGRES.

To verify that the query is correct and to run it, type:

```
* \p\g
retrieve (p.pname,p.pnum,tot = p.qoh * p.weight)
where p.pnum > 5 and p.pnum <= 10
Executing . . .
```

pname	pnum	tot
line printer	6	1734
l-p paper	7	1425
terminals	8	285

terminal paper	9	700
byte-soap	10	0

continue
*

Having exhausted the interesting queries concerning the parts relation, lets now look at a new relation called "supply". Type:

```
* print supply
* \g
Executing . . .
```

supply relation

snum	pnum	jnum	shipdate	quan
475	1	1001	73-12-31	1
475	2	1002	74-05-31	32
475	3	1001	73-12-31	2
475	4	1002	74-05-31	1
122	7	1003	75-02-01	144
122	7	1004	75-02-01	48
122	9	1004	75-02-01	144
440	6	1001	74-10-10	2
241	4	1001	73-12-31	1
62	3	1002	74-06-18	3
475	2	1001	73-12-31	32
475	1	1002	74-07-01	1
5	4	1003	74-11-15	3
5	4	1004	75-01-22	6
20	5	1001	75-01-10	20
20	5	1002	75-01-10	75
241	1	1005	75-06-01	1
241	2	1005	75-06-01	32
241	3	1005	75-06-01	1
67	4	1005	75-07-01	1
999	10	1006	76-01-01	144
241	8	1005	75-07-01	1
241	9	1005	75-07-01	144

continue
*

The supply relation contains snum (the supplier number), pnum (the part number which is supplied by that supplier), jnum (the job number), shipdate (the date it was shipped), and quan (the quantity shipped).

To find out what parts are supplied by supplier number 122 type:

```
* retrieve (s.pnum) where s.snum = 122
* \g
Executing . . .

2109: line 1, Variable 's' not declared in RANGE statement

continue
*
```

We have referenced the tuple variable "s" (i.e. s.pnum) without telling INGRES what "s" represents. We are missing a range declaration. Retype the query as follows:

```
* range of s is supply
* retrieve (s.pnum) where s.snum = 122
* \g
Executing . . .


|pnum  |
|------|
|     7|
|     7|
|     9|
|------|

continue
*
```

Supplier number 122 supplies part numbers 7, 7 and 9. Note that 7 is listed twice. When retrieving tuples onto a terminal it is more efficient for INGRES NOT to check for duplicate tuples. INGRES can be forced to remove duplicate tuples. We will come to that later.

We now know that supplier 122 supplies part numbers 7 and 9. If you haven't run this query a few hundred times you probably don't know what part names correspond to part numbers 7 and 9. We could find out simply by running the query:

```
* retrieve (p.pname) where p.pnum = 7 or
* p.pnum = 9
* \g
Executing . . .


|pname               |
|--------------------|
|l-p paper           |
|terminal paper      |
```

```
|--------------------|

continue
*
```

After two queries we know by part name what parts are supplied by supplier number 122. We could do the same thing in one query by asking:

```
* retrieve (p.pname) where p.pnum = s.pnum
* and s.snum = 122
* \g
Executing . . .


|pname               |
|--------------------|
|l-p paper           |
|l-p paper           |
|terminal paper      |
|--------------------|

continue
*
```

Again note that "l-p paper" is duplicated. Look closely at this query. Note that the domain pnum exists in both the parts and supply relations. By saying p.pnum = s.pnum, we are logically joining the two relations.

Suppose we wished to find all suppliers who supply the central processor. We know that we will want to retrieve s.snum. We want only those s.snum's where the corresponding s.pnum is equal to the part number for the central processor.

If we find the p.pname which is equal to "central processor" then that will tell us the correct p.pnum. Finally we want s.pnum = p.pnum. The query is:

```
* retrieve (s.snum) where
* s.pnum = p.pnum and p.pname = "central processor"
* \g
Executing . . .


|snum  |
|------|
|   475|
|   475|
|   241|
|------|
```

```
continue
*
```

Let's abandon the parts and supply relations and try another. First, we can see what other relations are in the database by typing:

```
* help \g
* Executing . . .


 relation name     relation owner

 relation          ingres
 attribute         ingres
 indexes           ingres
 integrity         ingres
 constraint        ingres
 item              ingres
 sale              ingres
 employee          ingres
 dept              ingres
 supplier          ingres
 store             ingres
 parts             ingres
 supply            ingres


continue
*
```

Let's look at the "employee" relation. Since we know nothing about the relation we can also use the "help" command to learn about it. Type:

```
* help employee
* \g
Executing . . .


Relation:                  employee
Owner:                     ingres
Tuple width:               30
Saved until:               Fri Mar 25 11:01:30 1977
Number of tuples:          24
Storage structure:         paged heap
relation type:             user relation

 attribute name    type  length  keyno.

 number             i       2
 name               c      20
```

```
salary              i         2
manager             i         2
birthdate           i         2
startdate           i         2
```

```
continue
*
```

The help command lists overall information about the employee relation together with each attribute, its type and its length.

INGRES supports three data types: integer numbers, floating point numbers, and characters strings. Character domains can be from 1 to 255 characters in length. Integer domains can be 1, 2, or 4 bytes in length. This means that integers can obtain a maximum value of 127; 32,767; and 2,147,483,647 respectively. Floating point numbers can be either 4 or 8 bytes. Both hold a maximum value of about 10**38; with 7 or 17 digit accuracy respectively.

To look at all domains we could use the print command or we could use the retrieve command and list each domain in the target list. INGRES provides a shorthand way of doing just that. Try the following:

```
* range of e is employee
* retrieve (e.all)
* \g
Executing . . .
```

number	name	salary	manage	birthd	startd
157	Jones, Tim	12000	199	1940	1960
1110	Smith, Paul	6000	33	1952	1973
35	Evans, Michael	5000	32	1952	1974
129	Thomas, Tom	10000	199	1941	1962
13	Edwards, Peter	9000	199	1928	1958
215	Collins, Joanne	7000	10	1950	1971
55	James, Mary	12000	199	1920	1969
26	Thompson, Bob	13000	199	1930	1970
98	Williams, Judy	9000	199	1935	1969
32	Smythe, Carol	9050	199	1929	1967
33	Hayes, Evelyn	10100	199	1931	1963
199	Bullock, J.D.	27000	0	1920	1920
4901	Bailey, Chas M.	8377	32	1956	1975
843	Schmidt, Herman	11204	26	1936	1956
2398	Wallace, Maggie J.	7880	26	1940	1959
1639	Choy, Wanda	11160	55	1947	1970
5119	Ferro, Tony	13621	55	1939	1963
37	Raveen, Lemont	11985	26	1950	1974
5219	Williams, Bruce	13374	33	1944	1959

```
|  1523|Zugnoni, Arthur A.  | 19868|    129|  1928|  1949|
|   430|Brunet, Paul C.     | 17674|    129|  1938|  1959|
|   994|Iwano, Masahiro     | 15641|    129|  1944|  1970|
|  1330|Onstad, Richard     |  8779|     13|  1952|  1971|
|    10|Ross, Stanley       | 15908|    199|  1927|  1945|
|    11|Ross, Stuart        | 12067|      0|  1931|  1932|
|------------------------------------------------------|

continue
*
```

"All" is a keyword which is expanded by INGRES to become all domains. The domains are not guaranteed to be in any particular order. The previous query is equivalent to:

```
        range of e is employee
        retrieve (e.number, e.name, e.salary, e.manager
                e.birthdate, e.startdate)
```

Let's retrieve the salary of Stan Ross. At this point we will need to be able to type both upper and lower case letters. If you are on an upper case only terminal, type a single "\" before a letter you wish to capitalize. Thus on an upper case only terminal type "\ROSS, \STAN". If you are on an upper and lower case terminal, use the shift key to capitalize a letter.

Run the query:

```
* retrieve (e.name,e.salary)
* where e.name = "Ross, Stan"
* \g
Executing . . .


|name                   |salary|
|------------------------------|
|------------------------------|

continue
*
```

The result is empty. There is no e.name which satisfies the qualification. That's strange because we know there is a Stan Ross. However, INGRES does not know, for example, that "Stanley" and "Stan" are semantically the same.

To get the correct answer in this situation you may use the special "pattern matching" characters provided by INGRES.

One such character is "*". It matches any string of zero or more characters. Try the query:

```
* retrieve (e.name,e.salary)
* where e.name = "Ross, S*"
* \g
Executing . . .


|name                      |salary|
|---------------------------------|
|Ross, Stanley             | 15908|
|Ross, Stuart              | 12067|
|---------------------------------|

continue
*
```

In the first case "*" matched the string "tanley" and in the second case it matched "tuart".

Here is another example. Find the salaries of all people whose first name is "Paul":

```
* retrieve (e.name,e.salary)
* where e.name = "*,Paul*"
* \g
Executing . . .


|name                      |salary|
|---------------------------------|
|Smith, Paul               |  6000|
|Brunet, Paul C.           | 17674|
|---------------------------------|

continue
*
```

Notice that if we had asked for e.name = "*,Paul" we would not have gotten the second tuple. Also, INGRES ignores blanks in any character comparison whether using pattern matching characters or not. This means that the following would all give the same results:

```
        e.name = "Ross,Stanley"
        e.name = "Ross,   Stanley   "
        e.name = "R o s s,Stanley"
```

Particular characters or ranges of characters can be put in square brackets ([]). For example, find all people whose names start with "B" through "F":

```
* retrieve (e.name,e.salary)
* where e.name = "[B-F]*"
```

```
* \g
Executing . . .


|name                      |salary|
|---------------------------------|
|Evans, Michael            |  5000|
|Edwards, Peter            |  9000|
|Collins, Joanne           |  7000|
|Bullock, J.D.             | 27000|
|Bailey, Chas M.           |  8377|
|Choy, Wanda               | 11160|
|Ferro, Tony               | 13621|
|Brunet, Paul C.           | 17674|
|---------------------------------|

continue
*
```

Notice that this last query could be done another way:

```
* retrieve (e.name,e.salary)
* where e.name >"B" and e.name <"G"
* \g
Executing . . .


|name                      |salary|
|---------------------------------|
|Evans, Michael            |  5000|
|Edwards, Peter            |  9000|
|Collins, Joanne           |  7000|
|Bullock, J.D.             | 27000|
|Bailey, Chas M.           |  8377|
|Choy, Wanda               | 11160|
|Ferro, Tony               | 13621|
|Brunet, Paul C.           | 17674|
|---------------------------------|

continue
*
```

The two results are identical; however, the second way is generally more efficient for INGRES to process.

There are three types of pattern matching constructs. All three can be used in any combination for character comparison. They are:

```
    *   matches any length character string
    ?   matches any one (non-blank) character
    [ ] can match any character listed in the brackets.  If two
```

characters are separated by a dash (-), then it matches any character falling between the two characters.

The special meaning of a pattern matching character can be turned off by preceeding it with a "\". This means that "*" refers to the character "*".

We turn now to the aggregation facilities supported by INGRES. This allows a user to perform computations on whole domains of a relation. For example, one aggregate is average (avg). To compute the average salary for all employees, we enter:

```
* retrieve (avgsal=avg(e.salary))
* \g
Executing . . .


|avgsal     |
|-----------|
| 11867.520 |
|-----------|

continue
*
```

The particular title "avgsal" is arbitrary, but necessary; INGRES needs some sort of title for any expression in the target list (other than a simple domain).

We can also find the minimum and maximum salaries:

```
* retrieve (minsal=min(e.salary),maxsal=max(e.salary))
* \g
Executing . . .


|minsal|maxsal|
|-------------|
|  5000| 27000|
|-------------|

continue
*
```

If we wanted to know the names of the employees who make the minimum and maximum salaries, that query would be:

```
* retrieve (e.name, e.salary)
* where e.salary = min(e.salary) or e.salary = max(e.salary)
* \g
Executing . . .
```

```
|name                     |salary|
|--------------------------------|
|Evans, Michael           |  5000|
|Bullock, J.D.            | 27000|
|--------------------------------|
```

```
continue
*
```

INGRES supports the following aggregates:

```
        count
        min
        max
        avg
        sum
        any
```

We now indicate the query to list each employee along with the average salary for all employees:

```
* retrieve (e.name,peersal=avg(e.salary))
* \g
Executing . . .


|name                     |peersal    |
|-------------------------------------|
|Jones, Tim               | 11867.520|
|Smith, Paul              | 11867.520|
|Evans, Michael           | 11867.520|
|Thomas, Tom              | 11867.520|
|Edwards, Peter           | 11867.520|
|Collins, Joanne          | 11867.520|
|James, Mary              | 11867.520|
|Thompson, Bob            | 11867.520|
|Williams, Judy           | 11867.520|
|Smythe, Carol            | 11867.520|
|Hayes, Evelyn            | 11867.520|
|Bullock, J.D.            | 11867.520|
|Bailey, Chas M.          | 11867.520|
|Schmidt, Herman          | 11867.520|
|Wallace, Maggie J.       | 11867.520|
|Choy, Wanda              | 11867.520|
|Ferro, Tony              | 11867.520|
|Raveen, Lemont           | 11867.520|
|Williams, Bruce          | 11867.520|
|Zugnoni, Arthur A.       | 11867.520|
|Brunet, Paul C.          | 11867.520|
|Iwano, Masahiro          | 11867.520|
|Onstad, Richard          | 11867.520|
|Ross, Stanley            | 11867.520|
```

```
|Ross, Stuart           | 11867.520|
|----------------------------------|

continue
*
```

An aggregate always evaluates to a single value. To process the last query, INGRES replicated the average salary next to each e.name.

Aggregates can have their own qualification. For example, we can retrieve a list of each employee along with the average salary of those employees over 50.

```
* retrieve (e.name,peersal=
* avg(e.salary where 1977-e.birthdate > 50))
* \g
Executing . . .


|name                   |peersal   |
|----------------------------------|
|Jones, Tim             | 19500.000|
|Smith, Paul            | 19500.000|
|Evans, Michael         | 19500.000|
|Thomas, Tom            | 19500.000|
|Edwards, Peter         | 19500.000|
|Collins, Joanne        | 19500.000|
|James, Mary            | 19500.000|
|Thompson, Bob          | 19500.000|
|Williams, Judy         | 19500.000|
|Smythe, Carol          | 19500.000|
|Hayes, Evelyn          | 19500.000|
|Bullock, J.D.          | 19500.000|
|Bailey, Chas M.        | 19500.000|
|Schmidt, Herman        | 19500.000|
|Wallace, Maggie J.     | 19500.000|
|Choy, Wanda            | 19500.000|
|Ferro, Tony            | 19500.000|
|Raveen, Lemont         | 19500.000|
|Williams, Bruce        | 19500.000|
|Zugnoni, Arthur A.     | 19500.000|
|Brunet, Paul C.        | 19500.000|
|Iwano, Masahiro        | 19500.000|
|Onstad, Richard        | 19500.000|
|Ross, Stanley          | 19500.000|
|Ross, Stuart           | 19500.000|
|----------------------------------|

continue
*
```

Contrast the previous query with this next one. We will retrieve the names of those employees over fifty and retrieve the average salary for all employees.

```
* retrieve (e.name,peersal=avg(e.salary))
* where 1977-e.birthdate > 50
* \g
Executing . . .


|name                |peersal    |
|--------------------------------|
|James, Mary         | 11867.520|
|Bullock, J.D.       | 11867.520|
|--------------------------------|

continue
*
```

There is a very important distinction between these last two queries. An aggregate is completely self-contained. It is not affected by the qualification of the query as a whole.

In the first case, average is computed only for those employees over fifty, and all employees are retrieved. In the second case, however, average is computed for all employees but only those employees over 50 are retrieved.

If we wanted a list of all employees over fifty together with the average salary of employees over fifty, we would combine the previous two queries into one. That query would be:

```
* retrieve (e.name, peersal=
* avg(e.salary where 1977 - e.birthdate > 50))
* where 1977 - e.birthdate > 50
* \g
Executing . . .


|name                |peersal    |
|--------------------------------|
|James, Mary         | 19500.000|
|Bullock, J.D.       | 19500.000|
|--------------------------------|

continue
*
```

It is sometimes useful to have duplicate values removed before an aggregation is computed. For example if you wanted to know how many managers there are, the following query will not give the right answer:

```
* retrieve (bosses = count(e.manager))
* \g
* Executing . . .


|bosses        |
|--------------|
|            25|
|--------------|

continue
*
```

Notice that that gives the count of how many tuples there are in employee. What we want to know is how many unique e.manager's there are.

INGRES provides three special forms of aggregation.

countu	count unique values
avgu	average unique values
sumu	sum unique values

It's interesting to note that minu, maxu, and anyu are not needed. Their values would be the same whether duplicates were removed or not.

The correct query to find the number of managers is:

```
* retrieve (bosses=countu(e.manager))
* \g
Executing . . .


|bosses        |
|--------------|
|             9|
|--------------|

continue
*
```

Another aggregate facility supported by INGRES is called aggregate functions. Aggregate functions group data into categories and perform separate aggregations on each category.

For example, what if you wanted to retrieve each employee, and the average salary paid to employees with the same manager? That query would be:

```
* retrieve (e.name,manageravg=avg(e.salary by e.manager))
* \g
```

```
Executing . . .

|name                          |manageravg|
|-------------------------------------------|
|Jones, Tim                    | 11117.555|
|Thomas, Tom                   | 11117.555|
|Edwards, Peter                | 11117.555|
|James, Mary                   | 11117.555|
|Thompson, Bob                 | 11117.555|
|Williams, Judy                | 11117.555|
|Smythe, Carol                 | 11117.555|
|Hayes, Evelyn                 | 11117.555|
|Ross, Stanley                 | 11117.555|
|Smith, Paul                   |  9687.000|
|Williams, Bruce               |  9687.000|
|Evans, Michael                |  6688.500|
|Bailey, Chas M.               |  6688.500|
|Collins, Joanne               |  7000.000|
|Bullock, J.D.                 | 19533.500|
|Ross, Stuart                  | 19533.500|
|Schmidt, Herman               | 10356.333|
|Wallace, Maggie J.            | 10356.333|
|Raveen, Lemont                | 10356.333|
|Choy, Wanda                   | 12390.500|
|Ferro, Tony                   | 12390.500|
|Zugnoni, Arthur A.            | 17727.666|
|Brunet, Paul C.               | 17727.666|
|Iwano, Masahiro               | 17727.666|
|Onstad, Richard               |  8779.000|
|-------------------------------------------|

continue
*
```

The first nine people all have the same manager and their average salary is 11117.555. The next two people have the same manager and their average salary is 9687. etc.

Once again, if we wanted to see the same list just for those employees over 50:

```
* retrieve (e.name,manageravg=avg(e.salary by e.manager))
* where 1977-e.birthdate > 50
* \g
Executing . . .

|name                          |manageravg|
|-------------------------------------------|
|James, Mary                   | 11117.555|
|Bullock, J.D.                 | 19533.500|
```

```
|--------------------------------|

continue
*
```

Aggregate functions (unlike simple aggregates) are not completely local to themselves. The domains upon which the data is grouped (called the by-list) are logically connected to the domains in the rest of the query.

In these last examples, the "e.manager" in the by-list refers to the same tuple as "e.name" in the target list.

