MINIX 1.5 REFERENCE MANUAL

ANDREW S. TANENBAUM
FRANS MEULENBROEKS
RAYMOND MICHIELS
JOST MÜLLER
JOSEPH PICKERT
STEVEN REIZ
JOHAN W. STEVENSON

PRENTICE HALL, ENGLEWOOD CLIFFS, NEW JERSEY 07632

© 1991 by **PRENTICE-HALL, INC.**
A Division of Simon & Schuster
Englewood Cliffs, N.J. 07632

All rights reserved.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

1 **INTRODUCTION** 1

1.1 HISTORY OF UNIX 1
1.2 HISTORY OF MINIX 3
1.3 STRUCTURE OF THIS MANUAL 5

2 **INSTALLING MINIX ON THE IBM PC, XT, AT, 386, AND PS/2** 6

2.1 MINIX HARDWARE REQUIREMENTS 6
2.2 HOW TO START MINIX 7
2.3 HOW TO INSTALL MINIX ON A HARD DISK 10
2.4 TESTING MINIX 20
2.5 TROUBLESHOOTING 22

3 **INSTALLING MINIX ON THE ATARI ST** 24

3.1 THE MINIX-ST DISTRIBUTION 25
3.2 NATIONAL KEYBOARDS 26
3.3 BOOTING MINIX-ST 27
3.4 INCREASING THE SIZE OF YOUR RAM DISK 30
3.5 ADAPTING PROGRAMS TO USE EXTRA RAM 31
3.6 USING SINGLE-SIDED DISKETTES 32
3.7 USING A HARD DISK 33

3.8 USING A MEGA ST 40
3.9 USING A DISK CONTROLLER BASED CLOCK 40
3.10 BOOT PROCEDURE OPTIONS 41
3.11 UNPACKING THE SOURCES 42
3.12 THE TOS TOOLS 43
3.13 TROUBLESHOOTING 45

4 INSTALLING MINIX ON THE COMMODORE AMIGA 51

4.1 MINIX HARDWARE REQUIREMENTS 51
4.2 HOW TO START MINIX 52
4.3 A MORE DETAILED LOOK 54
4.4 TROUBLESHOOTING 58

5 INSTALLING MINIX ON THE APPLE MACINTOSH 59

5.1 MACMINIX HARDWARE REQUIREMENTS 59
5.2 THE MACMINIX DISTRIBUTION 59
5.3 NATIONAL KEYBOARDS 60
5.4 BOOTING MACMINIX 60
5.5 INCREASING THE SIZE OF YOUR RAM DISK 63
5.6 ADAPTING PROGRAMS TO USE EXTRA RAM 64
5.7 USING A HARD DISK 65
5.8 UNPACKING THE SOURCES 69
5.9 THE MENUS 70
5.10 SETTING CONFIGURATION OPTIONS 71
5.11 MACINTOSH SYSTEM CALLS 72
5.12 RUNNING MACMINIX WITH MULTIFINDER 72
5.13 TROUBLESHOOTING 73

6 USING MINIX 74

6.1 MAJOR COMPONENTS OF MINIX 74

6.2 PROCESSES AND FILES IN MINIX 79

6.3 A TOUR THROUGH THE MINIX FILE SYSTEM 84

6.4 HELPFUL HINTS 88

6.5 SYSTEM ADMINISTRATION 93

7 RECOMPILING MINIX 97

7.1 REBUILDING MINIX ON AN IBM PC 97

7.2 REBUILDING MINIX ON AN ATARI ST 103

7.3 REBUILDING MINIX ON A COMMODORE AMIGA 109

7.4 REBUILDING MINIX ON AN APPLE MACINTOSH 109

8 MANUAL PAGES 115

9 EXTENDED MANUAL PAGES 189

9.1 ASLD—ASSEMBLER-LOADER [IBM] 189

9.2 BAWK—BASIC AWK 198

9.3 DE—DISK EDITOR 202

9.4 DIS88—DISASSEMBLER FOR THE 8088 [IBM] 207

9.5 ELLE—FULL-SCREEN EDITOR 208

9.6 ELVIS—A CLONE OF THE BERKELEY VI EDITOR 216

9.7 IC—INTEGER CALCULATOR 236

9.8 INDENT—INDENT AND FORMAT C PROGRAMS 239

9.9 KERMIT—A FILE TRANSFER PROGRAM 243

9.10 M4—MACRO PROCESSOR 246

9.11 MDB—MINIX DEBUGGER [68000] 249

9.12 MINED—A SIMPLE SCREEN EDITOR 253

9.13 NROFF—A TEXT PROCESSOR 257

9.14 PATCH—A PROGRAM FOR APPLYING DIFF LISTINGS TO UPDATE FILES
9.15 ZMODEM—FILE TRANSFER PROGRAM 269

10 SYSTEM CALLS 274

10.1 INTRODUCTION TO SYSTEM CALLS 274
10.2 LIST OF MINIX SYSTEM CALLS 275

11 NETWORKING 277

11.1 INTRODUCTION 277
11.2 OBJECTS 279
11.3 OVERVIEW OF TRANSACTIONS 281
11.4 SYNTAX AND SEMANTICS OF TRANSACTION PRIMITIVES 282
11.5 SERVER STRUCTURE 286
11.6 CLIENT STRUCTURE 287
11.7 SIGNAL HANDLING 287
11.8 IMPLEMENTATION OF TRANSACTIONS IN MINIX 288
11.9 COMPILING THE SYSTEM 289
11.10 HOW TO INSTALL AMOEBA 289
11.11 NETWORKING UTILITIES 290
11.12 REMOTE SHELL 290
11.13 SHERVERS 292
11.14 MASTERS 292
11.15 FILE TRANSFER 293
11.16 REMOTE PIPES 293
11.17 THE ETHERNET INTERFACE 293

A MINIX SOURCE CODE LISTING 296

B CROSS REFERENCE MAP 637

NOTE

The only pieces of MINIX software not included in this package are the C compiler sources. These were made using the Amsterdam Compiler Kit (see *Communications of the ACM*, Sept. 1983, pp. 654-660), which has also been used to make compilers for many other languages and machines. The compiler *sources* can be ordered from the following companies:

In North and South America:

UniPress Software
2025 Lincoln Highway
Edison, NJ 08817
U.S.A.
Telephone: (201) 985-8000
FAX: (201) 287-4929

In Europe and elsewhere

Transmediair Utrecht BV
Melkweg 3
3721 RG Bilthoven
Holland
Telephone: +31 30 281820
FAX: +31 30 292294

In addition, these companies also have a Pascal compiler for MINIX

1

INTRODUCTION

Every computer needs an operating system to manage its memory, control its I/O devices, implement its file system and provide an interface to its users. Many operating systems exist, such as MS-DOS, OS/2, and UNIX. This manual describes yet another operating system called MINIX. Although MINIX is entirely new, it was inspired by UNIX, and has many features in common with UNIX. For this reason it is fitting that we begin this introduction with a brief history of UNIX, and its ancestors. This will be followed by an equally brief history of MINIX. Finally, the chapter concludes with a summary of the rest of the manual.