If we wanted to compute the average salaries by manager for only managers 33 and 199, then the query would be:

```
* retrieve (e.name,manageravg=
* avg(e.salary by e.manager)
* where e.manager = 199 or e.manager = 33
* \g
Executing . . .


|name                |manageravg|
|-------------------------------|
|Jones, Tim          | 11117.555|
|Thomas, Tom         | 11117.555|
|Edwards, Peter      | 11117.555|
|James, Mary         | 11117.555|
|Thompson, Bob       | 11117.555|
|Williams, Judy      | 11117.555|
|Smythe, Carol       | 11117.555|
|Hayes, Evelyn       | 11117.555|
|Ross, Stanley       | 11117.555|
|Smith, Paul         |  9687.000|
|Williams, Bruce     |  9687.000|
|-------------------------------|

continue
*
```

Suppose we wanted to find out how many people work for each manager, and in addition wanted only to include those employees who have worked at least seven years.

```
* retrieve (e.manager,people=count(e.name by e.manager where
* e.startdate < 1970))
* \g
Executing . . .


|manage|people        |
```

```
|---------------------|
|    199|            8|
|     33|            2|
|     32|            0|
|     10|            0|
|      0|            2|
|     26|            2|
|     55|            1|
|    129|            2|
|     13|            0|
|---------------------|

continue
*
```

Notice that managers 32, 10, and 13 have no employees who started before 1970. Now suppose we want to know the average salary for those employees. Simply change "count" to "avg" and rerun the query.

```
* retrieve (e.manager,people=avg(e.salary by e.manager where
* e.startdate < 1970))
* \g
Executing . . .


|manage|people          |
|-----------------------|
|    199| 10882.250|
|     33| 22687.000|
|     32|     0.000|
|     10|     0.000|
|      0| 19533.500|
|     26|  9542.000|
|     55| 13621.000|
|    129| 18771.000|
|     13|     0.000|
|-----------------------|

continue
*
```

Notice what INGRES does for managers 32, 10 and 13. The average salary for those manager employees is actually undefined since there are no employees who started before 1970. INGRES always makes undefined values zero in aggregates.

If you want to remove the zero values from the output, a qualification can be added to the query. The following query will find the average salaries only for those which are greater than zero.

```
* retrieve (e.manager,people=avg(e.salary by e.manager where
```

```
* e.startdate < 1970))
* where avg(e.salary by e.manager where e.startdate < 1970) > 0
* \g
Executing . . .
```

manage	people
199	10882.250
33	22687.000
0	19533.500
26	9542.000
55	13621.000
129	18771.000

```
continue
*
```

Up until now we have been retrieving results directly onto the terminal. You can also save results by retrieving them into a new relation. This is done by saying:

```
        retrieve into newrel ( ... )
        where . . .
```

The rules are exactly the same as for retrieves onto the terminal. INGRES will create the new relation with the correct domains, and then put the results of the query in the new relation.

For example, create a new relation called "overpaid" which has only those employees who make more than $8000:

```
* retrieve into overpaid (e.all)
* where e.salary > 8000
* print overpaid
* \g
Executing . . .
```

overpaid relation

number	name	salary	manage	birthd	startd
10	Ross, Stanley	15908	199	1927	1945
11	Ross, Stuart	12067	0	1931	1932
13	Edwards, Peter	9000	199	1928	1958
26	Thompson, Bob	13000	199	1930	1970
32	Smythe, Carol	9050	199	1929	1967
33	Hayes, Evelyn	10100	199	1931	1963
37	Raveen, Lemont	11985	26	1950	1974

55	James, Mary	12000	199	1920	1969
98	Williams, Judy	9000	199	1935	1969
129	Thomas, Tom	10000	199	1941	1962
157	Jones, Tim	12000	199	1940	1960
199	Bullock, J.D.	27000	0	1920	1920
430	Brunet, Paul C.	17674	129	1938	1959
843	Schmidt, Herman	11204	26	1936	1956
994	Iwano, Masahiro	15641	129	1944	1970
1330	Onstad, Richard	8779	13	1952	1971
1523	Zugnoni, Arthur A.	19868	129	1928	1949
1639	Choy, Wanda	11160	55	1947	1970
4901	Bailey, Chas M.	8377	32	1956	1975
5119	Ferro, Tony	13621	55	1939	1963
5219	Williams, Bruce	13374	33	1944	1959

```
continue
*
```

On a "retrieve into" nothing is printed. We had to include a "print" command to see the results. Also, the relation name on a "retrieve into" must not already exist. For example, if we tried the same query again:

```
* \g
Executing . . .

5102: CREATE: duplicate relation name overpaid

continue
*
```

There are two special features about a "retrieve into". First, the result relation is automatically sorted and any duplicate tuples are removed. Second, the relation becomes part of the data base and is owned by you. If you don't want it to be saved you should remember to destroy it. The mechanism for destroying a relation will be mentioned a bit later.

So far we have only retrieved data but never changed it. INGRES supports three update commands: append, replace, and delete.

For example, to add "Tom Terrific" to the list of overpaid employees and start him off at $10000:

```
* append to overpaid(name = "Terrific, Tom",salary = 10000)
* \g
Executing . . .


continue
*
```

Notice that we specified values for only two of the six domains in "overpaid". That is fine. INGRES will automatically set numeric domains to zero and character domains to blank, if they are not specified.

Notice also that INGRES did not print anything after the query. This is true for all update commands.

Let's give everyone in overpaid a 10% raise. To do this we want to replace o.salary by 1.1 times its value. Type the query:

```
* range of o is overpaid
* replace o(salary = o.salary * 1.1)
* \g
Executing . . .


continue
*
```

While the append command requires that you give a relation name (e.g. append to overpaid), the replace and delete commands require a tuple variable. Note that the command is:

```
        replace o ( . . . )
                where . . .
```

and not:

```
        replace overpaid ( . . . )
                where . . .
```

Print the results of these last two updates:

```
* print overpaid
* \g
Executing . . .
```

overpaid relation

number	name	salary	manage	birthd	startd
10	Ross, Stanley	17498	199	1927	1945
11	Ross, Stuart	13273	0	1931	1932
13	Edwards, Peter	9899	199	1928	1958
26	Thompson, Bob	14299	199	1930	1970
32	Smythe, Carol	9954	199	1929	1967
33	Hayes, Evelyn	11109	199	1931	1963
37	Raveen, Lemont	13183	26	1950	1974
55	James, Mary	13199	199	1920	1969
98	Williams, Judy	9899	199	1935	1969

```
    129|Thomas, Tom            | 10999|   199|  1941|  1962|
    157|Jones, Tim             | 13199|   199|  1940|  1960|
    199|Bullock, J.D.          | 29699|     0|  1920|  1920|
    430|Brunet, Paul C.        | 19441|   129|  1938|  1959|
    843|Schmidt, Herman        | 12324|    26|  1936|  1956|
    994|Iwano, Masahiro        | 17205|   129|  1944|  1970|
   1330|Onstad, Richard        |  9656|    13|  1952|  1971|
   1523|Zugnoni, Arthur A.     | 21854|   129|  1928|  1949|
   1639|Choy, Wanda            | 12275|    55|  1947|  1970|
   4901|Bailey, Chas M.        |  9214|    32|  1956|  1975|
   5119|Ferro, Tony            | 14983|    55|  1939|  1963|
   5219|Williams, Bruce        | 14711|    33|  1944|  1959|
      0|Terrific, Tom          | 11000|     0|     0|     0|
|-----------------------------------------------------------|

continue
*
```

Let's fire whoever has the smallest salary:

```
* delete o where o.salary = min(o.salary) \g
Executing . . .


continue
*
```

Notice that the delete command requires a tuple variable (eg. delete o) and not a relation name.

What if we wanted to know who makes more that Tom Terrific? The query to do this is very subtle. First we use a new tuple variable called "t" which ranges over overpaid, and will be used to refer to Tom. t.name must equal "Terrific, Tom". Next, we use a tuple variable called "o" which will scan the whole relation. If we ever find an o.salary > t.salary then o.name must make more than Tom.

The complete query is:

```
* range of t is overpaid
* retrieve (o.name, osal=o.salary, tomsal = t.salary)
* where o.salary > t.salary
* and t.name = "Terrific, Tom"
* \g
* Executing . . .


|name                 |osal  |tomsal|
|-----------------------------------|
|Ross, Stanley        | 19247| 11000|
|Ross, Stuart         | 14600| 11000|
```

```
|Thompson, Bob        |  15728|  11000|
|Hayes, Evelyn        |  12219|  11000|
|Raveen, Lemont       |  14501|  11000|
|James, Mary          |  14518|  11000|
|Thomas, Tom          |  12098|  11000|
|Jones, Tim           |  14518|  11000|
|Bullock, J.D.        |  32668|  11000|
|Brunet, Paul C.      |  21385|  11000|
|Schmidt, Herman      |  13556|  11000|
|Iwano, Masahiro      |  18925|  11000|
|Zugnoni, Arthur A.   |  24039|  11000|
|Choy, Wanda          |  13502|  11000|
|Ferro, Tony          |  16481|  11000|
|Williams, Bruce      |  16182|  11000|
|-----------------------------------|

continue
*
```

If we wanted to give Tom Terrific $50 more than anyone else, the query would be:

```
* replace o(salary = max(o.salary) + 50)
* where o.name = "Terrific, Tom"
* \g
Executing . . .


continue
*
```

Finally, to destroy a relation owned by yourself, type the command:

```
* destroy overpaid
* \g
Executing . . .


Continue
*
```

We are now ready to leave INGRES. This is done either by typing an end-of-file (control/d) or more typically use the "\q" command:

```
* \q
INGRES vers 6.1/0 logout
Tue Aug 30 14:55:20 1977
goodbye bob -- come again
```

Berkeley Pascal User's Manual
Version 1.1 — April, 1979

*William N. Joy**

*Susan L. Graham**

*Charles B. Haley***

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

ABSTRACT

Berkeley Pascal is designed for interactive instructional use and runs on the PDP/11 and VAX/11 computers. It produces interpretive code, providing fast translation at the expense of slower execution speed. An execution profiler and Wirth's cross reference program are also available with the system.

The system supports full Pascal, with the exception of **procedure** and **function** names as parameters. The language accepted is very close to 'standard' Pascal, with only a small number of extensions.

The *User's Manual* gives a list of sources relating to the UNIX† system, the Pascal language, and the Berkeley Pascal system. Basic usage examples are provided for the Pascal interpreter components *pi*, *px*, *pix*, and *pxp*. Errors commonly encountered in these programs are discussed. Details are given of special considerations due to the interactive implementation. A number of examples are provided including many dealing with input/output. An appendix supplements Wirth's *Pascal Report* to form the full definition of the Berkeley implementation of the language.

December 11, 1979

This Manual can be used with Berkeley Pascal Version 1.2, which is currently running on our systems. Changes between Versions 1.1 and 1.2 were mostly bug fixes, so there is no new documentation.

(C) 1977, 1979 William N. Joy, Susan L. Graham, Charles B. Haley

* Author's current address: S & B Associates, 1110 Centennial Ave., Piscataway, NJ 08854

†UNIX is a Trademark of Bell Laboratories.

Berkeley Pascal User's Manual
Version 1.1 — April, 1979

*William N. Joy**

*Susan L. Graham**

*Charles B. Haley***

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

Introduction

The Berkeley Pascal *User's Manual* consists of five major sections and an appendix. In section 1 we give sources of information about UNIX,† about the programming language Pascal, and about the Berkeley implementation of the language. Section 2 introduces the Berkeley implementation and provides a number of basic examples. Section 3 discusses the error diagnostics produced by the translator *pi* and the runtime interpreter *px*. Section 4 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX. Section 5 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

History of the implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system result from the work of Charles Haley and William Joy during the latter half of 1976. Earlier versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick in the spring of 1979.

---

* The financial support of the first and second authors' work by the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291, and the first author's work by an IBM Graduate Fellowship are gratefully acknowledged.

** Author's present address: Bell Laboratories, Murray Hill, NJ 07974.

†UNIX is a Trademark of Bell Laboratories.

1. Sources of information

This section lists the resources available on the UC Berkeley campus for information about general features of UNIX, text editing, the Pascal language, and the Berkeley Pascal implementation, concluding with a list of references. The available documents include both so-called standard documents — those distributed with all UNIX system — and documents (such as this one) written at Berkeley.

1.1. Where to get documentation

On the UC Berkeley campus, documentation is available at the Computer Center Library, room 218B Evans Hall. The library is open from 8:00 A.M. to 5:00 P.M. Monday through Friday. Current documentation for most of the UNIX system is also available "on line" at your terminal. Details on getting such documentation interactively are given in section 1.3.

1.2. Computer Center short courses

For those not enrolled in Computer Science Division courses, and who have no prior experience using UNIX, the short-courses offered by the staff of the Computer Center are highly recommended. These courses are offered free of charge, and are usually held at the beginning of each quarter. The two most valuable short courses for the Berkeley Pascal user are the ones dealing with basic use of UNIX, and with text editing. If you are unable to attend the short courses, documents for these courses are available at the Computer Center Library, and are recommended. The documents are in a tutorial format, so it is possible to use them on your own.

1.3. Documentation describing UNIX

The following documents are those recommended as tutorial and reference material about the UNIX system. We give the documents with the introductory and tutorial materials first, the reference materials last.

UNIX For Beginners — Second Edition

This document is the basic tutorial for UNIX available with the standard system.

Communicating with UNIX

This is also a basic tutorial on the system and assumes no previous familiarity with computers; it was written at Berkeley and is used in the short courses.

An introduction to the C shell

This document introduces *csh*, the shell in common use at Berkeley, and provides a good deal of general description about the way in which the system functions. It provides a useful glossary of terms used in discussing the system.

UNIX Programmer's Manual

This manual is the major source of details on the components of the UNIX system. It consists of an Introduction, a permuted index, and eight command sections. Section 1 consists of descriptions of most of the "commands" of UNIX. Most of the other sections have limited relevance to the user of Berkeley Pascal, being of interest mainly to system programmers. The manual is available from the Computer Center Library.

UNIX documentation often refers the reader to sections of the manual. Such a reference consists of a command name and a section number or name. An example of such a reference would be: *ed* (1). Here *ed* is a command name — the standard UNIX text editor, and '(1)' indicates that its documentation is in section 1 of the manual.

The pieces of the Berkeley Pascal system are *pi* (1), *px* (1), the combined Pascal translator and interpretive executor *pix* (1), the Pascal execution profiler *pxp* (1), and the Pascal cross-

reference generator *pxref* (1).

It is possible to obtain a copy of a manual section by using the *man* (1) command. To get the Pascal documentation just described one could issue the command:

% **man pi**

to the shell. The user input here is shown in **bold face**; the '% ', which was printed by the shell as a prompt, is not. Similarly the command:

% **man man**

asks the *man* command to describe itself.

1.4. Text editing documents

The following documents introduce the various UNIX text editors. Most Berkeley users will use a version of the text editor *ex;* either *edit,* which is a version of *ex* for new and casual users, *ex* itself, or *vi* (visual) which focuses on the display editing portion of *ex.*

A Tutorial Introduction to the UNIX Text Editor

This document, written by Brian Kernighan of Bell Laboratories, is a tutorial for the standard UNIX text editor *ed.* It introduces you to the basics of text editing, and provides enough information to meet day-to-day editing needs, for *ed* users.

Edit: A tutorial

This introduces the use of *edit,* an editor similar to *ed* which provides a more hospitable environment for beginning users. The short courses on editing taught by the Computer Center use this document.

Ex/edit Command Summary

This summarizes the features of the editors *ex* and *edit* in a concise form. If you have used a line oriented editor before this summary alone may be enough to get you started.

Ex Reference Manual — Version 3.1

A complete reference on the features of *ex* and *edit.*

An Introduction to Display Editing with Vi

Vi is a display oriented text editor. It can be used on most any CRT terminal, and uses the screen as a window into the file you are editing. Changes you make to the file are reflected in what you see. This manual serves both as an introduction to editing with *vi* and a reference manual.

Vi Quick Reference

This reference card is a handy quick guide to *vi;* you should get one when you get the introduction to *vi.*

1.5. Pascal documents — The language

This section describes the documents on the Pascal language which are likely to be most useful to the Berkeley Pascal user. Complete references for these documents are given in section 1.7.

Pascal User Manual

By Kathleen Jensen and Niklaus Wirth, the *User Manual* provides a tutorial introduction to the features of the language Pascal, and serves as an excellent quick-reference to the language. The reader with no familiarity with Algol-like languages may prefer one of the Pascal text books listed below, as they provide more examples and explanation. Particularly important here are pages 116-118 which define the syntax of the language. Sections 13 and 14 and Appendix F pertain only to the 6000-3.4 implementation of Pascal.

Pascal Report

By Niklaus Wirth, this document is bound with the *User Manual*. It is the guiding reference for implementors and the fundamental definition of the language. Some programmers find this report too concise to be of practical use, preferring the *User Manual* as a reference.

Books on Pascal

Several good books which teach Pascal or use it as a medium are available. The books by Wirth *Systematic Programming* and *Algorithms + Data Structures = Programs* use Pascal as a vehicle for teaching programming and data structure concepts respectively. They are both recommended. Other books on Pascal are listed in the references below.

1.6. Pascal documents — The Berkeley Implementation

This section describes the documentation which is available describing the Berkeley implementation of Pascal.

User's Manual

The document you are reading is the *User's Manual* for Berkeley Pascal. We often refer the reader to the Jensen-Wirth *User Manual* mentioned above, a different document with a similar name.

Manual sections

The sections relating to Pascal in the *UNIX Programmer's Manual* are *pix* (1), *pi* (1), *px* (1), *pxp* (1), and *pxref* (1). These sections give a description of each program, summarize the available options, indicate files used by the program, give basic information on the diagnostics produced and include a list of known bugs.

Implementation notes

For those interested in the internal organization of the Berkeley Pascal system there are a series of *Implementation Notes* describing these details. The *Berkeley Pascal PXP Implementation Notes* describe the Pascal interpreter *px*; and the *Berkeley Pascal PX Implementation Notes* describe the structure of the execution profiler *pxp*.

1.7. References

UNIX Documents

Communicating With UNIX
Computer Center
University of California, Berkeley
January, 1978.

Edit: a tutorial
Ricki Blau and James Joyce
Computing Services Division, Computing Affairs
University of California, Berkeley
January, 1978.

Ex/edit Command Summary
Computer Center
University of California, Berkeley
August, 1978.

Ex Reference Manual — Version 3.1
An Introduction to Display Editing with Vi
Vi Quick Reference
William Joy
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
April, 1979.

An Introduction to the C shell
William Joy
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
January, 1979.

Brian W. Kernighan
UNIX for Beginners — Second Edition
Bell Laboratories
Murray Hill, New Jersey.

Brian W. Kernighan
A Tutorial Introduction to the UNIX Text Editor
Bell Laboratories
Murray Hill, New Jersey.

Dennis M. Ritchie and Ken Thompson
The UNIX Time Sharing System
Communications of the ACM
July 1974
365-378.

B. W. Kernighan and M. D. McIlroy
UNIX Programmer's Manual — Seventh Edition
Bell Laboratories
Murray Hill, New Jersey
December, 1978.

Pascal Language Documents

Conway, Gries and Zimmerman
A Primer on PASCAL
Winthrop, Cambridge Mass.
1976, 433 pp.

Kathleen Jensen and Niklaus Wirth
Pascal — User Manual and Report
Springer-Verlag, New York.
1975, 167 pp.

C. A. G. Webster
Introduction to Pascal
Heyden and Son, New York
1976, 129pp.

Niklaus Wirth
Algorithms + Data structures = Programs
Prentice-Hall, New York.
1976, 366 pp.

Niklaus Wirth
Systematic Programming
Prentice-Hall, New York.
1973, 169 pp.

Berkeley Pascal documents

The following documents are available from the Computer Center Library at the University of California, Berkeley.

William N. Joy, Susan L. Graham, and Charles B. Haley
Berkeley Pascal User's Manual — Version 1.1
April, 1979.

William N. Joy
Berkeley Pascal PX Implementation Notes
Version 1.1, April 1979.
(Vax-11 Version By Kirk McKusick, December, 1979)

William N. Joy
Berkeley Pascal PXP Implemetation Notex
Version 1.1, April 1979.

2. Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the text editor *ex* (1). Users of the text editor *ed* should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because it allows us to make clearer examples.† The new UNIX† user will find it helpful to read one of the text editor documents described in section 1.4 before continuing with this section.

2.1. A first program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the documents *Communicating with UNIX* and *UNIX for Beginners*.

Once we are logged in we need to choose a name for our program; let us call it 'first' as this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence '.p' so we will call our file 'first.p'.

A sample editing session to create this file would begin:

```
% ex first.p
"first.p" No such file or directory
:
```

We didn't expect the file to exist, so the error diagnostic doesn't bother us. The editor now knows the name of the file we are creating. The ':' prompt indicates that it is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
program first(output)
begin
        writeln('Hello, world!')
end.
.
:
```

The line containing the single '.' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As the editor operates in a temporary work space we must now store the contents of this work space in the file 'first.p' so we can use the Pascal translator and executor *pix* on it.

```
:write
"first.p" 4 lines, 59 characters
:quit
%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

---

† Users with CRT terminals should find the editor *vi* more pleasant to use; we do not show its use here because its display oriented nature makes it difficult to illustrate.

†UNIX is a Trademark of Bell Laboratories.

‡ Our examples here assume you are using *csh*.

We are ready to try to translate and execute our program.

```
% pix first.p
    2  begin
e  ----↑--- Inserted ';'
Execution begins...
Hello, world!
Execution terminated
1 statement executed in 0.04 seconds cpu time
%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we will see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

```
% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:!pi %
!pi first.p
!
:quit
%
```

†The standard Pascal warnings occur only when the associated **s** translator option is enabled. The **s** option is discussed in sections 5.1 and A.6 below. Warning diagnostics are discussed at the end of section 3.2, the associated **w** option is described in section 5.2.

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

The first command issued from *ex* with the '!' involved the use of the '%' character which stands in this command for the file we are editing. *Ex* made this substitution, and then echoed back the expanded line before executing the command. When the command finished, the editor echoed the character '!' so that we would know it was done.

If we now use the UNIX *ls* list files command we can see what files we have:

```
% ls
first.p
obj
%
```

The file 'obj' here contains the Pascal interpreter code. We can execute this by typing:

```
% px obj
Hello, world!
1 statement executed in 0.02 seconds cpu time
%
```

Alternatively, the command:

```
% obj
```

will have the same effect. Some examples of different ways to execute the program follow.

```
% px
Hello, world!
1 statement executed in 0.02 seconds cpu time
% pi −p first.p
% px obj
Hello, world!
% pix −p first.p
Hello, world!
%
```

Note that *px* will assume that 'obj' is the file we wish to execute if we don't tell it otherwise. The last two translations use the **−p** no-post-mortem option to eliminate execution statistics and 'Execution begins' and 'Execution terminated' messages. See section 5.2 for more details. If we now look at the files in our directory we will see:

```
% ls
first.p
obj
%
```

We can give our object program a name other than 'obj' by using the move command *mv* (1). Thus to name our program 'hello':

```
% mv obj hello
% hello
Hello, world!
% ls
first.p
hello
%
```

Finally we can get rid of the Pascal object code by using the
rm
(1) remove file command, e.g.:

```
% rm hello
% ls
first.p
%
```

For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

2.2. A larger program

Suppose that we have used the editor to put a larger program in the file 'bigger.p'. We can list this program with line numbers by using the program *num* i.e.:

```
% num bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;  (* 1/16, 16 lines for interval [x, x+1] *)
     8          s = 32;      (* 32 character width for interval [x, x+1]
     9          h = 34;      (* Character position of x-axis *)
    10          c = 6.28138;  (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim begin
    17                  x := d / i;
    18                  y := exp(-x9 * sin(i * x);
    19                  n := Round(s * y) + h;
    20                  repeat
    21                          write(' ');
    22                          n := n - 1
    23                  writeln('*')
    24  end.
%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual.* A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
% pix bigger.p
   9        h = 34;      (* Character position of x-axis *)
w ----------------------------↑------ (* in a (* ... *) comment
  16        for i := 0 to lim begin
e --------------------------------↑------ Inserted keyword do
  18              y := exp(-x9 * sin(i * x));
E --------------------------------↑------ Undefined variable
e ------------------------------------------------↑------ Inserted ')'
  19              n := Round(s * y) + h;
E -------------------------↑------ Undefined function
E --------------------------------------------↑------ Undefined variable
  23              writeln('*')
e -------------------↑------ Inserted ';'
  24 end.
E -----↑------ Expected keyword until
e ---------↑------ Inserted keyword end matching begin on line 15
In program graph1:
  w - constant c is never used
  E - x9 undefined on line 18
  E - Round undefined on line 19
  E - h undefined on line 19
Execution suppressed due to compilation errors
%
```

Since there were fatal 'E' errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

% pi −l bigger.p

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

% pi −l bigger.p | lpr

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

2.3. Correcting the first errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was 'malformed'. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in 'standard' Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing '*)' of the comment on line 8. We can begin an editor session to

correct this problem by doing:

```
% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ *)
      s = 32;      (* 32 character width for interval [x, x+1] *)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
      for i := 0 to lim begin
:s/begin/do &
      for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper- and lower-case. On UNIX lower-case is preferred†, and all keywords and built-in **procedure** and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

†One good reason for using lower-case is that it is easier to type.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
            y := exp(-x) * sin(i * x);
:+s/Round/round
            n := round(s * y) + h;
:/write
                  write(' ');
:/
            writeln('*')
:insert
            until n = 0;
.
:$
end.
:insert
      end
.
:
```

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to *sin*, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

```
:?i ?s//c /
            y := exp(-x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:!pi %
!pi bigger.p
!
:quit
%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced. Thus these warning diagnostics may not appear in a program with bad syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
% number bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;  (* 1/16, 16 lines for interval [x, x+1] *)
     8          s = 32;      (* 32 character width for interval [x, x+1] *)
     9          h = 34;      (* Character position of x-axis *)
    10          c = 6.28138; (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim do begin
    17                  x := d / i;
    18                  y := exp(-x) * sin(c * x);
    19                  n := round(s * y) + h;
    20                  repeat
    21                          write(' ');
    22                          n := n - 1
    23                  until n = 0;
    24                  writeln('*')
    25          end
    26  end.
%
```

2.4. Executing the second example

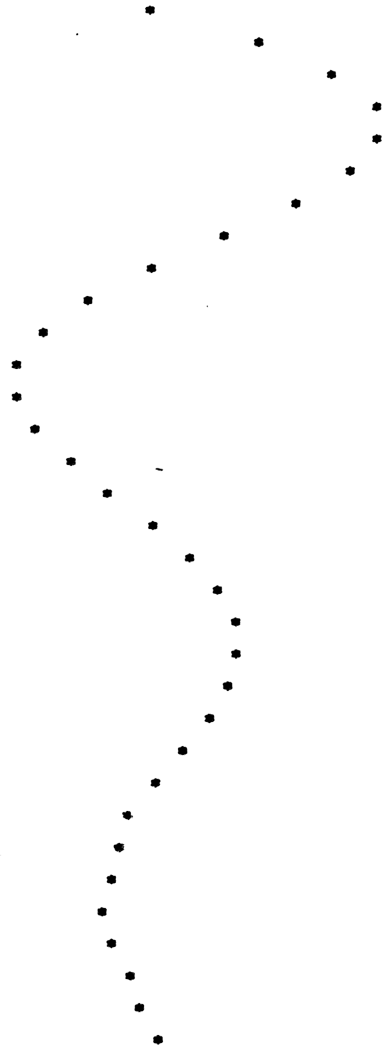
We are now ready to execute the second example. The following output was produced by our first run.

```
% px
Execution begins...
Floating divide by zero

        Error at "graph1"+2 near line 17

Execution terminated abnormally
2 statements executed in 0.04 seconds cpu time
%
```

Here the interpreter is presenting us with a runtime error diagnostic. It detected a 'division by zero' at line 17. Examining line 17, we see that we have written the statement 'x := d / i'

instead of 'x := d * i'. We can correct this and rerun the program:

```
% ex bigger.p
"bigger.p" 26 lines, 538 characters
:17
        x := d / i
:s'/'*
        x := d * i
:write
"bigger.p" 26 lines, 538 characters
:!pix %
!pix bigger.p
Execution begins...
```



```
Execution terminated
2550 statements executed in 0.21 seconds cpu time
!
:q
%
```

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
% px > graph
```

We can use *cat* (1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file, e.g.

```
% px | lpr
Execution begins...
Execution terminated
2550 statements executed in 0.31 seconds cpu time
%
```

Note here that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '|&' to the shell rather than '|', i.e.:

```
% px |& lpr
%
```

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

```
% pi −p bigger.p
% px | lpr
%
```

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

2.5. Formatting the program listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. See also section 5.2 for details on the **n** and **i** options of *pi.*

2.6. Execution profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program which was abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a re-examination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source level optimization.

An example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **z** option to *pix.* This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times

each statement in the program was executed.† When execution of the program completes, either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

```
% pix -l -z primes.p
Berkeley Pascal PI -- Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  primes.p

     1  program primes(output);
     2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
     3  var i,k,x,inc,lim,square,l: integer;
     4       prim: boolean;
     5       p,v: array[1..n1] of integer;
     6  begin
     7      write(2:6, 3:6); l := 2;
     8      x := 1; inc := 4; lim := 1; square := 9;
     9      for i := 3 to n do
    10      begin (*find next prime*)
    11          repeat x := x + inc; inc := 6-inc;
    12              if square <= x then
    13                  begin lim := lim+1;
    14                      v[lim] := square; square := sqr(p[lim+1])
    15                  end ;
    16              k := 2; prim := true;
    17              while prim and (k<lim) do
    18              begin k := k+1;
    19                  if v[k] < x then v[k] := v[k] + 2*p[k];
    20                  prim := x <> v[k]
    21              end
    22          until prim;
    23          if i <= n1 then p[i] := x;
    24          write(x:6); l := l+1;
    25          if l = 10 then
    26              begin writeln; l := 0
    27              end
    28      end ;
    29      writeln;
    30  end .
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated
```

†The counts are completely accurate only in the absence of runtime errors and nonlocal goto statements. This is not generally a problem, however, as in structured programs nonlocal goto statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* described below to get a count passes a suspended call point.

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc* (1). See *prof* (1) for a discussion of the C compiler profiling facilities.

```
1404  statements executed in 0.16 seconds cpu time
%
```

Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to *version stamp* the input program. *Pxp* also indicates the time at which the profile data was gathered.

```
% pxp -z primes.p
Berkeley Pascal PXP -- Version 1.1 (November 6, 1978)

Sat Mar 31 11:50 1979  primes.p

Profiled Sat Mar 31 13:02 1979

     1           1.-----|program primes(output);
     2                  |const
     2                  |   n = 50;
     2                  |   n1 = 7; (*n1 = sqrt(n)*)
     3                  |var
     3                  |   i, k, x, inc, lim, square, l: integer;
     4                  |   prim: boolean;
     5                  |   p, v: array [1..n1] of integer;
     6                  |begin
     7                  |   write(2: 6, 3: 6);
     7                  |   l := 2;
     8                  |   x := 1;
     8                  |   inc := 4;
     8                  |   lim := 1;
     8                  |   square := 9;
     9                  |   for i := 3 to n do begin (*find next prime*)
     9            48.-----|      repeat
    11               76.-----|      x := x + inc;
    11                       |      inc := 6 - inc;
    12                       |      if square <= x then begin
    13                  5.-----|         lim := lim + 1;
    14                         |         v[lim] := square;
    14                         |         square := sqr(p[lim + 1])
    14                       |      end;
    16                       |      k := 2;
    16                       |      prim := true;
    17                       |      while prim and (k < lim) do begin
    18               157.-----|         k := k + 1;
    19                         |         if v[k] < x then
    19                   42.-----|            v[k] := v[k] + 2 * p[k];
    20                         |         prim := x <> v[k]
    20                       |      end
    20                       |until prim;
    23                 |      if i <= n1 then
    23               5.-----|      p[i] := x;
    24                 |      write(x: 6);
    24                 |      l := l + 1;
```

```
25                    |        ...hen begin
26                             ...eln;
26                             ... 0
26                    |    ...
26               |  end;
29               |  writ...n
29               end.
%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not labelled, we look up in the listing to find the first '|' which directly above the original one which has a count and that is the answer. Thus, in our example, *k* was incremented 157 times on line 18, while the *write* procedure call on line 24 was executed 48 times as given by the count on the **repeat**.

More information on *pxp* can be found in its manual section *pxp* (1) and in sections 5.4, 5.5 and 5.10.

3. Error diagnostics

This section of the *User's Manual* discusses the error diagnostics of the programs *pi* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and section 5.2 below for more details.

3.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

Illegal characters

Characters such as '$', '!', and '@' are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be 'illegal characters'. This can happen if you leave off an opening string quote '''. Note that the character '"', although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character '?' so that they will show in your listing.

String errors

There is no character string of length 0 in Pascal. Consequently the input '''' is not acceptable. Similarly, encountering an end-of-line after an opening string quote ''' without encountering the matching closing quote yields the diagnostic "Unmatched ' for string". It is permissible to use the character '#' instead of ''' to delimit character and constant strings for portability reasons. For this reason, a spuriously placed '#' sometimes causes the diagnostic about unbalanced quotes. Similarly, a '#' in column one is used when preparing programs which are to be kept in multiple files. See section 5.9 for details.

Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments — those delimited by '{' and '}', and those delimited by '(*' and '*)'. Thus consider:

```
{ This is a comment enclosing a piece of program
a := functioncall;      (* comment within comment *)
procedurecall;
lhs := rhs;      (* another comment *)
}
```

By using one kind of comment exclusively in your program you can use the other delimiters when you need to "comment out" parts of your program†. In this way you will also allow the translator to help by detecting statements accidentally placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of "QUIT" below.

†If you wish to transport your program, especially to the 6000-3.4 implementation, you should use the character sequence '(*' to delimit comments. For transportation over the *rcslink* to Pascal 6000-3.4, the character '#' should be used to delimit characters and constant strings.

Digits in numbers

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

```
     4  r := 0.;
e  ------------↑------ Digits required after decimal point
     5  r := .0;
e  ----------↑------ Digits required before decimal point
     6  r := 1.e10;
e  ------------↑------ Digits required after decimal point
     7  r := .05e-10;
e  ----------↑------ Digits required before decimal point
```

These same constructs are also illegal as input to the Pascal interpreter *px*.

Replacements, insertions, and deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
% pix -l synerr.p
Berkeley Pascal PI -- Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  synerr.p

     1  progran syn(output);
e  ----↑--- Replaced identifier with a keyword program
     2  var i, j are integer;
e  ------------↑--- Replaced identifier with a ':'
     3  begin
     4          for j :* 1 to 20 begin
e  ------------------↑--- Replaced '*' with a '='
e  -----------------------------↑--- Inserted keyword do
     5                  write(j);
     6                  i = 2 ** j;
e  --------------------↑--- Inserted ':'
E  -------------------------↑--- Inserted identifier
     7                  writeln(i))
E  ----------------------------↑--- Deleted ')'
     8          end
     9  end.
%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that, if you assume that the language has a feature which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be summarized at the end of the containing **procedure** or **function** or at the end of the **program** if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, e.g. if a **type** identifier is used in an assignment statement, or if a simple variable is used where a **record** variable is required, a diagnostic will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print a diagnostic which indicates that the given symbol was 'Expected'.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates that the input is "malformed". If necessary, the translator may then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
% pix -l synerr2.p
Berkeley Pascal PI -- Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  synerr2.p

    1  program synerr2(input,outpu);
    2  integer a(10)
E  ----↑------ Malformed declaration
    3  begin
    4          read(b);
E  ----------------↑------ Undefined variable
    5          for c := 1 to 10 do
E  --------------↑------ Undefined variable
    6                  a(c) := b * c;
E  -----------------↑------ Undefined procedure
E  ------------------------↑------ Malformed statement
    7  end.
E 1 - File outpu listed in program statement but not declared
e 1 - The file output must appear in the program statement file list
In program synerr2:
  E - a undefined on line 6
  E - b undefined on line 4
  E - c undefined on lines 5 6
Execution suppressed due to compilation errors
%
```

Here we misspelled *input* and gave a FORTRAN style variable declaration which the translator diagnosed as a 'Malformed declaration'. When, on line 6, we used '(' and ')' for subscripting (as in FORTRAN) rather than the '[' and ']' which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

Expected and unexpected end-of-file, "QUIT"

If the translator finds a complete program, but there is more non-comment text in the input file, then it will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **ends** are present in the input. The message may appear after the recovery says that it "Expected '.'" since '.' is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by "QUIT". One example of this was given above — a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
% pix -l mism.p
Berkeley Pascal PI — Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  mism.p

    1  program mismatch(output)
    2  begin
e  ----↑------ Inserted ';'
    3          writeln('***');
    4          { The next line is the last line in the file }
    5          writeln
E  -------------------↑------ Unexpected end-of-file - QUIT
%
```

3.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line in the text of the program as well as the text of the error message. While this number is most often the line where the error occurred, it is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the **end** or **until**. to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary to some users. This is the only notable exception, however, to reasonable associations.

Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called 'type clashes' by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type 'classes':

array	Boolean	char	file	integer
pointer	real	record	scalar	string

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

```
E 7 – Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two line error message. If the same error occurs more than once, the same explanatory diagnostic will be given each time.

Scalar

The only class whose meaning is not self-explanatory is 'scalar'. Scalar has a precise meaning in the Jensen-Wirth *User Manual* where, in fact, it refers to *char*, *integer*, *real*, and *Boolean* types as well as the enumerated types. For the purposes of the Pascal translator, scalar in an error message refers to a user-defined, enumerated type, such as *ops* in the example above or *color* in

type color = (red, green, blue)

For integers, the more explicit denotation *integer* is used. Although it would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.

Function and procedure type errors

For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message similar to:

```
E 12 – sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
E 12 – sin's argument must be integer or real, not char
```

is produced. A few functions and procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.

Can't read and write scalars, etc.

The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that if you define

type color = (red, green, blue)

the translator does not associate these constants with the strings 'red', 'green', and 'blue' in any way. If you wish such an association to be made, you will have to write a routine to make it. Note, in particular, that you can only read characters, integers and real numbers from text files. You cannot read strings or Booleans. It is possible to make a

file of color

but the representation is binary rather than string.

Expression diagnostics

The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

```
% pi -l expr.p
Berkeley Pascal PI -- Version 1.1 (January 4, 1979)
```

```
Sat Mar 31 11:50 1979  expr.p

     1  program x(output);
     2  var
     3          a: set of char;
     4          b: Boolean;
     5          c: (red, green, blue);
     6          p: ↑ integer;
     7          A: alfa;
     8          B: packed array [1..5] of char;
     9  begin
    10          b := true;
    11          c := red;
    12          new(p);
    13          a := [];
    14          A := 'Hello, yellow';
    15          b := a and b;
    16          a := a * 3;
    17          if input < 2 then writeln('boo');
    18          if p <= 2 then writeln('sure nuff');
    19          if A = B then writeln('same');
    20          if c = true then writeln('hue''s and color''s')
    21   end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
e 20 - Input is used but not defined in the program statement
In program x:
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
%
```

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

Type equivalence

Several diagnostics produced by the Pascal translator complain about 'non-equivalent types'. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same type identifier. Thus, the variables *x* and *y* declared as

```
var
    x: ↑ integer;
    y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

```
E 7 - Type clash: non-identical pointer types
```

... Type of expression clashed with type of variable in assignment

Thus it is always necessary to declare a type such as

type intptr = ↑ integer;

and use it to declare

var x: intptr; y: intptr;

Note that if we had initially declared

var x, y: ↑ integer;

then the assignment statement would have worked. The statement

x↑ := y↑

is allowed in either case. Since the parameter to a **procedure** or **function** must be declared with a type identifier rather than a constructed type, it is always necessary, in practice, to declare any type which will be used in this way.

Unreachable statements

Berkeley Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement:

```
if false then
   writeln('impossible!')
```

Goto's into structured statements

The translator detects and complains about **goto** statements which transfer control into structured statements (**for**, **while**, etc.) It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. Such checks are made only within the body of a single procedure or function.

Unused variables, never set variables

Although Berkeley Pascal always clears variables to 0 at **procedure** and **function** entry, it is **not** good programming practice to rely on this initialization. To discourage this practice, and to help detect errors in program logic, *pi* flags as a 'w' warning error:

1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.

In fact, these diagnostics are applied to all declared items. Thus a **constant** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see sections 5.1 and 5.2.

3.3. Translator panics, i/o errors

Panics

One class of error which rarely occurs, but which causes termination of all processing when it does is a panic. A panic indicates a translator-detected internal inconsistency. A typical panic message is:

snark (rvalue) line=110 yyline=109

Snark in pi

If you receive such a message, the translation will be quickly and perhaps ungracefully terminated. You should contact a teaching assistant or a member of the system staff, after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. You should report the problem in any case. Pascal system bugs cannot be fixed unless they are reported.

Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up the full available process space of 64000 bytes on the PDP-11. On the VAX-11, table sizes are extremely generous and very large (10000) line programs have been easily accomodated. For the PDP−11, it is generally true that the size of the largest translatable program is directly related to **procedure** and **function** size. A number of non-trivial Pascal programs, including some with more than 2000 lines and 2500 statements have been translated and interpreted using Berkeley Pascal on PDP-11's. Notable among these are the Pascal-S interpreter, a large set of programs for automated generation of code generators, and a general context-free parsing program which has been used to parse sentences with a grammar for a superset of English.

If you receive an out of space message from the translator during translation of a large **procedure** or **function** or one containing a large number of string constants you may yet be able to translate your program if you break this one **procedure** or **function** into several routines.

I/O errors

Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.

3.4. Run-time errors

We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (1).

Start-up errors

These errors occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or being inaccessible to the user.

Program execution errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite loop, or running out of memory‡. The interpreter will produce a backtrace after the error occurs, showing all the active routine calls, unless the **p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.*

---

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image when it runs out of memory. This situation occurs very rarely.

* On the VAX-11, each stack frame, and each variable allocated with **new** is restricted to allocate at most 32000 bytes of storage (this is a PDP-11ism that has survived to the VAX.) The compiled version of the system, which is currently under development, will remove this restriction.

As an example of such an error, assume that we have accidentally declared the constant *n1* to be 6, instead of 7 on line 2 of the program primes as given in section 2.6 above. If we run this program we get the following response.