1.1. HISTORY OF UNIX

Prior to 1950, all computers were personal computers. At least in the sense that only one person could use a computer at a time. Only research laboratories owned computers in those days. The usual method of operation was for the programmer to sign up for an hour of machine time on a sign-up sheet posted on the bulletin board. When his time came, the programmer chased everybody else out of the machine room and went to work.

During the 1950s, **batch systems** were invented. With a batch system, the programmers punched their programs onto 80-column cards, and deposited them in a tray in the computer room. Every half hour or so, the computer operator would go

to the tray, pick up a batch of jobs, and read them in. The printed output was brought back later for the programmers to collect at their leisure.

While the batch system made more efficient use of the computer, it was not wildly popular with programmers, who often lost hours due to a single typing error that caused their compilations to fail. In the early 1960s, researchers at Dartmouth College and M.I.T. devised **timesharing systems**, in which many users could log into the same computer at once from remote terminals. The Dartmouth project led to the development of BASIC; the M.I.T. project was the great grandfather of MINIX.

The M.I.T. system, called **CTSS (Compatible Time Sharing System)** because it was more-or-less compatible with the batch system then in use at M.I.T., was a success beyond the wildest expectations of its designers. They quickly decided to build a new and much more powerful timesharing system on what was then one of the largest computers in the world, the GE 645, a machine about as fast as a modern PC/AT. This project, called **MULTICS (MULTiplexed Information and Computing Service)** was a joint effort of M.I.T., General Electric (then a computer vendor), and AT&T Bell Labs.

To make a long story short, the project was so ambitious that the system was difficult to program. Bell Labs eventually pulled out and GE sold its computer business to Honeywell. M.I.T. and Honeywell went on to complete MULTICS, and it ran at many installations for 25 years. In a way, it continues to live, since OS/2 has many ideas and features taken straight from MULTICS .

Meanwhile, one of the Bell Labs researchers who had worked on MULTICS, Ken Thompson, was hunting around for something interesting to do next. He spied an old PDP-7 minicomputer at Bell Labs that nobody was using, and decided to implement a stripped down version MULTICS on his own on this tiny minicomputer. The system he produced, while clearly not full MULTICS, did work and supported one user (Thompson). One of the other people at the Labs, Brian Kernighan, somewhat jokingly called it **UNICS (UNiplexed Information and Computing Service)**. Bell Labs management was so impressed that they bought Thompson a more modern minicomputer, a PDP-11, to continue his work.

The initial implementation was in PDP-7 assembly code. When this had to be completely rewritten for the PDP-11, it became clear that it would be much better to write the system in a high level language to make it portable. At this point, another Bell Labs Researcher, Dennis Ritchie, designed and implemented a new language called C in which the two of them rewrote the system, whose spelling had now changed to UNIX.

In 1974, Ritchie and Thompson wrote a now-classic paper about UNIX that attracted a great deal of attention. Many universities asked them if they could have a copy of the system, including all the source code. AT&T agreed. Within a few years, UNIX had become established as the de facto standard in hundreds of university departments around the world. The source code was widely available, and often studied in university courses. International meetings were organized, in

which speakers would get up and describe how they had modified some system routine to be a bit fancier. UNIX had by now become something of a cult item, with numerous fanatically devoted followers.

As UNIX kept spreading, AT&T began to see how valuable it was and began restricting access to the source code of new versions, starting with the Seventh Edition (usually referred to as **V7**). It was no longer permitted for universities to teach courses using the source code as an example, and public discussions of the internal workings of the code were severely restricted.

Nevertheless, UNIX' fame continued to spread. The University of California at Berkeley got a contract to port V7 it to the VAX, adding virtual memory and numerous other features in the process. This work led to **4.x BSD**. AT&T itself modified V7 extensively, leading to **System V**. It took a decade before these two versions could be partially reconciled, under the auspices of the IEEE, which led to the **POSIX** standard.

Although UNIX was now more popular, without the source code, it was also a lot less fun, especially for the people whose initial enthusiasm had made it a big success. The time was ripe for a new system.

1.2. HISTORY OF MINIX

Many university professors regretted that UNIX could no longer be taught in operating systems courses, but there appeared to be no choice. One of them, Andrew Tanenbaum, at the Vrije Universiteit in Amsterdam, The Netherlands, went out and bought an IBM PC, and like Ken Thompson a decade earlier, set out to write a new operating system from scratch. Just as Thompson was inspired by MULTICS, but ultimately wrote a new and much smaller operating system, Tanenbaum was inspired by UNIX, but ultimately also wrote a new and much smaller operating system—MINIX—which stands for Mini-UNIX.

Since MINIX contains no AT&T code whatsoever, it falls outside the AT&T licensing restrictions. The source code has been made widely available to universities for study in courses and otherwise. Like UNIX, MINIX quickly acquired an enthusiastic following and began to occupy the same niche that UNIX had filled in its early days—a small operating system available with all the source code that people could study and modify as they wished.

Within a month of its release (Jan. 1987), there was already so much interest in MINIX worldwide, that a news group (comp.os.minix) was set up on USENET, a computer network accessible to most universities and computer companies in North America, Europe, Japan, Australia, and elsewhere. A few months later, the news group had over 10,000 people reading and contributing to it.

The initial release of MINIX was for the IBM PC and PC/AT only. It did not take long before people with other kinds of computers began thinking about porting it to their machines. The first port was to a 68000-based machine, the Atari ST,

done primarily by Johan Stevenson, with assistance from Jost Müller The hard part was making MINIX, which, like UNIX, allows multiple processes to run simultaneously, run on a bare 68000, with no memory management or relocation hardware. Once this problem had been solved and new I/O device drivers were written for the Atari's keyboard, screen, disk, etc., MINIX-ST became a reality. The 1.5 Atari version was prepared by Frans Meulenbroeks.

The port to the Commodore Amiga was done by Raymond Michiels and Steven Reiz. The Macintosh port was done by Joseph Pickert. Unlike all the other ports, the Macintosh version does not replace the manufacturer's operating system and run on the bare machine. Instead, it runs on top of the Macintosh operating system, allowing the facilities of both systems to be used simultaneously. Of course, there is a price to to paid, in terms of both size and performance.

The contribution of USENET cannot be underestimated in the MINIX development. Hundreds of individuals have donated ideas, bug fixes, and software, many of which are included in this release. One person stands out above all the others though, Bruce Evans, who has produced improvements too numerous to count, and all of them in a very professional way. The authors are credited in the code for their contributions, where that is technically feasible.

MINIX is still a vigorous on-going development, with new software, ports to new machines (e.g. the SPARC), and many other activities in progress. In this respect, MINIX is very different from most software products. The usual model is that a company gets an idea for a product, writes the software, and then sells executable binaries for customers to use. Users are not able to modify the program, and requests for the source code are rejected out of hand. Worldwide public discussion of the system internals is not welcome.