```
% pix primes.p
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167Subscript out of range

          Error at "primes"+8 near line 14

Execution terminated abnormally
941  statements executed in 0.12 seconds cpu time
%
```

Here the interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program primes.

Interrupts

If the program is interrupted while executing and the **p** option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the program was translated with the **z** option enabled to *pi* or *pix*.

I/O interaction errors

The final class of interpreter errors results from inappropriate interactions with files, including the user's terminal. Included here are bad formats for integer and real numbers (such as no digits after the decimal point) when reading.

Panics

A small number of panics are possible with *px*. These should be reported to a teaching assistant or to the system staff if they occur.

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a procedure or function entry or exit. In this case, the backtrace will only contain the current line. A reverse call order list of procedures will not be given.

4. Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

4.1. Introduction

Our first sample programs, in section 2, used the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the following Pascal program which is similar to the program *cat* (1).

```
% pix -l kat.p <primes
Berkeley Pascal PI -- Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  kat.p

     1  program kat(input, output);
     2  var
     3      ch: char;
     4  begin
     5      while not eof do begin
     6          while not eoln do begin
     7              read(ch);
     8              write(ch)
     9          end;
    10          readln;
    11          writeln
    12      end
    13  end { kat }.
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated
925 statements executed in 0.21 seconds cpu time
%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our prime number program of section 2.6. It is also possible to 'pipe' input to this program much as we piped input to the line printer daemon *lpr* (1) before. Thus, the same output as above would be produced by

```
% cat primes | pix -l kat.p
```

All of these examples use the shell to control the input and output from files. One very simple way to associate Pascal files with named UNIX† files is to place the file name in the program statement. For example, suppose we have previously created the file *data*. We then use it as input to another version of a listing program.

†UNIX is a Trademark of Bell Laboratories.

```
% cat data
line one.
line two.
line three is the end.
% pix -l copydata.p
Berkeley Pascal PI — Version 1.1 (January 4, 1979)

Sat Mar 31 11:50 1979  copydata.p

     1  program copydata(data, output);
     2  var
     3       ch: char;
     4       data: text;
     5  begin
     6       reset(data);
     7       while not eof(data) do begin
     8            while not eoln(data) do begin
     9                 read(data, ch);
    10                 write(ch)
    11            end;
    12            readln(data);
    13            writeln
    14       end
    15  end { copydata }.
Execution begins...
line one.
line two.
line three is the end.
Execution terminated
134 statements executed in 0.01 seconds cpu time
%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we 'reset(data)', the Pascal system opens our file 'data' for reading. More sophisticated, but less portable, examples of using UNIX files will be given in sections 4.5 and 4.6. There is a portability problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

4.2. Eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of *eof* and *eoln*. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is "used up". In fact, there is no way the system can know whether the input is at the end-of-file or the end-of-line unless it attempts to read a line from it. If the input is from a previously created file, then this reading can take place without run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were a terminal, the user would have to type some input before execution could begin. This would make it impossible

†It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe*, the output of a program which is reading from the terminal.

for the program to begin by prompting for input or printing a herald.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file or end-of-line conditions are true. Thus, internally, these functions can have three values — true, false, and "I don't know yet; if you ask me I'll have to find out". All files remain in this last, indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, e.g. in a call to *read.* The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it will be necessary for it to attempt to read from the input.

Thus consider the following example code

```
while not eof do begin
   write('number, please? ');
   read(i);
   writeln('that was a ', i: 2)
end
```

At first glance, this may be appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait for the user to type a line. By producing the desired prompting before testing *eof,* the following code avoids this problem:

```
write('number, please ?');
while not eof do begin
   read(i);
   writeln('that was a ', i:2);
   write('number, please ?')
end
```

The user must still type a line before the **while** test is completed, but the prompt will ask for it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

4.3. More about eoln

To have a good understanding of when *eoln* will be true it is necessary to know that in any file there is a special character indicating end-of-line, and that, in effect, the Pascal system always reads one character ahead of the Pascal *read* commands.† For instance, in response to 'read(ch)', the system sets *ch* to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with

†In Pascal terms, 'read(ch)' corresponds to 'ch := input^; get(input)'

a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this will almost surely have the effect of ignoring the last character in the line. The 'read(ch)' belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a *readln* call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

4.4. Output buffering

A final point about Pascal input-output must be noted here. This concerns the buffering of the file *output*. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program such as the line printer daemon *lpr* (1). To gain efficiency, the Pascal system "buffers" the output characters (i.e. it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction.) However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) The procedure *message* or *flush* is called.

Thus, in the code sequence

```
for i := 1 to 5 do begin
    write(i: 2);
    Compute a lot with no output
end;
writeln
```

the output integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the **b** option to 0 before the **program** statement by inserting a comment of the form

```
(*$b0*)
```

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in section 5.

4.5. Files, reset, and rewrite

It is possible to use extended forms of the built-in functions *reset* and *rewrite* to get more general associations of UNIX file names with Pascal file variables. When a file other than *input* or *output* is to be read or written, then the reading or writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. As we saw in section 2.9, by mentioning the file in the **program** statement, we could cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the **program** statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *removed* by the Pascal system as soon as they become inaccessible. They are not removed, however, if a run-time error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in section 3.1 by doing:

```
reset(data, 'primes')
```

instead of a simple

```
reset(data)
```

In this case it is not essential to mention 'data' in the program statement, but it is still a good idea because is serves as an aid to program documentation. The second parameter to *reset* and *rewrite* may be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at run time. Full details on file name/file variable associations are given in section A.3.

4.6. Argc and argv

Each UNIX process receives a variable length sequence of arguments each of which is a variable length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*−1, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
% obj /etc/motd /usr/dict/words hello
```

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be 'obj', the first '/etc/motd', etc.

Pascal does not provide variable size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure and has the syntax

```
argv(i, a)
```

where *i* is an integer and *a* is a string variable. This procedure call assigns the (possibly truncated or blank padded) *i*'th argument of the current process to the string variable *a*. The file manipulation routines *reset* and *rewrite* will strip trailing blanks from their optional second

arguments so that this blank padding is not a problem in the usual case where the arguments are file names.

We are now ready to give a Berkeley Pascal program 'kat', based on that given in section 3.1 above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
% cat kat.p
program kat(input, output);
var
   ch: char;
   i: integer;
   name: packed array [1..100] of char;
begin
   i := 1;
   repeat
      if i < argc then begin
         argv(i, name);
         reset(input, name);
         i := i + 1
      end;
      while not eof do begin
         while not eoln do begin
            read(ch);
            write(ch)
         end;
         readln;
         writeln
      end
   until i >= argc
end { kat }.
%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed on other systems. As this program deals mostly with *argc* and *argv* and UNIX system dependent considerations, portability is of little concern.

If this program is in the file 'kat.p', then we can do

```
% pi kat.p
% mv obj kat
% kat primes
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

930 statements executed in 0.24 seconds cpu time
% kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)
287 statements executed in 0.02 seconds cpu time
%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat*, copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
% kat < primes
```

now equivalent to

```
% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
% kat xxxxqqq
xxxxqqq: No such file or directory

        Error at "kat"+5 near line 11

4 statements executed in 0.01 seconds cpu time
%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
% pi -pb kat.p
% mv obj kat
```

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the **t** option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
% kat xxxxqqq
xxxxqqq: No such file or directory
% kat primes
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

5. Details on the components of the system

5.1. Options

The programs *pi* and *pxp* take a number of options.† There is a standard UNIX† convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted of the character '−' followed by a single character option name. In fact, it is possible to place more than one option letter after a single '−', thus

% pi −lz foo.p

and

% pi −l −z foo.p

are equivalent.

There are 26 options, one corresponding to each lower case letter. Except for the **b** option which takes a single digit value, each option may be set on (enabled) or off (disabled.) When an on/off valued option appears on the command line of *pi* or it inverts the default setting of that option. Thus

% pi −l foo.p

enables the listing option **l**, since it defaults off, while

% pi −t foo.p

disables the run time tests option **t**, since it defaults on.

In additon to inverting the default settings of *pi* options on the command line, it is also possible to control the *pi* options within the body of the program by using comments of a special form illustrated by

{$l−}

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters‡. The most basic actions for options are to set them, thus

{$l+ Enable listing}

or to clear them

{$t−,p− No run−time tests, no post mortem analysis}

Notice that '+' always enables an option and '−' always disables it, no matter what the default is. Thus '−' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

---

†As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

†UNIX is a Trademark of Bell Laboratories.

‡This format was chosen because it is used by Pascal 6000-3.4. In general the options common to both implementations are controlled in the same way so that comment control in options is mostly portable. It is recommended, however, that only one control be put per comment for maximum portability, as the Pascal 6000-3.4 implementation will ignore controls after the first one which it does not recognize.

5.2. Pi (and pix)

We now summarize the options of *pi.* These options may also be specified on the command line to *pix* before the name of the file to be translated. Arguments to *pix* after the name of the file to be translated are passed to the executed program run by *px.* With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

Buffering of the file output — b

The **b** option controls the buffering of the file *output.* The default is line buffering, with flushing at each reference to the file *input* and under certain other circumstances detailed in section 5 below. Mentioning **b** on the command line, e.g.

% pi —b assembler.p

causes standard output to be block buffered, where a block is 512 characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, e.g.

{$b0}

causes the file *output* to be unbuffered. Any value 2 or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **b** must precede the **program** statement.

Include file listing — i

The **i** option takes a list of **include** files, **procedure** and **function** names and causes these portions of the program to be listed while translating†. All arguments after the **—i** flag up to the name of the file being translated, which ends in '.p', are in this list. Typical uses would be

% pix —i scanner.i compiler.p

to make a listing of the routines in the file scanner.i, and

% pix —i scanner compiler.p

to make a listing of only the routine *scanner.* This option is especially useful for conservation-minded programmers making partial program listings.

Make a listing — l

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The **l** option is pushed and popped by the **i** option at appropriate points in the program.

Eject new pages for include files — n

The **n** option causes *pi* to eject a new page in the listing and print a header line at **include** file boundaries, providing automatic pagination control. To have effect, either the **l** or **i** option should also be specified, or the input should contain listing control in comments. An example would be

% pi —in scan.i c.p

---

†Include files are discussed in section 5.9.

Post-mortem dump — p

The **p** option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. It also cause *px* to count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying **p** on the command line disables these checks and the ability to give this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the **p** option in comments. To prevent the post-mortem backtrace on error, **p** must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

% pi —pbt pxref.p

Standard Pascal only — s

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as 's' warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedures** and **functions**, extensions to the **procedure** *write*, and the padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters so that the case of keywords and identifiers is effectively ignored. The **s** option is most useful when a program is to be transported, thus

% pi —s isitstd.p

Runtime tests — t

The **t** option controls the generation of tests that subrange variable values are within bounds at run time. By default these tests are generated. If the **t** option is specified on the command line, or in a comment which turns it off, then the tests are not generated. Thus the first line of a program to run without tests might be

{$t— No runtime tests}

Disabling runtime tests also causes **assert** statements to be treated as comments.†

Card image, 72 column input — u

Turning the **u** option on, either on the command line or in a comment causes the input to be treated as card images with sequence numbers and truncated to 72 columns. Thus

% pix —u cards.p

Suppress warning diagnostics — w

The **w** option, which defaults on, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

{$w—}

or on the command line

% pi —w tryme.p

suppresses these usually useful diagnostics.

†See section A.1 for a description of **assert** statements.

Generate counters for an execution profile — z

The **z** option, which defaults off, enables the production of execution profiles. By specifying **z** on the command line, i.e.

% pi −z foo.p

or by enabling it in a comment before the **program** statement we cause *pi* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in section 2.6; its options are described in section 5.5.

5.3. Px

The first argument to *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc* and *argv* as described in section 4.6.

Px may also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

% obj primes

will be converted to read

% px obj primes

5.4. Pxp

Pxp takes, on its command line, a list of options followed by the program file name, which must end in '.p' as it must for *pi* and *pix*. *pxp* will produce an execution profile if any of the **z t** or **c** options are specified on the command line. If none of these options are specified, then *pxp* functions as a program reformatter. See section 5.5 for more details.

It is important to note that only the **z** option of *pxp*, and the **n**, **u**, and **w**, options, which are common to *pi* and *pxp* can be controlled in comments. All other options must be specified on the command line to have any effect.

The following options are relevant to profiling with *pxp*:

Include the bodies of all routines in the profile — a

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

Extract profile data from the file core — c

This option causes *pxp* to extract the data from the file *core* in the current directory. This is used in debugging the Pascal system, and should not normally be needed. When an abnormal termination occurs in *px* it writes the data to the file *pmon.out*. The **z** option enables profiling with data from this file.

Suppress declaration parts from a profile — d

Normally a profile includes declaration parts. Specifying **d** on the command line suppresses declaration parts.

Eliminate include directives — e

Normally, *pxp* preserves **include** directives to the output when reformatting a program, as though they were comments. Specifying **−e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate **include** directives, e.g. before transporting a program.

Fully parenthesize expressions — f

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

d := a + b mod c / e

with minimal parenthesization, the default, will print as

d := a + ((b mod c) / e)

with the f option specified on the command line.

Left justify all procedures and functions — j

Normally, each **procedure** and **function** body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

Print a table summarizing procedure and function calls — t

The **t** option causes *pxp* to print a table summarizing the number of calls to each **procedure** and **function** in the program. It may be specified in combination with the **z** option, or separately.

Enable and control the profile — z

The **z** profile option is very similar to the **i** listing control option of *pi*. If **z** is specified on the command line, then all arguments up to the source file argument which ends in '.p' are taken to be the names of **procedures** and **functions** or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

% pxp —z test parser.i compiler.p

This specifies that profiles of the routines in the file *parser.i* and the routine *test* are to be made.

5.5. Formatting programs using pxp

The program *pxp* can be used to reformat programs, by using a command of the form

% pxp dirty.p > clean.p

Note that since the shell creates the output file 'clean.p' before *pxp* executes, so 'clean.p' and 'dirty.p' must not be the same file.

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

1) Left marginal comments, which begin in the first column of the input line and are placed in the first column of an output line.
2) Aligned comments, which are preceded by no input tokens on the input line. These are aligned in the output with the running program text.
3) Trailing comments, which are preceded in the input line by a token with no more than two spaces separating the token from the comment.
4) Right marginal comments, which are preceded in the input line by a token from which they are separated by at least three spaces or a tab. These are aligned down the right margin of the output, currently to the first tab stop after the 40th column from the current "left margin".

Consider the following program.

```
% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
  An aligned, multi–line comment
  which explains what this program is
  all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;        {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced.

```
% pxp comments.p
{ This is a left marginal comment. }

program hello(output);
var
    i: integer; {This is a trailing comment}
    j: integer;                                        {This is a right marginal comment}
    k: array [1..10] of array [1..10] of integer;     {Marginal, but past the margin}
{
  An aligned, multi–line comment
  which explains what this program is
  all about
}
begin
    i := 1; {Trailing i comment}
{A left marginal comment}
    {An aligned comment}
    j := 1;                                            {Right marginal comment}
    k[1] := 1;
    writeln(i, j, k[1])
end.
%
```

The following formatting related options are currently available in *pxp*. The options f and j described in the previous section may also be of interest.

Strip comments −s

The s option causes *pxp* to remove all comments from the input text.

Underline keywords — _

A command line argument of the form —_ as in

% pxp —_ dirty.p

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

Specify indenting unit — [23456789]

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form $-d$ with d a digit, $2 \leqslant d \leqslant 9$ you can specify that d spaces are to be used per level instead.

5.6. Pcc and carriage control

The UNIX system printer driver does not implement FORTRAN style carriage control. Thus the function *page* on UNIX does not output a character '1' in column 1 of a line, but rather puts out a form-feed character. For those who wish to use carriage control, the filter *pcc* is available which interprets this control. A sample usage is:

% px | pcc

or

% pix prog.p | pcc | lpr

for printer copy. *Pcc* is fully described by its manual documentation *pcc* (1).

5.7. Pxref

The cross-reference program *pxref* may be used to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file 'foo.p' one can execute the command:

% pxref foo.p

The cross-reference is, unfortunately, not block structured. Full details on *pxref* are given in its manual section *pxref* (1).

5.8. Pascals

A version of Wirth's subset Pascal translator *pascals* is available on UNIX. It was translated to interpreter code by *pi* and is invoked by a command of the form:

% pascals prog.p

The program in the file given is translated to interpretive code which is then immediately executed. *Pascals* is thus similar to *pix*. Only small programs can be handled. *Pascals* is most interesting to those wishing to study its error recovery techniques, which are described in Wirth's book *Algorithms + Data Structures = Programs*.

5.9. Multi-file programs

A text inclusion facility is available with Berkeley Pascal. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character '#' in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word 'include' followed by a filename enclosed in single ''' or double '"' quotation marks. The file name may be followed by a semicolon ';' if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '.i'. An example of the use of included files in a main program would be:

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"

begin
  { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translate- and run-time diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. Nested includes are possible up to 10 deep.

See the descriptions of the **i** and **n** options of *pi* in section 5.2 above; these can be used to control listing when **include** files are present.

Include control lines are never printed in a listing. If the **n** option is not set, they are replaced by a line containing the file name and a ':' character. This is the default setting. If the **n** new page option is enabled then the **include** line is replaced with a banner line similar to the first line of a listing. This line is placed on a new page in the listing.

When a non-trivial line is encountered in the source text after an **include** finishes, the 'popped' filename is printed, in the same manner as above.

For the purposes of error diagnostics when not making a listing, the filename will be printed before each diagnostic if the current filename has changed since the last filename was printed.

A. Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available. It concludes with a list of differences with the commonly available Pascal 6000−3.4 implementation, and some comments on standard and portable Pascal.

A.1. Extensions to the language Pascal

This section defines non-standard language constructs available in Berkeley Pascal. The s standard Pascal option of the translator *pi* can be used to detect these extensions in programs which are to be transported.

String padding

Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make them as long as is required. The following is a legal Berkeley Pascal program:

```
program x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
   z := 'red';
   writeln(z)
end;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red          '
```

which is standard Pascal.

Octal constants, octal and hexadecimal write

Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

```
write(a:n hex)
```

cause the internal representation of expression *a*, which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal respectively.

Assert statement

An **assert** statement causes a *Boolean* expression to be evaluated each time the statement is executed. A runtime error results if any of the expressions evaluates to be *false*. The assert statement is treated as a comment if run-time tests are disabled. The syntax for assert is:

```
assert <expr>
```

A.2. Resolution of the undefined specifications

File name — file variable associations

Each Pascal file variable is associated with a named UNIX† file. Except for *input* and *output*, which are exceptions to some of the rules, a name can become associated with a file in any of three ways:

1) If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.

2) If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.

3) If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form 'tmp.x' is associated with the file. Temporary names start with 'tmp.1' and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

The program statement

The syntax of the **program** statement is:

program <id> (<file id> { , <file id > }) ;

The file identifiers (other than *input* and *output*) must be declared as variables of file type in the global declaration part.

The files input and output

The formal parameters *input* and *output* are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:

1) The program heading **must** contains the formal parameter *output*. If *input* is used, explicitly or implicitly, then it must also be declared here.

2) Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

   var input, output: text

3) The procedure *reset* may be used on *input*. If no UNIX file name has ever been associated with *input*, and no file name is given, then an attempt will be made to 'rewind' *input*. If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

   rewrite(output)

   associates a temporary name with *output*.

Details for files

If a file other than *input* is to be read, then reading must be initiated by a call to the procedure *reset* which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

---

†UNIX is a Trademark of Bell Laboratories.

Buffering

The buffering for *output* is determined by the value of the **b** option at the end of the **program** statement. If it has its default value 1, then *output* is buffered in blocks of up to 512 characters, flushed whenever a writeln occurs and at each reference to the file *input*. If it has the value 0, *output* is unbuffered. Any value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are always buffered in blocks of 512 characters. All output buffers are flushed when the files are closed at scope exit, whenever the procedure *message* is called, and can be flushed using the built-in procedure *flush*.

An important point for an interactive implementation is the definition of 'input↑'. If *input* is a teletype, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt could be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, reading from a file occurring only when necessary.

The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols 'and', 'or', 'not', '<=', '>=', '<>', and the uparrow '↑' (for pointer qualification) are recognized.† Less portable are the synonyms tilde '~' for **not**, '&' for **and**, and '|' for **or**.

Upper and lower case are considered distinct. Keywords and built-in **procedure** and **function** names are composed of all lower case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword **goto**. The standard type 'boolean' is also available as 'Boolean'.

Character strings and constants may be delimited by the character ''' or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line — see *Multi-file programs* below.

The standard types

The standard type *integer* is conceptually defined as

type integer = minint .. maxint;

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

const maxint = 2147483647; minint = −2147483648;

The standard type *char* is conceptually defined as

type char = minchar .. maxchar;

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0, ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 17 digits of precision with numbers as small as 10 to the negative 38th power and as large as 10 to the 38th power.

Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The

†On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes.

restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

Option control

Options of the translator may be controlled in two distinct ways. A number of options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '−' and cause the default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

% pi −ls foo.p

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. The format for option control in comments is identical to that used in Pascal 6000−3.4. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

{$l+,s+ listing on, standard Pascal}

as the first line of the program. The 'l' option is more appropriately specified on the command line, since it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

Directives consist of a letter designating the option, followed either by a '+' to turn the option on, or by a '−' to turn the option off. The **b** option takes a single digit instead of a '+' or '−'.

Notes on the listings

The first page of a listing includes a banner line indicating the version and date of generation of *pi*. It also includes the UNIX path name supplied for the source file and the date of last modification of that file.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed character, control-l, which causes a page eject in the listing, and a line with no characters which causes the line number to be suppressed in the listing, creating a truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many assemblers. Non-printing characters are printed as the character '?' in the listing.†

Multi-file programs

It is also possible to prepare programs whose parts are placed in more than one file. The files other than the main one are called **include** files and have names ending with '.i'. The contents of an **include** file are referenced through a pseudo-statement of the form:

#include "file.i"

The '#' character must be the first character on the line. The file name may be delimited with '"' or '' characters. Nested **includes** are possible up to 10 deep. More details are given in sections 5.9 and 5.10.

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

The standard procedure write

If no minimum field length parameter is specified for a *write*, the following default values are assumed:

integer	10
real	22
Boolean	10
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be explicitly indicated by 'writeln(f)', where 'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

Files

Files cannot be members of files or members of dynamically allocated structures.

Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to −32768, and the upper bound less than 32768. In particular, strings may have any length from 1 to 32767 characters, and sets may contain no more than 32767 elements.

Line and symbol length

There is *no* intrinsic limit on the length of identifiers. Identifiers are considered to be distinct if they differ in any single position over their entire length. There is a limit, however, on the maximum input line length. This is quite generous however, currently exceeding 160 characters.

Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the hardware and thus, on the PDP-11, by the 16 bit address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

On the VAX-11, there is an implementation defined limit of 32000 bytes in a stack frame and in a variable allocated by **new**. This restriction will not exist in the compiled version of Pascal, which is currently under development.

Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

Additional predefined types

The type *alfa* is predefined as:

type alfa = **packed array** [1..10] **of char**

The type *intset* is predefined as:

type intset = **set of** 0..127

In most cases the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer†. In the latter case the type defaults to the current binding of *intset*, which must be "type set of (a subrange of) integer" at that point.

Note that if *intset* is redefined via:

type intset = **set of** 0..58;

then the default integer set is the implicit *intset* of Pascal 6000−3.4

Additional predefined operators

The relationals '<' and '>' of proper set inclusion are available. With *a* and *b* sets, note that

(**not** (*a* < *b*)) <> (*a* >= *b*)

As an example consider the sets *a* = [0,2] and *b* = [1]. The only relation true between these sets is '<>'.

Non-standard procedures

argv(i,a)	where *i* is an integer and *a* is a string variable assigns the (possibly truncated or blank padded) *i*'th argument of the invocation of the current UNIX process to the variable *a*. The range of valid *i* is *0* to *argc−1*.
date(a)	assigns the current date to the alfa variable *a* in the format 'dd mmm yy ', where 'mmm' is the first three characters of the month, i.e. 'Apr'.
flush(f)	writes the output buffered for Pascal file *f* into the associated UNIX file.
halt	terminates the execution of the program with a control flow backtrace.
linelimit(f,x)‡	with *f* a textfile and *x* an integer expression causes the program to be abnormally terminated if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in procedure *write*, to be written unbuffered on the diagnostic unit 2, almost always the user's terminal.

†The current translator makes a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16 bit arithmetic given above in this case.
‡Currently ignored by *px*.

null	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
remove(a)	where *a* is a string causes the UNIX file whose name is *a*, with trailing blanks eliminated, to be removed.
reset(f,a)	where *a* is a string causes the file whose name is *a* (with blanks trimmed) to be associated with *f* in addition to the normal function of *reset*.
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where *i* is an integer sets the statement limit to be *i* statements. Specifying the **p** option to *pc* disables statement limit counting.
time(a)	causes the current time in the form ' hh:mm:ss ' to be assigned to the alfa variable *a*.

Non-standard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x*, i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).
random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 536870912. The initial seed is 7774755.
seed(i)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i*.
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.

A.5. Remarks on standard and portable Pascal

It is occasionally desirable to prepare Pascal programs which will be acceptable at other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the non-portable usages. Wirth's *Pascal Report* concludes with a standard for implementation and program exchange.

In particular, the following differences may cause trouble when attempting to transport programs between this implementation and Pascal 6000−3.4. Using the s translator option may serve to indicate many problem areas.†

†The **s** option does not, however, check that identifiers differ in the first 8 characters. *Pi* also does not check the semantics of **packed**.

Features not available in Berkeley Pascal

Formal parameters which are **procedure** or **function**.

Segmented files and associated functions and procedures.

The function *trunc* with two arguments.

Arrays whose indices exceed the capacity of 16 bit arithmetic.

Features available in Berkeley Pascal but not in Pascal 6000-3.4

The procedures *reset* and *rewrite* with file names.

The functions *argc*, *seed*, *sysclock*, and *wallclock*.

The procedures *argv*, *flush*, and *remove*.

Message with arguments other than character strings.

Write with keyword **hex**.

The **assert** statement.

Other problem areas

Sets and strings are more general in Berkeley Pascal; see the restrictions given in the Jensen-Wirth *User Manual* for details on the 6000−3.4 restrictions.

The character set differences may cause problems, especially the use of the function *chr*, characters as arguments to *ord*, and comparisons of characters, since the character set ordering differs between the two machines.

The Pascal 6000−3.4 compiler uses a less strict notion of type equivalence. In Berkeley Pascal, types are considered identical only if they are represented by the same type identifier. Thus, in particular, unnamed types are unique to the variables/fields declared with them.

Pascal 6000−3.4 doesn't recognize our option flags, so it is wise to put the control of Berkeley Pascal options to the end of option lists or, better yet, restrict the option list length to one.

For Pascal 6000−3.4 the ordering of files in the program statement has significance. It is desirable to place *input* and *output* as the first two files in the **program** statement.

Network System Manual

Eric Schmidt

Introduction

This documentation should be read by people responsible for maintaining the network (and the systems it runs on). It is divided into the following sections:

- Maintaining the Network
- Setting up the Network
- Future Plans
- For Berkeley
- Bugs

Besides the commands described in the net introduction, there are a number of network-internal commands and statistics files.

Maintaining the Network

1. Check the network:
   a) See if the network daemons are running with the command

      % ps ax | grep net

      If not running, see below.

   b) Check the network queue to see how long commands have been waiting to be sent.
2. To restart the network daemons, try
   a) See if they are running, as above.
   b) If so, there should be two net daemon processes per machine connection -- "kill -9" the child named "netdaemon" and the parent "netstart" will start a new one.
   c) If there are no "netstart's" or "netdaemon's", executing

      % /usr/net/bin/start

      will start up all the daemons on your machine.

   d) To have two "netdaemons" pointing to the same machine is to invite disaster. What happens is that a few small requests get through, and then the error rate goes up by a factor of a hundred. The first thing to do when you see this is to check the number of net daemons.

   (All this must be done as super-user).
3. There are files /usr/net/plogfile? with a log for each directly-connected machine. Example:

   % tail /usr/net/plogfiley

   will tell you in a cryptic form what the network has done on the Cory machine. This is a good file to inspect to see if transmissions are failing, etc. It is readable only by "schmidt" and "root" (and "staff" on Cory).

Basically, "sends" begin "^S" and end "^T". If a send fails for some reason, "^F" is printed instead of "^T". "^R" is printed when a receive begins. "RCV" is entered when the command is received and executed. "^P" indicates a pass through.

The file /usr/net/netstat?, one per directly-connected machine, have various statistics about the usage of the network, and the system load.

4. The overloading on various machines is causing high error rates. If these rates persist, the network can overload to the point where the queues are immense and nothing gets through. The only thing that can be done at this point is to remove the files (using *netrm* as super-user) and adjust the delay times in the 'initfile'.

5. If free space is a scarce commodity, truncate logfile and plogfile?, and check /usr/net/send and /usr/net/rcv. If there are any files there which are quite old, use your judgement to remove them.

6. Net news should be provided periodically (usually in '/usr/help' or '/usr/news').

Setting up the Network

1. Hardware

For another machine to join the network, there must be some hardware link, such as tty lines, special character-oriented hardware, or DMA lines between the two machines. The software does not require the link to be reliable or fast. The best way to start is with slow-speed TTY lines (say 1200 baud) which demonstrate the network's usefulness at low cost. The highest transfer speed on a TTY link is about one-half the link speed (at best), because of processing time, the 3 — 4 character expansion from 8 bits to 6, and the responses.

2. Software

There is a subdirectory "src/net" with all the network source files and a "makefile". The file "READ_ME" has information about the different conditional compilation option available, and table entries which must be made in the '.c' files.

Assuming the options have been specified in the makefile, the command

```
% make all
```

will make all the necessary files. Then the command

```
% make install DESTDIR=
```

will install the user commands and service programs. The directories are specified as options in the makefile. Finally,

```
% make clean
```

removes all the '.o' and executable files.

There are also other little-used programs, made by "make othernet". Included are programs to send and receive packets and files, and a program to simulate TTY lines using pipes. It should not be necessary to run these.

3. Directories and Files.

The central directory is '/usr/net', which has subdirectories 'bin', 'rcv', and 'send?', where the '?' represents the one-letter codes of the directly-connected machines. For various reasons, the support programs *(netdaemon, netstart, nmail, nwrite,* etc.) must be in 'bin'. The user programs may be anywhere but the pathnames in "Paths.h" must be reset correctly.

The logfiles are 'logfile' and 'plogfile?', one for each directly-connected machine. If not present in '/usr/net', they are not created.

The file 'bin/start' should start up all the net daemons on the current machine. This

file is normally executed by '/etc/rc'. The file 'initfile' has a format similar to '.netrc' but is read by the net daemons when they are started. It has the network device names, speed and various tuning parameters. The complete list is in the source file 'sub.c'. It is generally possible to change almost anything about the network through the 'initfile' and restarting the daemon.

The program 'bin/netstart' is a simple program to start up a net daemon, and if it should abort for any reason, restart it.

There must be an account 'network', which executes all responses and the free commands. Its login directory should be '/usr/net/network' and login shell should be 'nsh' in that directory. The list of free commands can be changed in 'nsh.c'.

The 'cat' command must be in '/bin' (used by *netcp).*

At Berkeley, we follow the convention that the TTY special files are named '/dev/net-X', where 'X' is the remote machine name.

The mail program should be modified to recognize remote names and to fork a "sendmail" command. If possible, a '−r' option should be added (see mmail.c).

Future Plans

It is important to understand the scope of this network; what it is and what it is not. Since it is "batched", there are a lot of things it cannot do. Our experience is that remote file copying, mailing and printing between machines are adequate for our immediate needs.

In the future, we will concentrate on improving the hardware and speeding up the network, rather than major user-interface changes.

This is a list of the things that have been planned for the future for the network.

1. Use Bill Joy's retrofit library to simulate the version 7 system calls. This would reduce the dependence on conditional compilation for V6 code.
2. The file length restriction is a major inconvenience. One way to allow large files would be to send large files (over 100,000 chars) only when there are no smaller ones. This would be non-preemptive, but might be workable. Another way would be to have two hardware links, and two sets of daemons, one for large files and one for small ones.
3. Bob Fabry has suggested generalizing the machine name to be user-definable as a login/machine pair, to make it easier for people with multiple accounts on multiple machines.
4. It is possible to share binaries between all the similar machine configurations (e.g. the Comp. Center machines). This involves "patching" the local machine in the binary.
5. Ed Gould suggested that the notion of "default" machine was too restrictive— that an appropriate default for, say, "netlpr" was a nearby machine with a quality printer, whereas the default for "net" should be the logical most useful machine.
6. Security — I have just recently bullet-proofed the network so 'root' commands are very restricted. However, the presence of passwords in the '.netrc' files poses a hazard to other machines when one machine is broken into. As long as the root password is not in a file, the root is safe. I am fairly convinced there is no way using encryption to handle the '.netrc' files. The introductory documentation is very explicit about the threat these passwords pose.
7. Certain other more exotic requests are unlikely to get done until things change:
   a) Having the same user-id's across machines.
   b) Adding an option to "net" to wait until a response has been received.

c) There should be a net status command which would give things like load averages, the number of users, etc.

d) The notion of a local queue is not general enough— *netq* should print out relevant queues on other machines.

e) Files on intermediate machines can't be *netrm*'ed.

For Berkeley

1. The root-ownership of *netlpr* queue files is a problem. No easy solution to this problem is known at this time.

2. There are hooks in for the B, INGRES, and Q machines, and I would love to have them added to the network.

3. I'd like to see the following things happen:

   A driver for the network links to avoid character processing, which would make 9600-baud practical.

   On the Computer Center machines:

   a) The mail program should be modified as it was on Cory and VAX to handle remote names (this is high priority).

   b) A high speed link through the Bussiplexor.

   c) The remote *troff* command should be modified to work on Cory and VAX.

   It has also been suggested that all the mail programs look at a file to see if they should forward this message to an account on another machine. This would allow people to get all their mail on one machine.

Bugs

1. Extra files beginning with 'df...' are created in the 'send?' directories, with no control files ('cf...'). They should be removed periodically. *netrm* will remove them.

2. *Netcp* creates files with filenames as login names. They will never be sent and subsequent requests will be blocked. Their net queue files should be removed.

3. In general, some requests can block the queue until removed. Shorter requests will get through, and longer ones will not. Again, their net queue files should be removed.

CREATING AND MAINTAINING A DATABASE USING INGRES

by
Robert Epstein

Memorandum No. ERL - M77-71
December 16, 1977

Electronics Research Laboratory
College of Engineering
University of California, Berkeley
94720

CREATING AND MAINTAINING A DATABASE USING INGRES

1. INTRODUCTION

In this paper we describe how to create, structure and maintain relations in INGRES. It is assumed that the reader is familiar with INGRES and understands QUEL, the INGRES query language. It is strongly suggested that the document "A Tutorial on INGRES" (ERL M77/25) be read first.

This paper is divided into six sections

1. Introduction
2. Creating a Relation
3. Using Copy
4. Storage Structures
5. Secondary Indices
6. Recovery and Data Update

To create a new data base you must be a valid INGRES user and have "create data base" permission. These permissions are granted by the "ingres" superuser. If you pass those two requirements you can create a data base using the command to the Unix shell:

```
% creatdb mydata
```

where "mydata" is the name of the data base. You become the "data base administrator" (DBA) for mydata. As the DBA you have certain special powers.

1. Any relation created by you can be accessed by anyone else using "mydata". If any other user creates a relation it is strictly private and cannot be accessed by the DBA or any other user.

2. You can use the "-u" flag in ingres and printr. This enables you to use ingres on "mydata" with someone else's id. Refer to the INGRES reference manual under sections ingres(unix) and users(files) for details.

3. You can run sysmod, restore and purge on "mydata".

4. The data base by default is created to allow for multiple concurrent users. If only one user will ever use the data base at a time, the data base administrator can turn off the concurrency control. Refer to creatdb(unix) in the INGRES reference manual.

Once a data base has been created you should immediately run

```
% sysmod mydata
```

This program will convert the system relations to their "best" structure for use in INGRES. Sysmod will be explained further in section 4.

As a DBA or as a user you can create and structure new relations in any data base to which you have access. The remainder of this paper describes how this is done.

2. CREATING NEW RELATIONS IN INGRES

There are two ways to create new relations in INGRES.

```
create
retrieve into
```

"Retrieve into" is used to form a new relation from one or more existing relations. "Create" is used to create a new relation with no tuples in it.

example 1:

```
range of p is parts
range of s is supply
retrieve into newsupply(
              number = s.snum,
              p.pname,
              s.shipdate)
where s.pnum = p.pnum
```

example 2:

```
create newsupply(
       number = i2,
       pname = c20,
       shipdate = c8)
```

In example 1 INGRES creates a new relation called "newsupply", computing what the format of each domain should be. The query is then run and newsupply is modified to "cheapsort". (This will be covered in more detail in section 4.)

In example 2 "newsupply" is created and the name and format for each domain is given. The format types which are allowed are:

i1	1	byte integer
i2	2	" "
i4	4	" "
f4	4	byte floating point number
f8	8	" " " "
c1,c2,..,c255	1,2,..,255	byte character

In example 2, the width of an individual tuple is 30 bytes (2 + 20 + 8), and the relation has three domains. Beware that INGRES has limits. A relation cannot have more than 49 domains and the tuple width cannot exceed 498 bytes.

UNIX allocates space on a disk in units of 512 byte pages. INGRES gets a performance advantage by doing I/O in one block units. Therefore relations are divided into 512 byte pages. INGRES never splits a tuple between two pages. Thus some space can be wasted. There is an overhead of 12 bytes per page plus 2 bytes for every tuple on the page. The formulas are:

    number tuples per page = 500/(tuple width + 2)

    wasted space = 500 - number of tuples per page
    *(tuple width +2)

For our example there are

    22 = 500/(20 + 2)

    16 = 500 - 22 * (20 + 2)

22 tuples per page and 16 bytes wasted per page. These computations are valid only for uncompressed relations. We will return to this subject in section 4 when we discuss compression.

If you forget a domain name or format, use the "help" command. For example if you gave the INGRES command:

    help newsupply

the following would be printed:

```
Relation:               newsupply
Owner:                  bob
Tuple width:            30
Saved until:            Thu Nov 10 16:17:06 1977
Number of tuples:       0
Storage structure:      paged heap
Relation type:          user relation

 attribute name    type  length  keyno.

 number             i       2
 pname              c      20
 shipdate           c       8
```

Notice that every relation has an expiration date. This is set to be one week from the time when it was created. The "save" command can be used to save the relation longer. See "save(quel)" and "purge(unix)" in the INGRES reference manual.

3. COPYING DATA TO AND FROM INGRES

Once a relation is created, there are two mechanisms for inserting new data:

append command
copy command

Append is used to insert tuples one at a time, or for filling one relation from other relations.

Copy is used for copying data from a UNIX file into a relation. It is used for copying data from another program, or for copying data from another system. It is also the most convenient way to copy any data larger than a few tuples.

Let's begin by creating a simple relation and loading data into it.

Example:

```
create donation (name = c10, amount = f4, ext = i2)
```

Now suppose we have two people to enter. The simplest procedure is probably to run the two queries in INGRES using the append command.