MINIX, in contrast takes a more open approach. The source code for the entire operating system is included in the basic software package, and users are encouraged to tailor the system to their own specific needs. Thousands of people on USENET have done just that. The internal workings of the system are described in detail in the following book:

Title: Operating Systems: Design and Implementation
Author: Andrew S. Tanenbaum
Publisher: Prentice Hall
ISBN: 0-13-637406-9

However, please note that the book describes a somewhat earlier version of the system. Nevertheless, the basic principles are still intact in this version.

MINIX was originally written to be system call compatible with V7 UNIX, the last of the small UNICES. This is the version described in the book.

Future versions of MINIX will migrate towards at least partial conformance with the ANSI C and IEEE POSIX standards. Complete conformance is unlikely, as conformant systems are necessarily so huge that no one person can possibly understand them. Making MINIX fully conformant would defeat the goal of having a

system that is small enough that people can actually understand it. It would also require much larger and more expensive hardware than is currently the case. MINIX 1.5 is an intermediate form: it still supports the V7 system calls and has a Kernighan and Ritchie C compiler, but virtually all the individual files are ANSI C conformant (and also K&R conformant). Furthermore, all of the header files provided in */usr/include* conform to both the ANSI and POSIX standards. This makes porting programs from conformant systems to MINIX 1.5 and vice versa much easier.

1.3. STRUCTURE OF THIS MANUAL

Chapters 2 through 5 tell how to install MINIX on the IBM PC, Atari, Amiga, and Macintosh, respectively. You should read the one appropriate for your machine. Then skip to Chap. 6 to learn more about how to use MINIX.

For people interested in modifying the operating system itself, Chap. 7 has been provided. It discusses things needed by people who want to recompile the system. If you do not plan to do this (initially), you may safely skip this chapter for the time being.

Chapter 8 contains the manual pages for the commands that come with MINIX. Each entry describes one utility program (or sometimes several closely related ones). Every MINIX user, even though intimately familiar with UNIX, should read Chap. 8 very carefully. Some of the commands have extended manual pages. These are present in Chap. 9.

Chapter 10 discusses the MINIX system calls. The treatment here is brief, since most of the system calls should be familiar to UNIX users.

Chapter 11 contains information about networking in MINIX.

Appendix A contains a (nearly complete) listing of the MINIX 1.5 operating system code. The kernel listing is for the Intel (IBM) line, but the file system and memory manager are identical for all versions, as is the general structure of the kernel. Parts of the kernel, especially the I/O device drivers, are different for each version. Most of these machine-dependent parts (mostly deice drivers) have not been listed here, but since the sources are present on the disks, you can easily make your own listing of the missing pieces. The MINIX program *mref* can be used, for example. Appendix B is a cross reference map of the source code listing.

One final note. There is a large MINIX user group on USENET with over 16,000 people. It is called *comp.os.minix*. Thousands of messages have been posted to this group dealing with bugs, improvements, suggestions, and new software. Unfortunately, USENET is not a public network, so one cannot just join, but over a million people are on it, mostly at universities and companies in the computer industry in the U.S., Canada, Europe, Japan, Australia, and other countries. If you are interested in following all the latest developments on the MINIX front, and there are many in the works (e.g., ANSI and POSIX conformance), it is worthwhile trying to find someone who has access.

2

MINIX ON THE IBM PC, XT, AT, 386, AND PS/2

This chapter tells you how to install and run MINIX on a computer powered by the Intel 8088, 80286, 80386 or related chips. If you have an Atari, Amiga, or Macintosh, you should skip this chapter and go directly to chapter 3, 4, or 5, respectively.

Five sections are present in this chapter. The first section discusses the kind of hardware you need to run MINIX. The second section tells you how to boot your computer to get MINIX running. The third one tells you how to install MINIX on your hard disk. The fourth one is about testing MINIX The final one gives hints about troubleshooting. If during the installation you have problems, please check the troubleshooting section to see if your problem is discussed there. When you have finished reading this chapter and have successfully installed MINIX, please skip to Chap. 6 to learn about using your newly installed MINIX system.

2.1. MINIX HARDWARE REQUIREMENTS

MINIX will run on the original IBM PC, XT, and AT, and on all other machines that are 100 percent hardware compatible with one of these machines. This point deserves some explanation. Many manufacturers have brought out machines that are similar to their IBM counterparts in some ways, but different in other ways. MINIX will not necessarily run on all these machines. Like all versions of UNIX for

the IBM PC, MINIX normally does not use the BIOS (because the BIOS is not interrupt-driven, making it unsuitable for timesharing). Instead, it programs all the I/O chips directly. Therefore it will only run on machines using the same I/O chips as the IBM PC. MINIX will not run on a machine whose manufacturer says that it is "IBM compatible" or "MS-DOS compatible" unless the *hardware* is IBM compatible; having different hardware but masking this in the BIOS will not work since MINIX does not use the BIOS (with one exception).

Experience has shown that most problems occur in two areas: video boards and hard disks. MINIX supports the original IBM PC monochrome display interface, the CGA interface, the Hercules interface, and the EGA interface. The EGA support is done in software and is enabled by hitting the F3 function key. If you have an EGA board and the screen goes blank periodically, hitting the F3 key will solve the problem by enabling software scrolling. Hitting it again will disable it. Other video boards can be used, provided that they accurately and completely emulate one of the above interfaces.

The other problem area is the hard disk controller. IBM used different (and incompatible) controllers in the XT and AT. Both of these are supported in MINIX. In fact, the only difference between the PC Boot Diskette (#1) and the AT Boot Diskette (#2) is the hard disk controller used. As an emergency measures, the Universal Boot Diskette (#3) is also provided. Unlike the other ones, this one uses the BIOS for disk I/O. It is slow and cannot run in protected mode, but should work on most machines.

MINIX requires a minimum of 512K RAM to work. However, if more RAM is available, it can and will be used, up to a maximum of 16M on the 80286 and 80386.

Up to two RS-232 ports are supported. They may be used for either additional terminals, to run MINIX as a multiuser timesharing system, or to connect to modems, printers, or other devices. In addition, one Centronics parallel port is supported.

2.2. HOW TO START MINIX

Before running MINIX for the first time, make a backup of all the diskettes, to prevent disaster if one of them should be subsequently damaged. They are not copy protected. However, they are also not MS-DOS disks, so do not use a copy program that expects that. Instead use one that copies entire disks, sector by sector.

Throughout the discussion below, lines printed in the Helvetica typeface are either commands you should type on the keyboard, or are lines that the computer will display for you. In a few of the examples, *italics* characters or words appear in a command. These represent values that you are to fill in.

To boot MINIX, proceed as follows. If you make a typing error, use the backspace key to erase it, or the @ sign to erase the current line.

1. Choose one of the various boot diskettes that come with MINIX. If your machine uses the standard hard disk present on the original IBM PC/XT, try the PC Boot Diskette (#1) first. If your machine uses the standard hard disk present on the original IBM PC/AT, try the AT Boot Diskette (#2) first. If you have a nonstandard hard disk or no hard disk at all, use the Universal Boot Diskette (#3). If you have a PS/2 with a Microchannel and an ST-506 disk interface, try the PS/2 Boot Diskette (#12, 3.5 inch only). With an ESDI drive, you have to use #3. In general, it is better to avoid #3 if possible, since the others run in protected mode (on 80286 and 80386 CPUs) and are faster. If you have problems, boot on a friend's machine and try recompiling the system with the various drivers in the kernel directory. One might work. There are drivers there for which no boot diskette is provided.