```
append to donation (name="frank",amount = 5,ext = 204)

append to donation (name="harry",ext = 209,amount = 4.50)
```

Note that the order in which the domains are given does not matter. INGRES matches by recognizing attribute names and does not care in what order attributes are listed. Here is what the relation "donation" looks like now:

donation relation

name	amount	ext
frank	5.000	204
harry	4.500	209

We now have two people entered into the donation relation. Suppose we had fifty more to enter. Using the append command is far too tedious since so much typing is involved for each tuple. The copy command will better suit our purposes.

Copy can take data from a regular Unix file in a variety of formats and append it to a relation. To use the copy command first create a Unix file (typically using "ed") containing the data.

For example, let's put five new names in a file using the editor.

```
% ed
a
bill,3.50,302
sam,10.00,410
susan,,100
sally,.5,305
george,4.00,302
.
w newdom
68
q
%
```

The format of the above file is a name followed by a comma, followed by the amount, then a comma, then the extension, and finally a newline. Null entries, for example the amount for susan, are perfectly legal and default to zero for numerical domains and blanks for character domains.

To use copy we enter INGRES and give the copy command.

```
copy donation (name = c0, amount = c0, ext = c0)
        from "/mnt/bob/newdom"
```

Here is how the copy command works:

```
copy relname (list of what to copy) from "full pathname"
```

In the case above we wrote:

```
copy donation (. . .) from "/mnt/bob/newdom"
```

Although amount and ext are stored in the relation as f4 (floating point) and i2 (integer), in the Unix file they were entered as characters. In specifying the format of the domain, copy accepts:

```
domain = format
```

where domain is the domain name and the format in the UNIX file is one of

```
i1, i2, i4           (true binary integer of size 1, 2, or 4)
f4, f8               (true binary float point of size 4 or 8)
c1, c2, c3,...c255   (a fixed length character string)
c0                   (a variable length character string de-
                     limited by a comma, tab or new line)
```

In the example we use

```
name = c0, amount = c0, extension = c0
```

This means that each of the domains was stored in the Unix file as variable length character strings. Copy takes the first com-

ma, tab, or new line character as the end of the string. This by far is the most common use of copy when the data is being entered into a relation for the first time.

Copy can also be used to copy data from a relation into a Unix file. For example:

```
copy donation (name = c10, amount = c10, ext = c5)
         into "/mnt/bob/data"
```

This will cause the following to happen:

1. If the file /mnt/bob/data already exists it will be destroyed.

2. The file is created in mode 600 (read/write by you only)

3. Name will be copied as a 10 character field, immediately followed by amount, immediately followed by ext. Amount will be converted to a character field 10 characters wide. Ext will be converted to a character field 5 characters wide.

The file "/mnt/bob/data" would be a stream of characters looking like this:

```
frank          5.000  204harry           4.500  209bill
           3.500  302sam          10.000  410susan
       0.000  100sally         0.500  305george
   4.000  302
```

The output was broken into four lines to make it fit on this page. In actuality the file is a single line. Another example:

```
copy (name = c0, colon = d1, ext = c0, comma = d1
        amt = c0, nl = d1) into "/mnt/bob/data"
```

In this example "c0" is interpreted to mean "use the appropriate character format". For character domains it is the width of the domain. Numeric domains are converted to characters according to the INGRES defaults (see ingres(unix)).

The statements:

```
colon = d1
comma = d1
nl = d1
```

are used to insert one colon, comma, and newline into the file. The format "d1" is interpreted to mean one dummy character. When copying into a Unix file, a selected set of characters can be inserted into the file using this "dummy domain" specification. Here is what the file "/mnt/bob/data" would look like:

```
frank      :   204,     5.000
harry      :   209,     4.500
bill       :   302,     3.500
sam        :   410,    10.000
susan      :   100,     0.000
sally      :   305,     0.500
george     :   302,     4.000
```

If you wanted a file with the true binary representation of the numbers you would use:

```
copy (name = c10, amount = f4, ext = i2)
```

This would create a file with the exact copy of each tuple, one after the other. This is frequently desireable for temporary backup purposes and it guarantees that floating point domains will be exact.

TYPICAL ERRORS

There are 17 different errors that can occur in copy. We will go through the most common ones.

Suppose you have a file with

```
bill,3.5,302
sam,10,410,
susan,3,100
```

and run the copy command

```
copy donation (name = c0, amount = c0, ext = c0)
      from "/mnt/bob/data"
```

You would get the error message

5809: COPY: bad input string for domain amount. Input was "susan". There were 2 tuples sucessfully copied from /mnt/bob/data into donation.

What happened is that line 2 had an extra comma. The first two tuples were copied correctly. For the next tuple, name = "" (blank), amount = "susan", and ext = "3". Since "susan" is not a proper floating point number, an error was generated and processing was stopped after two tuples.

If you tried to copy the file with a file such as

```
nancy,5.0,35000
```

you would get the error message

5809: COPY: bad input string for domain ... Input was "35000".
There were 0 tuples successfully copied ...om /mnt/bob/data into donation.

Here, since ext is an i2 (integer) domain, it cannot exceed the value 32767.

There are numerous other error messages, most of which are self-explanatory.

In addition there are three, non-fatal warnings which may appear on a copy "from".

If you are copying from a file into a relation which is ISAM or hash, a count of the number of duplicate tuples will appear, (if there were any). This will never appear on a "heap" because no duplicate checking is performed.

INGRES does not allow control characters (such as "bell" etc.) to be stored. If copy reads any control characters, it converts them to blanks and reports the number of domains that had control characters in them.

If you are copying using the c0 option, copy will report if any character strings were longer than their domains and had to be truncated.

SPECIAL FEATURES

There are a few special functions that make copy a little easier to use

Bulk copy

If you ask for:

```
copy relname () from "file"
    or
copy relname () into "file"
```

copy expands the statement to mean:

copy each domain in its proper order according to its proper format.

So, if you said

```
copy donation () into "/mnt/bob/donation"
```

it would be the same as asking for:

```
copy donation (name = c10, amount = f4, ext = i2)
    into "/mnt/bob/donation"
```

This provides a convenient way to copy whole relations to and from INGRES.

2. Dummy Domains

If you are copying data from another computer or program, frequently there will be a portion of data that you will want to ignore. This can be done using the dummy domain specifications d0, d1, d2 ... d511. For example

```
copy rel (dom1 = c5, dummy = d2, dom2 = i4,
    dumb = d0) from "/mnt/me/data"
```

The first five characters are put in dom1, the next two characters are ignored. The next four bytes are an i4 (integer) and go in dom2, and the remaining delimited string is ignored. The name given to a dummy specifier is ignored.

As mentioned previously, dummy domains can be used on a copy "into" a Unix file for inserting special characters. The list of recognizable names includes:

nl	newline
tab	tab character
sp	space
nul	a zero byte
null	a zero byte
comma	,
dash	-
colon	:
lparen	(
rparen	)

3. Truncation

It is not uncommon to have a mistake occur and need to start over. The simplest way to do that is to "truncate" the relation. This is done by the command:

```
modify relname to truncated
```

This has the effect of removing all tuples in relname, releasing all disk space, and making relname a heap again. It is the logical equivalent of a destroy followed by a create (but with a lot less typing).

Since formatting mistakes are possible with copy, it is not generally a good idea to copy data into a relation that already has valid data in it. The best procedure is to create a temporary relation with the same domains as the existing relation. Copy data into the temporary relation and then append it to the real relation. For example:

```
create tempdom(name=c10,amount=f4,ext=i2)

copy tempdom(name=c0,amount=c0,ext=c0)
from "/mnt/bob/data"

range of td is tempdom
append to donation(td.all)
```

4. Specifing Delimitors.

Sometimes it is desirable to specify what the delimiting character should be on a copy "from" a file. This can be done by specifing:

```
domain = c0delim
```

where "delim" is a valid delimitor taken from the list of recognizable names. This list was summarized on the previous page under "dummy domains". For example:

```
copy donation (name = c0nl) from "/mnt/me/data"
```

will copy names from the file to the relation. Only a new line will delimit the names so any commas or tabs will be passed along as part of the name.

When copying "into" a Unix file, the "delim" is actually written into the file, so on a copy "into" the specification:

```
copy donation (name = c0nl) into "/mnt/me/file"
```

will cause "name" to be written followed by a new line character.

4. CHOOSING THE BEST STORAGE STRUCTURES

We now turn to the issue of efficiency. Once you have created a relation and inserted your data using either copy or append, INGRES can process any query on the relation. There are several things you can do to improve the speed at which INGRES can process a query.

INGRES can store a relation in three different internal structures. These are called "heap", "isam", and "hash". First we will briefly describe each structure and then later expand our discussion.

HEAP

When a relation is first created, it is created as a "heap". There are two important properties about a heap: duplicate tuples are not removed, and nothing is known about the location of the tuples. If you ran the query:

```
range of d is donation
retrieve (d.amount) where d.name = "bill"
```

INGRES would have to read every tuple in the relation looking for those with name "bill". If the relation is small this isn't a serious matter. But if the relation is very large, this can take minutes (or even hours!).

HASH

A relation whose structure is "hash" can give fast access to searches on certain domains. (Those domains are usually referred to as "keyed domains".) In addition, a "hashed" relation contains no duplicate tuples. For example, suppose the donation relation is stored hashed on domain "name". Then the query:

```
retrieve (d.amount) where d.name = "bill"
```

will run quickly since INGRES knows approximately where on disk the tuple is stored. If the relation contains only a few tuples you won't notice the difference between a "heap" and a "hash" structure. But as the relation becomes larger, the difference in speed becomes much more noticeable.

ISAM

An isam structure is one where the relation is sorted on one or more domains, (also called keyed domains). Duplicates are also removed on "isam relations". When new tuples are appended they are placed "approximately" in their sorted position in the relation. (The "approximately" will be explained a bit later.)

Suppose donation is isam on name. To process the query

```
retrieve (d.amount) where d.name = "bill"
```

INGRES will determine where in the sorted order the name "bill" would be and read only those portions of the relation.

Since the relation is approximately sorted, an isam structure is also efficient for processing the query:

```
retrieve (d.amount) where d.name >= "b" and d.name < "g"
```

This query would retrieve all names beginning with "b" through "f". The entire relation would not have to be searched since it is isam on name.

SPECIFYING THE STORAGE STRUCTURE

Any user created relation can be converted to any storage structure using the "modify" command. For example

```
modify donation to hash on name
```

or

```
modify donation to isam on name
```

or even

```
modify donation to heap
```

PRIMARY AND OVERFLOW PAGES

At this point it is necessary to introduce the concepts of primary and overflow pages on hash and isam structures. Both hash and isam are techniques for assigning specific tuples to specific pages of a relation based on the tuple's keyed domains. Thus each page will contain only a certain specified subset of the relation.

When a new tuple is appended to a hash or isam relation, INGRES first determines what page it belongs to, and then looks for room on that page. If there is space then the tuple is placed on that page. If not, then an "overflow" page is created and the tuple is placed there.

The overflow page is linked to the original page. The original page is called the "primary" page. If the overflow page became full, then INGRES would connect an overflow page to it. We would then have one primary page linked to an overflow page, linked to another overflow page. Overflow pages are dynamically added as needed.

SPECIFYING FREE SPACE

The modify command also lets you specify how much room to leave for the relation to grow. As was mentiond in "create", relations are divided into pages. A "fillfactor" can be used to specify how full to make each primary page. This decision should be based only on whether more tuples will be appended to the relation. For example:

```
modify donation to isam on name where fillfactor = 100
```

This tells modify to make each page 100% full if at all possible.

```
modify donation to isam on name where fillfactor = 25
```

This will leave each page 25% full or, in other words, 75% empty. We would do this if we had roughly 1/4 of the data already loaded and it was fairly well distributed about the alphabet.

Keep in mind that if you don't specify the fillfactor, INGRES will typically default to a reasonable choice. Also when a page becomes full, INGRES automatically creates an "overflow" page so it is never the case that a relation will be unable to expand.

When modifying a relation to hash, an additional parameter "minpages" can be specified. Modify will guarantee that at least "minpage" primary pages will be allocated for the relation.

Modify computes how may primary pages will be needed to store the existing tuples at the specified fillfactor assuming that no overflow pages will be necessary originally. If that number is less than minpages, then minpages is used instead.

For example:

```
modify donation to hash on name where fillfactor = 50,
    minpages = 1

modify donation to hash on name where minpages = 150
```

In the first case we guarantee that no more pages than are necessary will be used for 50% occupancy. The second case is typically used for modifying an empty or near empty relation. If the approximate maximum size of the relation is known in advance, minpages can be used to guarantee that the relation will have its expected maximum size.

There is one other option available for hash called "maxpages". Its syntax is the same as minpages. It can be used to specify the maximum number of primary pages to use.

COMPRESSION

The three storage structures (heap, hash, isam) can optionally have "compression" applied to them. To do this, refer to the storage structures as cheap, chash, and cisam. Compression

reduces the amount of space needed to store each tuple internally. The current compression technique is to suppress trailing blanks in character domains. Using compression will never require more space and typically it can save disk space and improve performance. Here is an example:

```
modify donation to cisam on name where fillfactor = 100
```

This will make donation a compressed isam structure and fill every page as full as possible. With compression, each tuple can have a different compressed length. Thus the number of tuples that can fit on one page will depend on how successfully they can be compressed.

Compressed relations can be more expensive to update. In particular if a replace is done on one or more domains and the compressed tuple is no longer the same length, then INGRES must look for a new place to put the tuple.

TWO VARIATIONS ON A THEME

As mentioned, duplicates are not removed from a relation stored as a heap. Frequently it is desirable to remove duplicates and sort a heap relation. One way of doing this is to modify the relation to isam specifying the order in which to sort the relation. An alternative to this is to use either "heapsort" or "cheapsort". For example

```
modify donation to heapsort on name, ext
```

This will sort the relation by name then ext. The tuples are further sorted on the remaining domains, in the order they were listed in the original create statement. So in this case the relation will be sorted on name then ext and then amount. Duplicate tuples are always removed. The relation will be left as a heap. Heapsort and cheapsort are intended for sorting a temporary relation before printing and destroying it. It is more efficient than modifying to isam because with isam INGRES creates a "directory" containing key information about each page. The relation will NOT be kept sorted when further updates occur.

Examples:

Here are a collection of examples and comments as to the efficiency of each query. The queries are based on the relations:
parts(pnum, pname, color, weight, qoh)
supply(snum, pnum, jnum, shipdate, quan)

```
range of p is parts
range of s is supply

modify parts to hash on pnum
modify supply to hash on snum,jnum
```

```
retrieve (p.all) where p.pnum = 10
```

INGRES will recognize that parts is hashed on pnum and go directly to the page where parts with number 10 would be stored.

```
retrieve (p.all) where p.pname = "tape drive"
```

INGRES will read the entire relation looking for matching pnames.

```
retrieve (p.all) where p.pnum < 10  and p.pnum > 5
```

INGRES will read the entire relation because no exact value for pnum was given.

```
retrieve (s.shipdate) where s.snum = 471 and s.jnum = 1008
```

INGRES will recognize that supply is hashed on the combination of snum and jnum and will go directly to the correct page.

```
retrieve (s.shipdate) where s.snum = 471
```

INGRES will read the entire relation. Supply is hashed on the combination of snum and jnum. Unless INGRES is given a unique value for both, it cannot take advantage of the storage structure.

```
retrieve (p.pname, s.shipdate) where
p.pnum = s.pnum and s.snum = 471 and s.jnum = 1008
```

INGRES will take advantage of both storage structures. It will first find all s.pnum and s.shipdate where s.snum = 471 and s.jnum = 1008. After that it will look for all p.pname where p.pnum is equal to the correct value.

This example illustrates the idea that it is frequently a good idea to hash a relation on the domains where it is "joined" with another relation. For example, in this case it is very common to ask for p.pnum = s.pnum

To summarize:

To take advantage of a hash structure, INGRES needs an exact value for each key domain. An exact value is anything such as:

```
s.snum = 471
s.pnum = p.pnum
```

An exact value is not

```
s.snum >= 471
(s.snum = 10 or s.snum = 20)
```

Now let's consider some cases using isam

```
modify supply to isam on snum,shipdate
retrieve (s.all) where s.snum = 471
and s.shipdate > "75-12-31"
and s.shipdate < "77-01-01"
```

Since supply is sorted first on snum and then on shipdate, INGRES can take full advantage of the isam structure to locate the portions of supply which satisfy the query.

```
retrieve (s.all) where s.snum = 471
```

Unlike hash, an isam structure can still be used if only the first key is provided.

```
retrieve (s.all) where s.snum > 400 and s.snum < 500
```

Again INGRES will take advantage of the structure.

```
retrieve (s.all) where s.shipdate >= "75-12-31" and
s.shipdate <= "77-01-01"
```

Here INGRES will read the entire relation. This is because the first key (snum) is not provided in the query.

To summarize:

Isam can provide improved access on either exact values or ranges of values. It is useful as long as at least the first key is provided.

To locate where the tuples are in an isam relation, INGRES searches the isam directory for that relation. When a relation is modified to isam, the tuples are first sorted and duplicates are removed. Next, the relation is filled (according to the fillfactor) starting at page 0, 1, 2... for as many pages as are needed.

Now the directory is built. The key domains from the first tuple on each page are collected and organized into a directory (stored in the relation on disk). The directory is never changed until the next time a modify is done.

Whenever a tuple is added to the relation, the directory is searched to find which page the new tuple belongs on. Within that page, the individual tuples are NOT kept sorted. This is what is meant by "approximately" sorted.

HEAP v. HASH v. ISAM

Let's now compare the relative advantages and disadvantages of each option. A relation is always created as a heap. A heap is the most efficient structure to use to initially fill a relation using copy or append.

Space from deleted tuples of a heap is only reused on the last page. No duplicate checking is done on a heap relation.

Hash is advantageous for locating tuples referenced in a qualification by an exact value. The primary page for tuples with a specific value can be easily computed.

Isam is useful for both exact values and ranges of values. Since the isam directory must be searched to locate tuples, it is never as efficient as hash.

OVERFLOW PAGES

When a tuple is to be inserted and there is no more room on the primary page of a relation, then an overflow page is created. As more tuples are inserted, additional overflow pages are added as needed. Overflow pages, while necessary, decrease the system performance for retrieves and updates.

For example, let's suppose that supply is hashed on snum and has 10 primary pages. Suppose the value snum = 3 falls on page 7. To find all snum = 3 requires INGRES to search primary page 7 and all overflow pages of page 7 (if any). As more overflow pages are added the time needed to search for snum = 3 will increase. Since duplicates are removed on isam and hash, this search must be performed on appends and replaces also.

When a hash or isam relation has too many overflow pages it should be remodified to hash or isam again. This will clear up the relation and eliminate as many overflow pages as possible.

UNIQUE KEYS

When choosing key domains for a relation it is desirable to have each set of key domains as unique as possible. For example, employee id numbers typically have no duplicate values, while something like color is likely to have only a few distinct values, and something like sex, to the best of our knowledge, has only two values.

If a relation is hashed on domain sex then you can expect to have all males on one primary page and all its overflow pages and a corresponding situation with females. With a hash relation there is no solution to this problem. A trade-off must be made between the most desirable key domains to use in a qualification versus the uniqueness of the key values.

Since isam structure can be used if at least the first key is provided, extra key domains can sometimes be added to increase uniqueness. For example, suppose the supply relation has only 10 unique supplier numbers but thousands of tuples. Choosing an isam structure with the keys snum and jnum will probably give

many more unique keys. However, the directory size will be larger and consequently it will take longer to search. When providing additional keys just for the sake of increasing uniqueness, try to use the smallest possible domains.

SYSTEM RELATIONS

INGRES uses three relations ("relation", "attribute", and "indexes") to maintain and organize a data base. The "relation" relation has one tuple for each relation in the data base. The "attribute" relation has one tuple for each attribute in each relation. The "indexes" relation has one tuple for each secondary index.

INGRES accesses these relations in a very well defined manner. A program called "sysmod" should be used to modify these relations to hash on the appropriate domains. To use sysmod the data base administrator types

```
% sysmod data-base-name
```

Sysmod should be run initially after the data base is created and subsequently as relations are created and the data base grows. It is insufficient to run sysmod only once and forget about it. Rerunning sysmod will cause the system relations to be remodified. This will typically remove most overflow pages and improve system response time for everything.

5. SECONDARY INDICES

Using an isam or hash structure provides a fast way to find tuples in a relation given values on the key domains. Sometimes this is not enough. For example, suppose we have the donation relation

```
donation(name, amount, ext)
```

hashed on name. This will provide fast access to queries where the qualification has an exact value for name. What if we also will be doing queries giving exact values for ext?

Donation can be hashed either on name or ext, so we would have to choose which is more common and hash donation on that domain. The other domain (say ext) can have a secondary index. A secondary index is a relation which contains each "ext" together with the exact location of where the tuple is in the relation donation.

The command to create a secondary index is:

```
index on donation is donext (ext)
```

The general format is:

```
index on relation_name is secondary_index_name (domains)
```

Here we are asking INGRES to create a secondary index on the relation donation. The domain being indexed is "ext". Indices are formed in three steps:

1. "Donext" is created as a heap.
2. For each tuple in donation, a tuple is inserted in "donext" with the value for ext and the exact location of the corresponding tuple in donation.
3. By default "donext" is modified to isam.

Now if you run the query

```
range of d is donation
retrieve(d.amount) where d.ext = 207
```

INGRES will automatically look first in "donext" to find ext = 207. When it finds one it then goes directly to the tuple in the donation relation. Since "donext" is isam on ext, search for ext = 207 can typically be done rapidly.

If you run the query

```
retrieve(d.amount) where d.name = "frank"
```

then INGRES will continue to use the hash structure of the relation "donation" to locate the qualifying tuples.

Since secondary indices are themselves relations, they also can be either hash, isam, chash or cisam. It never makes sense to a secondary index a heap.

The decision as to what structure to make them on involves the same issues as were discussed before:

Will the domains be referenced by exact value?
Will they be referenced by ranges of value?
etc.

In this case the "ext" domain will be referenced by exact values, and since the relation is nearly full we will do:

```
modify donext to hash on ext where fillfactor = 100
and minpages = 1
```

Secondary indices provide a way for INGRES to access tuples based on domains that are not key domains. A relation can have any number of secondary indices and in addition each secondary index can be an index on up to six domains of the primary relation.

Whenever a tuple is replaced, deleted or appended to a primary relation, all secondary indices must also be updated. Thus secondary indices are "not free". They increase the cost of updating the primary relation, but can decrease the cost of finding tuples in the primary relation.

Whether a secondary index will improve performance or not strongly depends on the uniqueness of the values of the domains being indexed. The primary concern is whether searching through the secondary index is more efficient than simply reading the entire primary relation. In general it is if the number of tuples which satisfy the qualification is less than the number of total pages (both primary and overflow) in the primary relation.

For example if we frequently want to find all people who donated less than five dollars, consider creating

```
index on donation is donamount (amount)
```

By default donamount will be isam on amount. IF INGRES processes the query:

```
retrieve(d.name) where d.amount < 5.0
```

it will locate d.amount < 5.0 in the secondary index and for each tuple it finds will fetch the corresponding tuple in donation. The tuples in donamount are sorted by amount but the tuples in donation are not. Thus in general each tuple fetch from donation via donamount will be on a different page. Retrieval using the secondary index can then cause more page reads than simply reading all of donation sequentially! So in this example it would be a bad idea to create the secondary index.

6. RECOVERY AND DATA UPDATE

INGRES has been carefully designed to protect the integrity of a data base against certain classes of system failures. To do this INGRES processes changes to a relation using what we call "deferred update" or "batch file update". In addition there are two INGRES programs "restore" and "purge" that can be used to check out a data base after a system failure. We will first discuss how deferred updates are created and processed, and second we will discuss the use of purge and restore.

DEFERRED UPDATE (Batch update)

An append, replace or delete command is run in four steps:

1. An empty batch file is created.
2. The command is run to completion and each change to the result relation is written into the batch file.
3. The batch file is read and the relation and its secondary indices (if any) are actually updated.
4. The batch file is destroyed and INGRES returns back to the user.

Deferred update defers all actual updating until the very end of the query. There are three advantages to doing this.

1. Provides recovery from system failures

If the system "crashes" during an update, the INGRES recovery program will decide to either run the update to completion or else "back out" the update, leaving the relation as it looked before the update was started.

2. Prevents infinite queries

If "donation" were a heap and the query

```
range of d is donation
append to donation(d.all)
```

were run without deferred update, it would terminate only when it ran out of space on disk! This is because INGRES would start reading the relation from the beginning and appending each tuple at the end. It would soon start reading the tuples it had just previously appended and continue indefinitely to "chase its tail".

While this query is certainly not typical, it illustrates the point. There are certain classes of queries where problems occur if WHEN an update actually occurs is not precisely defined. With deferred update we can guarantee consistent and logical results.

3. Speeds up processing of secondary indices

Secondary indices can be updated faster if they are done one at a time instead of all at once. It also insures protection against the secondary index becoming inconsistent with its primary relation.

TURNING DEFERRED UPDATE OFF

If you are not persuaded by any of these arguments, INGRES allows you to turn deferred update off! Indeed there are certain cases when it is appropriate (although certainly not essential) to perform updates directly, that is, the relation is updated while the query is being processed.

To use direct update, you must be given permission by the INGRES super user. Then when invoking INGRES specify the "-b" flag which turns off batch update.

```
% ingres mydate -b
```

INGRES will use direct update on any relation without secondary indices. It will still silently use deferred update if a relation has any secondary indices. By using the "-b" flag you are sacrificing points 1 and 2 above. In most cases you SHOULD NOT use the -b flag.

If you are using INGRES to interactively enter or change one tuple at a time, it is slightly more efficient to have deferred update turned off. If the system crashes during an update the person entering the data will be aware of the situation and can check whether the tuple was updated or not.

RESTORE

INGRES is designed to recover from the common types of system crashes which leave the Unix file system intact. It can recover from updates, creates, destroys, modifies and index commands.

INGRES is designed to "fail safe". If any inconsistancies are discovered or any failures are returned from Unix, INGRES will generate a system error message (SYSERR) and exit.

Whenever Unix crashes while INGRES is running or whenever an INGRES syserr occurs, it is generally a good idea to have the date base administrator run the command

```
% restore data_base_name
```

The restore program performs the following functions:

1. Looks for batch update files. If any are found, it examines each one to see if it is complete. If the system crash occured while the batch file was being read and the data base being updated, then restore will complete the update. Otherwise the batch file was not completed and it is simply des-

troyed; the effect is as though the query had never been run.

2. Checks for uncompleted modify commands. This step is crucial. It guarantees that you will either have the relation as it existed before the modify, or restore will complete the modify command. Modify works by creating a new copy of the relation in the new structure. Then when it is ready to replace the old relation, it stores the new information in a "modify batch file". This enables restore to determine the state of uncompleted modifies.

3. Checks consistency of system relations. This check is used to complete "destory" commands, back out "create" commands, and back out or complete "index" commands that were interrupted by a system crash.

4. Purges temporary relations and files. Restore executes the "purge" program to remove temporary relations and temporary files created by the system. Purge will be discussed in more detail a bit later.

Restore cannot tell the user which queries have run and which have not. It can only identify those queries which were in the process of being run when the crash occured. When batching queries together, it is a good idea to save the output in a file. By having the monitor print out each query or set of queries, the user can later identify which queries were run.

Restore has several options to increase its usability. They are specified by "flags". The options include:

-a	ask before doing anything
-f	passed to purge. used to remove temporary files.
-p	passed to purge. used to destory expired relations.
no database	restores all data bases for which you are the dba.

Of these options the "-a" is the most important. It can happen that a Unix crash can cause a page of the system catalogues to be incorrect. This might cause restore to destory a relation. In fact, you might want to "patch" the system relations to correct the problem. No restore program can account for all possibilities. It is therefore no replacement (fortunately) for a human.

If "-a" is specified, restore will state what it wants to do and then ask for permission. It reads standard input and accepts "y" to mean go ahead and anything else to mean no. For example, to have restore ask you before doing anything

```
restore -a mydatabase
```

To have it take "no" for all its questions

```
restore -a mydatabase </dev/null
```

Using the -a flag, restore might ask for permission to perform some cleanup; for example, if it finds an attribute for which there is no corresponding relation, or if it finds a secondary index for which there is no primary relation, etc.

To date, we have never had a system crash which INGRES could not recover from. This does not mean that it will never happen, but rather that it shouldn't be too great a concern for you. It should be mentioned that restore is not a substitution for doing periodic backing up, nor does it ever perform such a function.

PURGE

Purge can be used to report expired relations, destroy temporary system relations, remove extraneous files, and destory expired relations. To use purge you must be the DBA for the data base.