2. Turn off the PC, then insert the chosen boot diskette in drive 0, and then turn on the PC. You can also type CRTL-ALT-DEL to boot a running PC. However, if that fails, turn the power off and then on again.

3. You should get a message like: "Booting MINIX 1.5" as soon as the power-on self-tests have finished.

4. After the operating system has been completely read in, you will get a menu on the screen offering you several options. Remove the boot diskette from drive 0, insert the root file system (diskette #4) in its place, and hit the = (equal sign) key (or one of the others, depending on your keyboard). If you get a message about an invalid root file system, you probably forget to put the root file system diskette in. Turn off the power and start again.

5. MINIX will now erase the screen and display a line at the top telling how much memory the machine has, how large the operating system (including all its tables and buffers) is, how large the RAM disk is, and how much memory is available for user programs (the first number minus the next two). Check to see that the available memory is at least positive. MINIX will not run with negative memory. To do anything useful, however, at least 200K of available memory is needed.

6. Now the root file system will be copied from drive 0 to the RAM disk. The MINIX root device is initially on the RAM disk, but later you can put it on the hard disk.

7. When the RAM disk has been loaded, the system initialization file, */etc/rc*, is executed. It asks you to remove the root file system and then insert the */usr* file system (diskette #5) in drive 0 and type a carriage return. Do so.

8. After */usr* has been mounted, you may be requested to enter the date and time of day. If so, please enter a 12-digit number in the form MMDDYYhhmmss, followed by a carriage return. For example, 3:35 p.m. on July 4, 1976 was 070476153500. (If the system is able to locate a suitable real-time clock, it will not ask.)

9. You will now get the message:

   ```
   login:
   ```

   on the screen. Type:

   ```
   ast
   ```

   and wait for the system to ask for your password. Then type:

   ```
   Wachtwoord
   ```

   being careful to type the first letter in upper case. Lower and upper case letters are always distinct in MINIX. Do not use an upper case letter when a lower case one is called for or vice versa. Like UNIX, MINIX regards "a" and "A" as two distinct characters.

10. If you have successfully logged in, the shell will display a prompt (dollar sign) on the screen. Try typing:

    ```
    ls –l /bin
    ```

    to see what is in the *bin* directory. Note that there is a space after *ls*. In all commands given below, be sure to insert spaces where they are shown in the text. Then type:

    ```
    ls –l /usr/bin
    ```

    to see what is on the drive 0 diskette. To stop the display from scrolling out of view, type CTRL-S; to restart it, type CTRL-Q. (Note that CTRL-S means depress the "control" key on the keyboard and then hit the *S* key while "control" is still depressed.)

11. If you have two diskette drives, you can mount one of the other diskettes by inserting it into drive 1 and typing:

    ```
    /etc/mount /dev/fd1 /user
    ```

    Use *ls* to inspect */user*. You can now try out other commands. However there is so little free disk space available at this point, that once you are convinced that MINIX works, it is best to start thinking about installing MINIX on your hard disk.

12. When you are finished working, type:

    ```
    sync
    ```

    and then CTRL-D to log out. The

    ```
    login:
    ```

    prompt will be displayed. If you want to shut the computer down, make sure all processes have finished, if need be, by killing them with *kill*, before issuing the *sync* command. When the disk light goes out, you can turn the computer power off. *Never*, *ever* turn the system off without first running *sync*. Failure to obey this rule will generally result in a garbled file system and lost data. If you forget and just turn off the computer, next time you boot, be sure to run *fsck* to repair the file system.

2.3. HOW TO INSTALL MINIX ON A HARD DISK

Many MINIX users have a hard disk. This section describes how to set up MINIX on such a system. If you do not have a hard disk, you may skip this section.

2.3.1. Step 1: Backup the Hard Disk

If you are already using your hard disk for another operating system, such as MS-DOS or XENIX, before even *contemplating* installing MINIX, you should make a complete backup of the contents of your hard disk onto diskette or another medium. As a bare minimum, installing MINIX will require erasing one partition of your hard disk, and possibly two. However, to prevent disaster in the event that you make an error during the setup procedure, it is highly desirable that you backup the *entire* disk before you even start. Your files are too valuable to put at risk.

It is worth noting that MINIX has a program, *dosread*, that can read MS-DOS diskettes. Thus if you make your backup on diskettes, you will be able to read the files into the MINIX file system after you have completed the hard disk installation.

2.3.2. Step 2: Verify that Your Hard Disk is IBM Compatible

The vendors of all MS-DOS machines claim that their machines are IBM compatible. Unfortunately, some machines have different hardware than their IBM counterparts, with these differences being compensated for by the BIOS. Since MINIX does not normally use the BIOS, but, like XENIX, directly controls the I/O chips, it may not work on some of these machines. The device most prone to compatibility problems is the hard disk controller.

To verify that MINIX is indeed able to correctly access your hard disk, boot MINIX as described above, but instead of logging in as *ast*, log in as *root*, using *Geheim* as password (note the upper case *G*). If you are already logged in as *ast*, use CTRL-D to log out, then log in again as *root* (without rebooting). Logging in as *root* makes you the superuser and gives you the sharp sign (#) as prompt instead of the usual dollar sign. The superuser is the system administrator and has special privileges denied ordinary users. To install MINIX on your hard disk, you will need these privileges. Once the installation is complete, you should always log in as *ast*, or create your own login name as described later in this manual.

Once you are successfully logged in as *root*, type:

```
dd if=/dev/hd0 of=/dev/null count=200
```

After a short time, you should get the message:

```
200+0 records in
200+0 records out
```

If you get an error message instead, MINIX cannot use your hard disk controller. Please reboot with one of the other MINIX boot diskettes. If none of them work, your hard disk controller is not compatible with any of those used by IBM, unless you have a PS/2 Model 30/286, in which case you should try installing MINIX on a friend's machine, and then rebuilding the kernel after first copying *ps_wini.c* to *wini.c*. Compile it using **–DPS30_286**.

2.3.3. Step 3: Partition the Hard Disk

Hard disks may be divided up into sections called **partitions**. MINIX supports disks with up to four partitions, some of which may be allocated to MS-DOS or other operating systems. To discover how your hard disk is currently partitioned, log into MINIX as *root* and type:

```
fdisk –hm –sn /dev/hd0
```

where *m* is the number of heads the hard disk has and *n* is the number of sectors per track. For example:

```
fdisk –h4 –s17 /dev/hd0
```

Note that the values follow the *h* and *s* flags directly, with no spaces in between. If you omit either the *h* or *s* flags, the default values of 4 and 17, respectively, are used.

If you are not sure how many heads and sectors your disk has, examine the data sheet that came with the disk, ask your computer dealer or make a guess. For disks up to 40M, 4 heads and 17 sectors are common. For larger disks, 9 heads and 17 sectors are common. For RLL disks, 25 or 26 sectors are usual, depending on the controller. If you guess wrong, *fdisk* will complain and tell you what it thinks the

parameters are. Usually its opinion is worth listening to. Type a *q* to quit *fdisk* and try again with the new parameters.