```
% purge mydatabase
```

Purge has several options which are specified by flags which are worth noting:

-f (default is off) remove all extraneous files. Each file is reported and then removed. If "-f" is not specified then the file is only reported.

-p (default is off) destroy all expired relations. Each expired relation is reported and if "-p" was specified the relation is destroyed.

Purge always destroys relations and files which are known to be INGRES system temporaries. When processing multi-variable queries and queries with aggregate functions, INGRES will usually create temporary relations with intermediate results. These relations always begin with the characters "_SYS". Other INGRES commands create temporary files which also begin with "_SYS". Under normal processing they are always destroyed. If a system crash occurs, they might be left. Purge will always clean up the temporary system files. It cleans up the user relations only when specifically asked to.

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

ABSTRACT

Ex a line oriented text editor, which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex;* the display editing features of *ex* are described in *An Introduction to Display Editing with Vi.* Other documents about the editor include the introduction *Edit: A tutorial*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

September 16, 1980

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

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

1. Starting ex

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

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

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

ex [**−**] [**−v**] [**−t** *tag*] [**−r**] [**−l**] [**−w***n*] [**−x**] [**−R**] [**+***command*] name ...

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

ex name

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

---

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

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

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

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

2. File manipulation

2.1. Current file

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

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

2.2. Alternate file

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

2.3. Filename expansion

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

2.4. Multiple files and named buffers

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

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

2.5. Read only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from 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

---

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

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

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

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

3. Exceptional Conditions

3.1. Errors and interrupts

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

3.2. Recovering from hangups and crashes

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

ex −r *resume*

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

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

ex −r

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

4. Editing modes

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

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

5. Command structure

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

---

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

5.1. Command parameters

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

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

5.2. Command variants

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

5.3. Flags after commands

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

5.4. Comments

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

5.5. Multiple commands per line

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

5.6. Reporting large changes

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

6. Command addressing

6.1. Addressing primitives

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

---

† Counts are rounded down if necessary.

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

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

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

6.2. Combining addressing primitives

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

7. Command descriptions

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

address **command** *! parameters count flags*

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

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

abbreviate *word rhs* abbr: **ab**

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

(.) **append** abbr: **a**
text
.

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

---

† The forms '.+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.
‡ The forms \/ and \? scan using the last regular expression used in a scan; after a substitute // and ?? would scan using the substitute's regular expression.
† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

a!
text
.

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

args

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

(. , .) **change** *count* abbr: **c**
text
.

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

c!
text
.

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

(. , .) **copy** *addr flags* abbr: **co**

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

(. , .) **delete** *buffer count flags* abbr: **d**

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

edit *file* abbr: **e**
ex *file*

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

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

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty*, a terminal, or a binary or executable file (as indicated by the first word).
‡ If executed from within *open* or *visual*, the current line is initially the first line of the file.

e! *file*

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

e +*n file*

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

file abbr: **f**

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

file *file*

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

(1 , $) **global** */pat/ cmds* abbr: **g**

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

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

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

g! */pat/ cmds* abbr: **v**

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

(.) **insert** abbr: **i**
text
.

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

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

i!
text
.

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

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

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

j!

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

(.) **k** *x*

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

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

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

map *lhs rhs*

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

(.) **mark** *x*

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

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

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

next abbr: **n**

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

n!

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

n *filelist*
n +*command filelist*

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

(. , .) **number** *count flags* abbr: **#** or **nu**

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

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

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

preserve

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

(. , .) **print** *count* abbr: **p** or **P**

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

(.) **put** *buffer* abbr: **pu**

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

quit abbr: **q**

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

q!

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

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

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

---

‡ Not available in all v2 editors due to memory constraints.
* But no modifying commands may intervene between the *delete* or *yank* and the *put*, nor may lines be moved between files without using a named buffer.
† *Ex* will also issue a diagnostic if there are more files in the argument list.

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

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

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

recover *file*

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

rewind abbr: **rew**

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

rew!

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

set *parameter*

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

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

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

shell abbr: **sh**

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

source *file* abbr: **so**

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

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

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

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

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

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

stop

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

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

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

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

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

ta *tag*

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

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

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

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

undo abbr: **u**

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

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

unmap *lhs*

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

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

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

version abbr: **ve**

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

---

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

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

(.) **visual** *type count flags* abbr: **vi**

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

visual file
visual +*n* file

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

(1 , $) **write** *file* abbr: **w**

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

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

(1 , $) **write>>** *file* abbr: **w>>**

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

w! *name*

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

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

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

wq *name*

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

wq! *name*

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

xit *name*

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

(. , .) **yank** *buffer count* abbr: **ya**

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

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

(.**+1**) **z** *count*

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

(.) **z** *type count*

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

! *command*

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

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

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

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

(**$**) **=**

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

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

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

^D

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

(.+1 , .+1)
(.+1 , .+1) |

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

---

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

(. , .) & *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

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

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

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

8.2. Magic and nomagic

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

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

8.3. Basic regular expression summary

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

char — An ordinary character matches itself. The characters ‘↑’ at the beginning of a line, ‘$’ at the end of line, ‘*’ as any character other than the first, ‘.’, ‘\’, ‘[’, and ‘~’ are not ordinary characters and must be escaped (preceded) by ‘\’ to be treated as such.

↑ — At the beginning of a pattern forces the match to succeed only at the beginning of a line.

$ — At the end of a regular expression forces the match to succeed only at the end of the line.

. — Matches any single character except the new-line character.

\< — Forces the match to occur only at the beginning of a “variable” or “word”; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> — Similar to ‘\<’, but matching the end of a “variable” or “word”, i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

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

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

8.4. Combining regular expression primitives

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

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

8.5. Substitute replacement patterns

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

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

9. Option descriptions

autoindent, ai default: noai

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

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

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

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a **^D**. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

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

autoprint, ap default: ap

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

autowrite, aw default: noaw

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

beautify, bf default: nobeautify

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

directory, dir default: dir=/tmp

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

edcompatible default: noedcompatible

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

errorbells, eb default: noeb

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

hardtabs, ht default: ht=8

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

ignorecase, ic default: noic

‡‡ Version 3 only.

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

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

lisp default: nolisp

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

list default: nolist

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

magic default: magic for *ex* and *vi*†

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

mesg default: mesg

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

number, nu default: nonumber

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

open default: open

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

optimize, opt default: optimize

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

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the { and } operations in *open* and *visual.* The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt default: prompt

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

redraw default: noredraw

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

---

† *Nomagic* for *edit.*

‡‡ Version 3 only.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. ‡‡ For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll=½ window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

sections default: sections=SHNHH HU

Specifies the section macros for the **[[** and **]]** operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8

Gives the width a software tab stop, used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

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

taglength, tl default: tl=0

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

---

‡‡ Version 3 only.
† 2 for *edit*.

tags default: tags=tags /usr/lib/tags

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

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

warn default: warn

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

window default: window=speed dependent

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

w300, w1200, w9600

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

wrapscan, ws default: ws

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

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.

writeany, wa default: nowa

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

10. Limitations

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

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

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

---

‡‡ Version 3 only.

Appendix: List of Changes from Version 3.5 to Version 3.6 of the Text Editor ex/vi

- A kernel problem on the version 7 PDP-11 overlay systems which causes bad EMT traps to happen randomly, core dumping the editor, has been programmed around by catching EMT traps.

- A bug which prevented using a screen larger than 48 lines has been fixed.

- A bug which allowed you to set window to a value larger than your screen size has been fixed.

- The screen size limit on non-VM UNIX systems has been increased to 66 lines or 5000 characters, to allow the Ann Arbor Ambassador terminal to be used.

- A bug which caused hangups to be ignored on USG systems has been fixed.

- A bug which caused maps with multiple changes on multiple lines to mess up has been fixed.

- If you get I/O errors, the file is considered "not edited" so that you don't accidently clobber the good file with a munged up buffer.

- An inefficiency in 3.5 which caused the editor to always call **ttyname** has been fixed.

- A bug which prevented the source (**.so**) command from working in an EXINIT or from visual has been fixed.

- A bug which caused readonly to be cleared when reading from a writable file with **r** has been fixed.

- The name **suspend** has been made an alias for **stop**.

- The **stop** command now once again works correctly from command mode.

- On a dumb terminal at 1200 baud, **slowopen** is now the default.

- A bug in the shell script **makeoptions** which searched for a string that appeared earlier in a comment has been fixed.

- A bug that caused an infinite loop when you did **:s/</&/g** has been fixed.

- A bug that caused & with no previous substitution to give "re internal error" has been fixed.

- A bug in the binary search algorithm for **tags** which sometimes prevented the last tag in the file from being found has been fixed.

- Error messages from expreserve no longer output a linefeed, messing up the screen.

- The message from expreserve telling you a buffer was saved when your phone was hung up has be amended to say the "editor was terminated," since a **kill** can also produce that message.
- The **directory** option, which has been broken for over a year, has been fixed.
- The **r** command no longer invokes input mode macros.
- A bug which caused strangeness if you set **wrapmargin** to 1 and typed a line containing a backslash in column 80 has been fixed.
- A bug which caused the **r**<RETURN> at the wrapmargin column to mess up has been fixed.
- On terminals with both scroll reverse and insert line, the least expensive of the two will be used to scroll up. This is usually scroll reverse, which is much less annoying than insert line on terminals such as the mime I and mime 2a.
- A bug which caused vi to estimate the cost of cursor motion without taking into account padding has been fixed.
- The failure of the editor to check counts on ^F and ^B commands has been fixed.
- The **remap** option failed completely if it was turned off. This has been fixed.
- A check of the wrong limit on a buffer for the right hand side of substitutions has been fixed. Overflowing this buffer could produce a core dump.
- A bug causing the editor to go into insert mode if you pressed the RETURN key during an **R** command has been fixed.
- A bug preventing the + command from working when you edit a new file has been fixed by making it no longer an error to edit a new file (when you first enter the editor). Instead you are told it is a new file.
- If an error happens when you are writing out a file, such as an interrupt, you are warned that the file is incomplete.

−ME REFERENCE MANUAL

Release 1.1/25

Eric P. Allman

Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes in extremely terse form the features of the **−me** macro package for version seven NROFF/TROFF. Some familiarity is assumed with those programs, specifically, 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.

For a more casual introduction to text processing using NROFF, refer to the document *Writing Papers with NROFF using −me.*

There are a number of macro parameters that may be adjusted. Fonts may be set to a font number only. In NROFF font 8 is underlined, and is set in bold font in TROFF (although font 3, bold in TROFF, is not underlined in NROFF). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are "pseudo-fonts"; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

    \f8

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is:

    .nr pi 8n

and not

    .nr pi 8

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form **$***x* may be used in expressions but should not be changed. Macros of the form **$***x* perform some function (as described) and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

All names in −me follow a rigid naming convention. The user may define number registers, strings, and macros, provided that s/he uses single character 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

---

†NROFF and TROFF are Trademarks of Bell Laboratories.

too small for easy readability, so the default is to space one sixth inch.

This documentation was TROFF'ed on December 14, 1979 and applies to version 1.1/25 of the −me macros.

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.

.ip *T I* Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or **\n(ii** [5n] spaces if *I* is not specified) more than a 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. If *T* will not fit in the space provided, **.ip** will start a new line.

.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**.

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($0**) is used. The values of the individual parts of the section number are maintained in **\n($1** through **\n($6**. There is a **\n(ss** [1v] space before the section. *T* is printed as a section title in font **\n(sf** [8] and size **\n(sp** [10p]. The "name" of the section may be accessed via ***($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** [0] is added to the section title (but not to the body of the section). The font is then set to the paragraph font, so that more information may occur on the line with the section number and title. **.sh** insures that there is enough room to print the section head plus the beginning of a paragraph (about 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 ("_") then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.

.sx +*N* — Go to section depth *N* [**−1**], but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.

.uh *T* — Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as for **.sh**.

.$p *T B N* — Print section heading. May be redefined to get fancier headings. *T* is the title passed on the **.sh** or **.uh** line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, **.sh** passes all three, **.uh** passes only the first, and **.sx** passes three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.

.$0 *T B N* — This macro is called automatically after every call to **.$p**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similiar function. *T* is the section title for the section title which was just printed, *B* is the section number, and *N* is the section depth.

.$1 − .$6 — Traps called just before printing that depth section. May be defined to (for example) give variable spacing before sections. These macros are called from **.$p**, so if you redefine that macro you may lose this feature.

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** [10p]. Each of the definitions apply as of the *next* page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers are 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. May be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the **.he**, **.fo**, **.eh**, **.oh**, **.ef**, and **.of** requests, as well as the chapter-style title feature of **.+c**.

.$f Print footer; same comments apply as in **.$h**.

.$H A normally undefined macro 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.

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 **.ba**) is not zero); and if **C** the list is centered on a line-by-line basis. The list is set in font **\n(df** [0]. Must be matched by a **.)l**. This macro is almost like **.(b** except that no attempt is made to keep the display on one page.

.)l End list.

.(q Begin major quote. These are single spaced, filled, moved in from the text on both sides by **\n(qi** [4n], 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* 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** [0] 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 will be left-justified if *m* is **L**, indented by **\n(bi** [4n] if *m* is **I** or absent, centered (line-for-line) if *m* is **C**, and left justified to the margin (not to the base indent) if *m* is **M**. The block is set in font **\n(df** [0].

.)b End block.

.(z *m f* Begin floating keep. Like **.(b** except that the keep is *floated* to the bottom of the page or the top of the next page. Therefore, its position 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 Begin centered block. The next keep is centered as a block, rather than on a line-by-line basis as with **.(b C**. This call may be nested inside keeps.

.)c End centered block.

5. Annotations

.(d Begin delayed text. Everything in the next keep is saved for output later with **.pd**, in a manner similar to footnotes.

.)d *n* End delayed text. The delayed text number register **\n($d** and the associated string ***#** are incremented if ***#** has been referenced.

.pd Print delayed text. Everything diverted via **.(d** is printed and truncated. This might be used at the end of each chapter.

.(f Begin footnote. The text of the footnote is floated to the bottom of the page and set in font **\n(ff** [1] and size **\n(fp** [8p]. Each entry is preceeded by **\n(fs** [0.2v] space, is indented **\n(fi** [3n] on the first line, and is indented **\n(fu** [0] from the right margin. Footnotes line up underneath two columned output. If the text of the footnote will not all fit on one page it will be carried over to the next page.

.)f *n* End footnote. The number register **\n($f** and the associated string ****** are incremented if they have been referenced.

.$s The macro to output the footnote seperator. This macro may be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.

.(x *x* Begin index entry. Index entries are saved in the index *x* [**x**] until called up with **.xp.** Each entry is preceeded by a **\n(xs** [0.2v] space. Each entry is "undented" by **\n(xu** [0.5i]; this register tells how far the page number extends into the right margin.

.)x *P A* End index entry. The index entry is finished with a row of dots with *A* [null] right justified on the last line (such as for an author's name), followed by P [**\n%**]. If *A* is specified, *P* must be specified; **\n%** can be used to print the current page number. If *P* is an underscore, no page number and no row of dots are printed.

.xp *x* Print index *x* [**x**]. The index is formated in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

6. Columned Output

.2c *+S N* Enter two-column mode. The column separation is set to *+S* [4n, 0.5i in ACM mode] (saved in **\n($s**). The column width, calculated to fill the single column line length with both columns, is stored in **\n($l**. The current column is in **\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.

7. Fonts and Sizes

.sz *+P* The pointsize is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to pointsize is stored in **\n($r**. The ratio used internally by displays and annotations is stored in **\n($R** (although this is not used by **.sz**).

.r *W X* — Set *W* in roman font, appending *X* in the previous font. To append different font requests, use *X* = **\c**. If no parameters, change to roman font.

.i *W X* — Set *W* in italics, appending *X* in the previous font. If no parameters, change to italic font. Underlines in NROFF.

.b *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. 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 right 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 just surrounds *W* with double quote marks ('"'), but in TROFF uses directed quotes.

.bi *W X* — Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in NROFF. It won't work right 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 right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

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**.

.n1 — Number lines in margin from one on each page.

.n2 *N* — Number lines from *N*, stop if *N* = 0.

.sk — Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram 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, be warned: if *N* is greater than the amount of available space on an empty page, no space will ever be output.

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** [0.5v 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. If you have 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 you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z**.

10. Miscellaneous

.re — Reset tabs. Set to every 0.5i in TROFF and every 0.8i in NROFF.

.ba +*N* — Set the base indent to +*N* [0] (saved in **\n($i**). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The **.sh** request performs a **.ba** request if **\n(si** [0] is not zero, and sets the base indent to **\n(si*\n($0**.

.xl +*N* — Set the line length to *N* [6.0i]. This differs from **.ll** because it only affects the current environment.

.ll +*N* — Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-columned output. The current line length is stored in **\n($l**.

.hl — Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.

.lo — This macro loads another set of macros (in **/usr/lib/me/local.me**) which is intended to be a set of locally defined macros. These macros should all be of the form **.***X*, where *X* is any letter (upper or lower case) or digit.

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. If you want the header to have the chapter number in it, Use the string \\\\n(ch. For example, to number appendixes **A.1** etc., 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.

.+c *T*
: Begin chapter with title *T*. The chapter number is maintained in \n(ch. This register is incremented every time .+c is called with a parameter. The title and chapter number are printed by .$c. The header is moved to the footer on the first page of each chapter. If *T* is omitted, .$c is not called; this is useful for doing your own "title page" at the beginning of papers without a title page proper. .$c calls .$C as a hook so that chapter titles can be inserted into a table of contents automatically. The footnote numbering is reset to one.

.$c *T*
: Print chapter number (from \n(ch) and *T*. This macro can be redefined to your liking. 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 may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in TROFF, since it sets the page length wider than the physical width of the phototypesetter roll.

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 ('['). Extra space is left above the line to allow room for the superscript.

*]
: Unsuperscript. Inverse to *[. For example, to produce a superscript you might type x*[2*], which will produce x^2.

*<
: Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.

*>
: Inverse to *<.

***(dw** The day of the week, as a word.

***(mo** The month, as a word.

***(td** Today's date, directly printable. The date is of the form December 14, 1979. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 14), ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, December is 12), and **\n(yr** (the last two digits of the current year).

***(lq** Left quote marks. Double quote in NROFF.

***(rq** Right quote.

***—** ¾ em dash in TROFF; two hyphens in NROFF.

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, you must call the macro **.sc** to define the characters before using them.

.sc Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization.

The special characters available are listed below.

Name	Usage	Example	
Acute accent	*'	a*'	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*~	n*~	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Å
There exists	*(qe		∃
For all	*(qa		∀

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the —me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

FSCK — The UNIX File System Check Program

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The UNIX† File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. *Fsck* is frequently able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

1. INTRODUCTION

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. No changes are made to any file system by *fsck* without prior operator approval.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of heuristically sound corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. UPDATE OF THE FILE SYSTEM

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

2.1 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 super-block of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a *sync* command is issued.

† UNIX is a Trademark of Bell Telephone Laboratories.

2.2 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 upon closure[1] of the file associated with the inode.

2.3 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 one 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. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released[2] by the operating system.

2.4 Data Blocks

A data block may contain 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 whenever they have been modified and released by the operating system.

2.5 First Free-List Block

The super-block contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the super-block, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of 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 whenever they have been modified and released by the operating system.

3. CORRUPTION OF THE FILE SYSTEM

A file system can become corrupted in a variety of ways. The most common of these ways are improper shutdown procedures and hardware failures.

3.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to *sync* the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

3.2 Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

4. DETECTION AND CORRECTION OF CORRUPTION

A quiescent[3] file system may be checked for structural integrity by performing consistency

1. All in core blocks are also written to the file system upon issue of a *sync* system call.
2. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by UNIX or a *sync* command is issued.
3. I.e., unmounted and not being written on.

checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multi-pass nature of the *fsck* program.

When an inconsistency is discovered *fsck* reports the inconsistency for the operator to chose a corrective action.

Discussed in this section are how to discover inconsistencies and possible corrective actions for the super-block, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the *fsck* command under control of the operator.

4.1 Super-Block

One of the most common corrupted items is the super-block. The super-block is prone to corruption because every change to the file system's blocks or inodes modifies the super-block.

The super-block and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a *sync* command.

The super-block can be checked for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

4.1.1 File-System Size and Inode-List Size. The file-system size must be larger than 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. The file-system size and inode-list size are critical pieces of information to the *fsck* program. While there is no way to actually check these sizes, *fsck* can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

4.1.2 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 can be 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.

The first free-block list is in the super-block. *Fsck* checks the list count for a value of less than zero or greater than fifty. It also checks each block number for a value of less than the first data block in the file system or greater than the last 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 non-zero, 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.

If anything is wrong with the free-block list, then *fsck* may rebuild it, excluding all blocks in the list of allocated blocks.

4.1.3 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. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-block count.

4.1.4 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. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-inode count.

4.2 Inodes

An individual inode is not as likely to be corrupted as the super-block. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the super-block.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.2.1 Format and Type. Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types: regular inode, directory inode, special block inode, and special character inode. If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states: unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for *fsck* is to clear the inode.

4.2.2 Link Count. Contained in each inode is a count of the total number of directory entries linked to the inode.

Fsck verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, calculating an actual link count for each inode.

If the stored link count is non-zero and the actual link count is zero, it means that no directory entry appears for the inode. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is non-zero and the actual link count is zero, *fsck* may link the disconnected file to the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, *fsck* may replace the stored link count by the actual link count.

4.2.3 Duplicate Blocks. Contained in each inode is 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 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* will make a partial second pass of the inode list to find the inode of the duplicated block, because without examining the files associated with these inodes for correct content, there is not enough information available to decide which inode is corrupted and should be cleared. Most times, the inode with the earliest modify time is incorrect, and should be cleared.

This condition can occur by using a file system with blocks claimed by both the free-block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.4 Bad Blocks. Contained in each inode is 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 lower than that of the first data block, or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.5 Size Checks. Each inode contains a thirty-two bit (four-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of sixteen characters, or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the UNIX file system has the directory bit on in the inode mode word. The directory size must be a multiple of sixteen because a directory entry contains sixteen bytes of information (two bytes for the inode number and fourteen bytes for the file or directory name).

Fsck will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in a inode by the number of characters per block (512) and rounding up. *Fsck* adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, *fsck* will warn of a possible file-size error. This is only a warning because UNIX does not fill in blocks in files created in random order.

4.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, apply iteratively Sections 4.2.3 and 4.2.4 to each level of indirect blocks.

4.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be 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 "." and "..", and directories which are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, then *fsck* may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written to the file system while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, *fsck* may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to 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* may replace them by the correct values.

Fsck checks the general connectivity of the file system. If directories are found not to be linked into the file system, *fsck* will link 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 with the corresponding directory data blocks not being written to the file system.

4.5 Free-List Blocks

Free-list blocks are owned by the super-block. Therefore, inconsistencies in free-list blocks directly affect the super-block.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks see Section 4.1.2.

ACKNOWLEDGEMENT

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS.

REFERENCES

[1] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.

[2] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).

[3] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

Appendix—FSCK ERROR CONDITIONS

1. CONVENTIONS

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the free-block list (possibly rebuilding it), and performs some cleanup.

When an inconsistency is detected, *fsck* reports the error condition 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 can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

2. INITIALIZATION

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section 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 to *fsck*; legal options are −y, −n, −s, −S, and −t. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

Bad −t option

The −t option is not followed by a file name. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

Invalid −s argument, defaults assumed

The −s option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. *Fsck* assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-skip. See the *fsck*(1M) manual entry for more details.

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. See the *fsck*(1M) manual entry for further detail.

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 a guru.

Can't open checklist file: F

The default file system checklist file **F** (usually */etc/checklist*) can not 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 a guru.

Can't stat F

Fsck's request for statistics about the file system **F** failed. It 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

You have given *fsck* a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of **F**.

Can't open F

The file system **F** can not be opened for reading. It 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. It ignores this file system and continues checking the next file system given. See Section 4.1.1.

Can't create F

Fsck's request to create a scratch file **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

CAN NOT 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 a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT READ: BLK B (CONTINUE)

Fsck's request for reading a specified block number **B** in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT 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 a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

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, special character inode, regular inode, or directory inode. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

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 will not allow a complete check of the file system. A second run of *fsck* should be made to re-check 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 may invoke 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 will always invoke the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.4.

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. See Section 4.2.4.

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 will not allow a complete check of the file system. A second run of *fsck* should be made to re-check 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 may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode I has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.3.

EXCESSIVE DUP BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See Section 4.2.3.

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 the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check 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 will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.
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. See Section 4.2.5.

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. See Section 4.2.5.

PARTIALLY ALLOCATED INODE I=I (CLEAR)

Inode I is neither allocated nor unallocated. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.
NO ignore this error condition.

4. PHASE 1B: RESCAN FOR MORE DUPS

When a duplicate block is found in the file system, the file system is rescanned to find the inode which 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 will always invoke the BAD/DUP error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1. See Section 4.2.3.

5. PHASE 2: CHECK PATH-NAMES

This phase concerns itself with removing directory entries pointing to error conditioned inodes 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 will terminate. See Section 4.2.1.

ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type. See Section 4.2.1.

Possible responses to the FIX prompt are:

- YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.
- NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system. See Section 4.2.3 and 4.2.4.

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, then this may result 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. See Section 4.4.

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. See Section 4.4.

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 DIR=F (REMOVE)

Phase 1 or Phase 1b have 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. See Section 4.2.3 and 4.2.4.

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. See Section 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES	the directory entry **F** is removed.
NO	ignore this error condition.

6. PHASE 3: CHECK CONNECTIVITY

This phase concerns itself with 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 traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of directory inode **I** are printed. See Section 4.4 and 4.2.2.

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 may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode **I** to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.
NO	ignore this error condition. This will always invoke 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 will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck*(1M) manual entry for further detail.

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 will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck*(1M) manual entry for further detail.

DIR I=I1 CONNECTED. PARENT WAS I=I2

This is an advisory message indicating a directory inode **I1** was 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. See Section 4.4 and 4.2.2.

7. PHASE 4: CHECK REFERENCE COUNTS

This phase concerns itself with 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 traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Section 4.2.2.

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 may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode **I** to *lost+found*.
- NO ignore this error condition. This will always invoke 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 will always invoke 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 will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*.

(CLEAR)

The inode mentioned in the immediately previous error condition can not be reconnected. See Section 4.2.2.

Possible responses to the CLEAR prompt are:

- YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.
- NO ignore this error condition.

LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for inode I which is a file, is X but should be Y. The owner O, mode M, size S, and modify time T are printed. See Section 4.2.2.

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 SHOULD BE Y (ADJUST)

The link count for inode I which is a directory, is X but should be Y. The owner O, mode M, size S, and modify time T of directory inode I are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode I with Y.
NO ignore this error condition.

LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

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. See Section 4.2.2.

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)

Inode I which is a file, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.
NO ignore this error condition.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Inode I which is a directory, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES de-allocate 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 **I**. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
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. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
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. See Section 4.1.4.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.
NO ignore this error condition.

8. PHASE 5: CHECK FREE LIST

This phase concerns itself with the free-block 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 the total free-block count incorrect.

EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See Section 4.1.2 and 4.2.4.

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 will always invoke 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. See Section 4.1.2 and 4.2.3.

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 will always invoke 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 will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

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 will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.4.

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 will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.3.

X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

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. See Section 4.1.3.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by 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. See Section 4.1.2, 4.2.3, and 4.2.4.

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.

9. PHASE 6: SALVAGE FREE LIST

This phase concerns itself with 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 9 blocks-to-skip and 400 blocks-per-cylinder are used. See the *fsck*(1M) manual entry for further detail.

10. CLEANUP

Once a file system has been checked, a few 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 UNIX (NO SYNC!) *******

This is an advisory message indicating that a mounted file system or the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX 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* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

INDEX OF MESSAGES
(Alphabetically within each section)

INITIALIZATION

Bad −t option . . . 7
C option? . . . 7
CAN NOT READ: BLK B (CONTINUE) . . . 8
CAN NOT SEEK: BLK B (CONTINUE) . . . 8
CAN NOT WRITE: BLK B (CONTINUE) . . . 9
Can't create F . . . 8
Can't get memory . . . 7
Can't open checklist file: F . . . 7
Can't open F . . . 8
Can't stat F . . . 8
Can't stat root . . . 7
F is not a block or character device . . . 8
Incompatible options: −n and −s . . . 7
Invalid −s argument, defaults assumed . . . 7
Size check: fsize X isize Y . . . 8

PHASE 1: CHECK BLOCKS AND SIZES

B BAD I=I . . . 9
B DUP I=I . . . 10
DIRECTORY MISALIGNED I=I . . . 10
DUP TABLE OVERFLOW (CONTINUE) . . . 10
EXCESSIVE BAD BLKS I=I (CONTINUE) . . . 10
EXCESSIVE DUP BLKS I=I (CONTINUE) . . . 10
LINK COUNT TABLE OVERFLOW (CONTINUE) . . . 9
PARTIALLY ALLOCATED INODE I=I (CLEAR) . . . 10
POSSIBLE FILE SIZE ERROR I=I . . . 10
UNKNOWN FILE TYPE I=I (CLEAR) . . . 9

PHASE 1B: RESCAN FOR MORE DUPS

B DUP I=I . . . 11

PHASE 2: CHECK PATH-NAMES

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE) . . . 12
DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE) . . . 12
DUPS/BAD IN ROOT INODE (CONTINUE) . . . 11
I OUT OF RANGE I=I NAME=F (REMOVE) . . . 11
ROOT INODE NOT DIRECTORY (FIX) . . . 11
ROOT INODE UNALLOCATED. TERMINATING. . . . 11
UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE) . . . 12

PHASE 3: CHECK CONNECTIVITY

DIR I=I1 CONNECTED. PARENT WAS I=I2 . . . 13
SORRY. NO SPACE IN lost+found DIRECTORY . . . 13
SORRY. NO lost+found DIRECTORY . . . 12
UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT) . . . 12

PHASE 4: CHECK REFERENCE COUNTS

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . 15
BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . 15
(CLEAR) . . . 13
FREE INODE COUNT WRONG IN SUPERBLK (FIX) . . . 15
LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . 14
LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . 14
LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . 14
SORRY. NO SPACE IN lost+found DIRECTORY . . . 13
SORRY. NO lost+found DIRECTORY . . . 13
UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . 14
UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . 14
UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT) . . . 13

PHASE 5: CHECK FREE LIST

BAD FREE LIST (SALVAGE) . . . 16
BAD FREEBLK COUNT . . . 16
EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE) . . . 15
EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE) . . . 16
FREE BLK COUNT WRONG IN SUPERBLOCK (FIX) . . . 16
X BAD BLKS IN FREE LIST . . . 16
X BLK(S) MISSING . . . 16
X DUP BLKS IN FREE LIST . . . 16

PHASE 6: SALVAGE FREE LIST

Default free-block list spacing assumed . . . 17

CLEANUP

***** BOOT UNIX (NO SYNC!) ***** . . . 17
***** FILE SYSTEM WAS MODIFIED ***** . . . 17
X files Y blocks Z free . . . 17