When *fdisk* is called with the correct number of heads and sectors, it gives a display with one line for each partition, each line containing 12 or 13 columns. For example, a disk with four MINIX partitions might show:

				------first------			-------last------			----------sectors----------		
#	Sorted	Active	Type	Cyl	Head	Sec	Cyl	Head	Sec	Base	Last	Size
1	1		MINIX	0	0	2	418	8	17	1+	64106	64106
2	2		MINIX	419	0	1	837	8	16	64107+	128212	64106
3	3		MINIX	838	0	1	905	8	17	128214	138617	10404
4	4		MINIX	906	0	1	1023	8	17	138618	156671	18054

(Enter 'h' for help. A null line will abort any operation)

To get a list of the *fdisk* commands, type an *h* followed by a carriage return.

The first two columns give the partition number, both absolute and sorted. To avoid confusion, always partition your disk with the lowest numbered cylinders in partition 1, the next lowest in partition 2, and so on. To do otherwise is looking for trouble, since other operating systems may not agree on which partition is which.

A partition can be made active (bootable) using the *a* command, in which case it will be marked in the *Active* column. MINIX does not care, but MS-DOS does.

There are almost 20 partition types recognized by *fdisk*. To get a list of them, type a *t* followed by a carriage return.

The remaining information in the display gives the start and end of each partition, both in terms of cylinder/head/sector notation and in terms of sector number starting at the beginning of the disk. Note that MINIX follows the IBM convention of numbering cylinders and heads starting at 0, but sectors starting at 1.

Now you must decide how many partitions you want and how big they should be. It is suggested that you allocate either partition 1 or 2 as your MINIX partition, and the other as an MS-DOS partition, if you wish. To use MINIX conveniently, you should allocate *at least* 5 megabytes to its partition, 10 if you want to keep the sources on line, and more if at all possible. Partition sizes up to 32M should work without problems, as should disks up to 1024 cylinders.

If you have a 80286 or 80386 CPU with 1M or more, you will get the best performance by allocating some of your memory as a RAM disk. This memory will operate as a high-speed, low-latency disk, and can be used for keeping binary programs, user files or other data to which you want quick access. If you want to use a RAM disk, you must now decide how much memory will be allocated to executing programs (including the 150K operating system), and how much to RAM disk. The larger the program memory, the more programs that can execute at once. With a total of 1M memory, 512K for programs and 512K for RAM disk is a reasonable choice. With 1.5M memory, 768K for each is a good choice. With 2M or more, allocate 1M for programs and the rest for the RAM disk.

If you have decided to have a RAM disk, you should use partition 3 to store its image. Partition 3 must be greater or equal to the size of the RAM disk. When the system is booted, the contents of partition 3 will be copied to the RAM disk before the login prompt is displayed. The RAM disk is *not* copied back to partition 3 when you stop MINIX, so any changes you make to the RAM disk will be lost when you next boot, unless you explicitly mount partition 3 and copy the changes back to it. Most users put the system binaries, libraries, compilers, etc. on the RAM disk, so there is usually no need to copy them back to the hard disk when stopping, as they are rarely changed.

Once you have decided how many partitions you need, and how big they must be, you can partition the hard disk using *fdisk*. Type a *c* and answer all the questions it asks (partition number, starting and ending cylinders, etc.). When you are all done creating partitions, examine the partition table very carefully to see that it is correct. If you have made a mistake, just type *c* again to change partitions. In particular, be sure that all partitions are marked with the correct type. The *m* command can be used to mark a partition as being of type MINIX. MINIX partitions must begin at an even sector address and contain an even number of sectors, something *fdisk* will arrange for you by rounding the base up 1 sector or the size down 1 sector if need be (but only for MINIX partitions). *Fdisk* also allocates the first two sectors in partition 1 (and only partition 1) to the boot block.

When you have double checked the partition table, triple check it. If you are the nervous type, quadruple check it. An incorrect partition table will cause you more trouble than you want to hear about now. When you are convinced it is correct, type a *w* to write it back to the disk. After doing the write, *fdisk* will automatically exit back to the shell and you will get a prompt.

Now type:

```
sync
```

and reboot the computer so the kernel can read the new partition table. To reboot, insert your boot diskette in drive 0. Then depress the CTRL and ALT keys, and while holding them down, hit the DEL key. After you have brought MINIX back up, log in as *root* again and rerun *fdisk*. Check that the partition table is correct. If not, fix it and reboot. If it is correct, proceed with the next step.

2.3.4. Step 4: Make a MINIX File System on Each MINIX Partition

Now that the disk is physically partitioned, it is time to put a MINIX file system on each MINIX partition. To do this, determine the number of sectors in each partition by examining the last column in the *fdisk* output. This may not be quite what you had expected due to the use of entire cylinders and rounding effects. Also, the first two sectors of partition 1 (and only partition 1) are reserved for the partition table and do not count. Compute the number of 1K blocks in each MINIX partition by dividing the number of sectors by 2 (one block is two 512-byte sectors).

To create a file system of, say, 32000 blocks on partition 2 and 2048 blocks on partition 3, log in as root and type:

```
mkfs /dev/hd2 32000
mkfs /dev/hd3 2048
```

For other MINIX partitions (or sizes) type the analogous commands. Do not run *mkfs* on MS-DOS or other partitions. Be very careful not to make a typing error here, as making a new file system destroys all information on the partition specified.

You can verify that the file systems have been made by typing:

```
df /dev/hd2
df /dev/hd3
```

which will report on the i-nodes and blocks present on each file system. The total number of blocks should agree with the number you used in the *mkfs* command.

You can now mount your new file systems. To mount */dev/hd2* (partition 2) on */user*, type:

```
/etc/mount /dev/hd2 /user
```

To change to */dev/hd2*, type:

```
cd /user
```

This puts you in the root directory of the partition 2 file system.

2.3.5. Step 5: Check for Bad Blocks

With current manufacturing technology, it is nearly impossible for disk vendors to deliver perfect drives. Almost every drive has some bad blocks on it. If MINIX were to use a bad block in one of your files, you might lose some valuable data, so it is important to locate all the bad blocks before putting any files on the disk and make sure they do not cause trouble.

The scheme used in MINIX is to put all the bad blocks into dummy files, so that the disk space allocator will think they are in use and leave them alone. This method is more efficient than wasting entire tracks as spares, as is sometimes done. Suppose that you have allocated partitions 2 and 3 for MINIX. To locate the bad blocks on partition 2, first log in as *root*, mount */dev/hd2* on */user* and go to the root directory by typing:

```
cd /
```

It is important that the next command be executed on the root device, since it will attempt to unmount */dev/hd2*, which will fail if your working directory is there. To locate all the bad blocks, type:

```
readall –b /dev/hd2 >bad.2
```

Depending on the size and speed of your disk, this operation may take a substantial fraction of an hour. Please be patient. When it is finished, a prompt will appear on the screen. When it does, you can examine the output files using *cat*, for example, by typing:

```
cat bad.2
```

You can also examine it with the *mined* editor. The output will be a shell script that calls *badblocks* with up to seven arguments, each one the number of a bad block. Bad blocks often cluster together. This is normal.

To mark the blocks as bad, type:

```
sh <bad.2
```

When this command finishes, several files full of bad blocks may have been created. You can examine them by typing:

```
ls -la
```

They will all have names starting with *.Bad_*, followed by some numbers. You can now remove the shell script by typing:

```
rm bad.2
```

If you now type:

```
df /dev/hd2
```

you will notice that the number of blocks used has increased by the number of bad blocks found, and the number of free blocks has decreased by the same amount.

If you have more MINIX partitions, go to the root directory and unmount the current partition. Then mount the next partition and repeat the same process. If the next partition is 3, the sequence is as follows (where the text starting at the # signs are just comments):

```
cd /                          # go to the root directory
/etc/umount /dev/hd2          # Note: umount, not unmount (no n)
/etc/mount /dev/hd3 /user     # now mount partition 3
readall -b /dev/hd3 >bad.3    # find the bad blocks on partition 3
sh <bad.3                     # mark the bad blocks on partition 3
rm bad.3                      # remove the shell script
/etc/umount /dev/hd3          # again umount, not unmount (no n)
```

There is a small chance that a bad block will occur in the i-node list of a new file system. If this occurs, you must go back to Step 3 and repartition the disk with different sizes, trying until all of the i-node blocks are good.

2.3.6. Step 6: Initialize the Root File System

When MINIX boots, it needs a root file system. This file system can be on a diskette, a hard disk partition, or on the RAM disk. If it is on a hard disk partition, say partition 2, then certain directories and special files must be created on that partition. If it is on RAM disk, then an image of the RAM disk must be created on the partition that will be copied to the RAM disk, usually partition 3. Either way, a root file system is needed on some hard disk partition (unless you have a diskette-only system). In the discussion below, we will assume that a RAM disk is being used, so we will put the root file system on */dev/hd3*. If you want put it somewhere else, you will have to change *RAM_IMAGE* in *fs/main.c* and recompile the operating system.

The root file system normally has certain standard directories in it, to be described later. One of these, */dev*, contains all the character and block special files. To create the directories and special files, first change to the root directory, unmount all hard disk partitions (using */etc/umount*), then type:

```
/etc/setup_root /dev/hd3 ram hd1 hd2 hd3 hd4
```

where *ram* is the size of the RAM disk in blocks (1K), and the next four numbers are the sizes of the four hard disk partitions, also in blocks. You must be logged in as root to run */etc/setup_root*. As an example, with */dev/hd3* as 2M root device, and the four hard disk partitions being 32000, 32000, 2048, and 14000 blocks, respectively, you should type:

```
/etc/setup_root /dev/hd3 2048 32000 32000 2048 14000
```

You *must* specify all four partition sizes. If a partition is used by MS-DOS, use the actual size of the partition in blocks. (The last column of the *fdisk* listing gives the number of sectors; to get blocks, divide by 2.) If a partition has size zero, use 0.

At this point, the new hard disk root will contain the same files as the root file system diskette. To try it out, type *sync*, insert the boot diskette in drive 0, and hit CRL-ALT-DEL. When the menu appears, what you should do depends on whether you have a RAM disk or not. If you have a RAM disk (and have just set up */dev/hd3* to contain its image), leave the boot diskette in drive 0 and hit the = key (equal sign). After a few seconds, MINIX will begin loading the RAM disk from */dev/hd3*. When it is finished, you will be asked to insert the */usr* diskette (#5) and hit the ENTER key. Do so, and then enter the date when you are asked.

On the other hand, if you want the root device on, say, */dev/hd2*, when the menu appears, type:

```
r
```

to change the root device. You will be prompted again. This time type:

```
h2
```

to select (for example) hard disk partition 2 as the root device. The menu will

appear again, only this time with an important difference: */dev/hd2* will be the default root device. Now hit the = key to boot MINIX using your new root device. Subsequent boots should be done the same way as this one, first changing the root device (if need be).

The initial menu also gives you the possibility of specifying the RAM disk size and the place where the RAM image is to come from. If the latter is diskette 0, and the drive still contains the boot diskette, the system automatically uses partition 3 of the hard disk, as described above.

Having booted from the hard disk, you should type:

```
df
```

to see how much space is still available on the new root device. You need 100K to 200K free for scratch files in */tmp*, but if there is more than that available, you may wish to copy other files from */bin* on one of the other diskettes to the root (meaning */dev/hd3* if you have a RAM disk). A typical sequence for a RAM disk user might be first to unmount everything and then type:

```
/etc/mount /dev/fd0 /user     # mount a diskette on /user
/etc/mount /dev/hd3 /usr      # mount RAM image partition on /usr
cd /user/bin                  # change directories
cp file1 /usr/bin             # copy a file to the RAM image
cp file2 /usr/bin             # copy another file
cd /usr/bin                   # change to /usr/bin
chown bin *                   # change owner to bin
cd /                          # go to the root directory
/etc/umount /dev/hd3          # unmount RAM image
/etc/umount /dev/fd0          # unmount diskette
```

Again, this should all be done when logged in as root. Files can also be copied from */lib* in an analogous way. However, note that the C compiler expects to find all its passes in */usr/lib* rather than */lib*. This expectation can easily be changed by editing and recompiling *commands/cc.c*.

If the initial setup has copied files to the RAM image that you do not want there, you should remove them. After modifying the RAM image, do a *sync* and reboot to computer to see if all is well.

2.3.7. Step 7: Initialize */usr*

The next step is creating all the directories. A shell script called */etc/setup_usr* has been provided to do most of the work. It mounts the main hard disk partition (specified by its argument) and creates a large number of directories. Next, it copies files from the root file system and from diskette 5 to the */usr* tree on the hard disk. When it is finished, it asks for more diskettes to be inserted so it can copy files from them to the hard disk. Just follow the instructions that appear on the screen

until the "Installation completed" message appears. To perform the installation be sure you are logged in as *root* and type:

```
/etc/setup_usr /dev/hd2
```

assuming the main MINIX partition is partition 2. If you are not using a RAM disk, it is normal that the root partition set up in the previous section and the main partition set up in this section are the same.

Except for the three boot diskettes, all the distribution diskettes are normal MINIX file systems that you can mount and inspect if something should go wrong. When this shell script finishes, the entire MINIX file system will be installed on the hard disk. Most of the files on the distribution diskettes are compressed files (with suffix *.Z*) or compressed archives (with suffix *.a.Z*). If, for some reason, installation fails part way through, you may be left with some *.a.Z*, *.a* or *.Z* files on the disk A file *file.a.Z* can be decompressed using :

```
compress -d file.a.Z
```

If the result is an archive (with suffix *.a*), you can extract the files from the archive with the *ar* command, for example:

```
ar x file.a
```

At this point the files *file.a.Z* and *file.a* can be removed. The only archive that you *must* keep as an archive is *libc.a* as the C compiler expects it this way. Do not extract the individual files from it (unless, for some reason, you want to examine them by hand).

The */etc/setup_usr* shell script assumes that you will be mounting */dev/hd2* on */usr*, so it creates top-level directories *bin*, *lib* and so on. When mounted, these will become */usr/bin*, */usr/lib* and so on. However, if you are using a hard disk partition as root device, they will show up in the tree as */bin*, */lib* and so on, which will cause problems because the C compiler expects certain binaries and libraries in */usr/lib* and many programs expect the shell in */usr/bin*. To solve all these problems, it is necessary to make a directory *usr* at the top level of the hard disk partition tree and then move *bin*, *lib* and so on into it. A shell script, */etc/setup_move*, has been provided to do this. Thus, *if, and only if*, you are going to use a hard disk partition as root device (i.e., no RAM disk), type:

```
/etc/setup_move
```

now. If you are planning to run with a RAM disk and mount the hard disk partition on */usr*, do not run */etc/setup_move*. Please note that this shell script assumes that the MINIX hard disk partition is still mounted on */usr*, which it will be if */etc/setup_usr* completes normally and you have not unmounted it.

2.3.8. Step 8: Set up Diskette Special Files

MINIX supports three kinds of 5.25 inch diskettes, double density (360K), quad density (720K), and high density (1.2M), and two kinds of 3.5 inch diskettes (720K and 1.44M). Thus in theory there are 9 combinations of 5.25 inch drive and diskette and 4 combinations of 3.5 inch drive and diskette (although some are illegal). Since the same MINIX binary will run on any PC, with 5.25 or 3.5 inch drives, there are only two ways for the driver to figure out which drive/diskette combination is being used: (1) it runs experiments on its own, or (2) you tell it.

If you use the special file */dev/fd0* to access drive 0, option 1 is used. The driver hunts around, trying all the combinations. This can be slow.

Alternatively, you can invoke option 2 by telling the driver which combination you want. Each legal combination has been assigned four minor device numbers (for drives 0 through 3). For example, when drive 0 is a 360K drive and is being used to read a 360K diskette, minor device 4 can be used to avoid hunting; the driver knows to try only the parameters for 360K diskettes in 360K drives. Similarly, when reading a 360K diskette in a 1.2M drive, minor device 20 can be used to prevent the driver from having to hunt for the correct parameters. The table below gives the names of names, device numbers, and characteristics of the special files for diskettes drive 0 and 1 (drives A and B).

Minor devices for diskette/drive combinations						
Name	**Minor dev.**	**Inches**	**Drive**	**Diskette**	**Size**	**Parameters**
/dev/fd0	0	5.25	360K	360K/1.2M	All	Variable
/dev/fd1	1	5.25	360K	360K/1.2M	All	Variable
/dev/pc0	4	5.25	360K	360K	360K	Fixed
/dev/pc1	5	5.25	360K	360K	360K	Fixed
/dev/at0	8	5.25	1.2M	1.2M	1.2M	Fixed
/dev/at1	9	5.25	1.2M	1.2M	1.2M	Fixed
/dev/qd0	12	5.25	720K	360K	360K	Fixed
/dev/qd1	13	5.25	720K	360K	360K	Fixed
/dev/ps0	16	3.5	720K	720K	720K	Fixed
/dev/ps1	17	3.5	720K	720K	720K	Fixed
/dev/pat0	20	5.25	1.2M	360K	360K	Fixed
/dev/pat1	21	5.25	1.2M	360K	360K	Fixed
/dev/qh0	24	5.25	1.2M	720K	720K	Fixed
/dev/qh1	25	5.25	1.2M	720K	720K	Fixed
/dev/PS0	28	3.5	1.44M	1.44M	1.44M	Fixed
/dev/PS1	29	3.5	1.44M	1.44M	1.44M	Fixed

For example, to read a 1.2M diskette in a 1.2M drive 0, use */dev/at0*. No hunting will be done. Only the parameters for this combination will be used and under no conditions will an attempt be made to read data beyond 1.2M. Reading a 360K diskette with */dev/at0* will fail with errors, in contrast to using */dev/fd0* where, after considerable hunting, the driver will discover the correct parameters and be able to read the diskette correctly. Throughout this manual, wherever*/dev/fd0* is used as an example, it is permitted (encouraged!) to substitute another block special file for corresponds to a specific parameter set rather than the general one.

For each combination, four minor device numbers have been reserved, so MINIX can handle up to four diskette drives. Only drives 0 and 1 are listed above, but drives 2 and 3 are also supported. For example, minor device 6 can be used for a 360K drive 2 reading a 360K diskette. Not all the special files are present in */dev*. You can make new combinations with *mknod*. For example:

```
mknod /dev/pc2 b 2 6 360
```

makes */dev/pc2* as block device with major device number 2 (required for all diskette devices) and minor device number 6, with a size of 360K.

2.3.9. Step 9: Reading-in MS-DOS Diskettes

If you have MS-DOS diskettes containing files that you want to read in, you can do so using *dosread*. For example, to read an ASCII file *foobar* from drive A and put it in the MINIX file system as */usr/ast/phoobar* type:

```
dosread –a foobar >/usr/ast/phoobar
```

While it is possible to read in MS-DOS executable programs and store them using MINIX, since the MINIX and MS-DOS system calls are totally different, you cannot run them.

2.4. TESTING MINIX

After having installed MINIX you should test it to see if everything is working correctly. To do this, type:

```
sync
```

and then log out using CTRL-D. Reboot the computer and log in as *root*. Logging out and rebooting is part of the testing process and should not be skipped. During the boot process you will be asked to insert the */usr* diskette (#5), as usual, because this is part of the standard */etc/rc*. You must obey it.

However, after having logged in, you can switch to the hard disk (still assuming partition 2) by typing:

```
cd /
/etc/umount /dev/fd0
/etc/mount /dev/hd2 /usr
```

At this point you will be running MINIX using the hard disk. Since it is a nuisance to keep having to log in using the */usr* diskette, you may wish to edit */etc/rc* to replace the line that reads:

```
/etc/mount /dev/fd0 /usr        # mount the diskette
```

with a line that mounts the hard disk instead, for example:

```
/etc/mount /dev/hd2 /usr        # mount the hard disk
```

You should then remove the call to *getlf* since there is no need for human intervention when mounting a hard disk. Please note very carefully that editing */etc/rc* is pointless if it is on the RAM disk. On the next reboot, the */etc/rc* file from the RAM image (e.g., */dev/hd3*) will be used, so the changes will be lost. You must mount the partition containing the RAM image and edit it. On the other hand, if you are not using a RAM disk, it is sufficient to edit the true */etc/rc* on the root device.

Now go to directory */usr/src/test* and compile the tests (as *root*) by typing:

```
cd /usr/src/test
make all
```

This will compile the MINIX test suite. If it compiles correctly, log out and then log in again as *ast* to run the tests. You need not reboot the computer. The test programs may not work correctly if you run as *root*, because in addition to trying legal operations, they try illegal ones to see if they fail with the proper error codes. As *root* some of them will not fail and the test programs will (incorrectly) report errors. There is also a danger that they may make the file system inconsistent (in which case run *fsck* to repair it). After having logged out with CTRL-D and then back in again as *ast*, type:

```
cd /usr/src/test
run
```

to run all the test programs. They should all run and produce no error messages. The tests may take 15 minutes or more on a slow machine. Please do not interrupt them, as this may leave garbage files all over.

2.5. TROUBLESHOOTING

Sometimes things can go wrong. In this section we will describe some of the more common problems and what you can do about them. Far and away, the most common problem is caused by trying to run MINIX on a machine whose manufacturer claims that it is 100 percent IBM compatible, but which in reality has somewhat different hardware. The machine *must* have a NEC 765 diskette controller, a Motorola 6845 video controller, an Intel 8259A interrupt controller, and so on. Accept no substitutes. For MS-DOS this difference will be masked by a BIOS that hides the differences. Since MINIX does not use the BIOS, MINIX may not run on such machines. If you are having trouble getting started at all, one thing to try is find a friend's machine and try MINIX there. If it works, then the problem is clearly due to incompatible hardware. Experience shows that virtually all problems booting MINIX are caused by hardware whose manufacturer's idea of "IBM compatible" means that it can run MS-DOS.

If you get a message about an invalid root file system while booting, chances are you have put the wrong diskette in the drive. Turn off the computer and start all over.

Another problem that often occurs is the presence of nonstandard video cards. If you are having screen problems, hit the F3 key to enable software scrolling. This may solve the problems. Hitting F3 again goes back to hardware scrolling.

Nonstandard video cards sometimes give problems with screen editors and other programs that use the */etc/termcap* file. If this happens, again try F3. If that does not help, you might try removing the *DC*, *dc*, *DL*, and/or *dl* entries from */etc/termcap* to try to pinpoint the problem.

Another video problem is the presence of "snow" on the screen when scrolling when using some older CGA cards. This problem is caused by the fact that these cards are not dual ported, which means that the screen can only be written on when the electron beam is making a vertical retrace, which happens every 16.67 msec for a period of 4 msec. Restricting writing to the video RAM to this interval greatly slows down the system, so the default is to write whenever there is data. If the snow disturbs you and you are willing to accept much slower output, go the the file *kernel/klib88.s* and remove the comment symbols on the three lines starting at the label *vid.2*. Then recompile the operating system. Doing this will cause the screen driver to delay writing until a retrace has begun.

Sometimes the presence of unusual I/O devices causes trouble. If MINIX will not boot at all, try removing all mice, analog/digital converters, streamer tapes, modems, printers, and so on, and try again. If this works, replace the devices one at a time until the guilty party is pinpointed.

Autoconfigure hard disks sometimes give problems because their ROM's store parameters and scratch variables at the bottom of memory. Although MS-DOS does not use these locations, MINIX does. If you have an autoconfigure hard disk and are having problems, try MINIX on a friend's machine that does not have such a disk. If

it works there, the problem is probably the disk. The solution is to disable autoconfigure.

MINIX is supplied with a number of hard disk drivers, all of them ending in *wini.c*. If you are having hard disk problems, try building a kernel (on a friend's machine) using each driver in turn to see if any of them work. Please note that the PS/2 drivers expect an ST-506 interface. They do not work with ESDI drives. If you have an ESDI or other non-ST-506 drive, you must write your own driver. This is not actually as bad as it sounds. After all, quite a few examples of disk drivers are provided to study.

Some Model 50s use integrated controllers and other Model 50s have different model bytes in the ROM. This wide variety of possibilities, none of them well documented, can lead to problems. It is likely that other models have similar troubles. The net result is that MINIX cannot always figure out whether the machine has an AT bus or a Microchannel. Consequently, you may have to make minor changes to the logic of *kernel/cstart.c* to tell the system what kind of a machine you actually have. By studying the code and doing some experimenting, you should be able to get MINIX running in protected mode. Of course if you do not mind running in real mode, like MS-DOS, the BIOS driver should work fine on all models.

3

MINIX ON THE ATARI ST

In this chapter we will describe how to boot and install MINIX on the Atari ST. It is assumed that the reader is already familiar with MINIX in general, and has at least some knowledge of UNIX.

Booting and installing MINIX on the Atari ST is complicated by a variety of factors:

- Atari STs are sold with a wide variety of incompatible keyboards
- Some versions can only handle 360K diskettes; others can handle 360K and 720K diskettes
- Some people have winchester disks (hard disks); others do not
- The amount of memory available ranges from 512K to 4M
- The Mega ST differs in some ways from the Atari ST 520 and 1040

Together, these different configurations give problems. Our solution has been to provide a base version that will require at least 1MB of memory and one 720K disk drive, and make it possible for people with larger configurations to adapt the system to take advantage of their extra hardware. This chapter explains how that is done.

It is possible to run MINIX on a system with 512K of memory, but that leaves

very little space for applications. In this case the best thing to do is to work without a ram disk at all, and keep the root filesystem on either hard disk or diskette. Running MINIX on a system with only a 360K disk drive is also possible. However in that case you must first split each 720K diskettes in the distribution into two 360K ones. Since you do not have a disk drive capable of dealing with 720K diskettes, you should do this on a friends system. Sec. 3.6 describes how to split a 720K diskette into two 360K ones.

3.1. THE MINIX-ST DISTRIBUTION

The MINIX-ST distribution consists of ten diskettes. One of them contains a binary of the operating system and is used for booting MINIX-ST. Eight others contain MINIX file systems. Only one of them contains a TOS file system. (We use TOS as the name for any combination of BIOS, XBIOS, GEMDOS, GEM, AES and VDI).

All distribution diskettes are double-side, and formatted on both sides. However, two diskettes contain only 360K of information written on one side of the diskette. In other words, these two diskettes are written as if they were single sided diskettes. Here is the list of the diskettes:

Name	Sides	Size	File sys.	Description
00.TOS	DS	720K	TOS	utilities that run as TOS programs
01.BOOT	DS	360K	special	used for booting MINIX
02.ROOT	DS	360K	MINIX	root file system copied to RAM disk
03.USR1	DS	720K	MINIX	most commonly used commands
04.USR2	DS	720K	MINIX	commands part 2 of 3
05.USR3	DS	720K	MINIX	commands part 3 of 3
06.ACK	DS	720K	MINIX	compiler binaries and libraries
07.SRC1	DS	720K	MINIX	sources of the MINIX operating system
08.SRC2	DS	720K	MINIX	sources of commands part 1 of 2
09.SRC3	DS	720K	MINIX	sources of commands part 2 of 2

We will refer to these diskettes in the rest of this manual by their name in the first column of this table, for example, 01.BOOT.

Before you start working with these diskettes we urge you to copy all of them. You can use normal TOS procedures, like dragging icon FLOPPY DISK A onto icon FLOPPY DISK B, to make the copies. Whenever we refer to diskettes as 01.BOOT, 02.ROOT, 03.USR1 etc. we always mean a write-enabled copy of the

original diskettes. Store the original diskettes after copying and keep them write protected under all circumstances. Do not use your originals as work diskettes.

3.2. NATIONAL KEYBOARDS

The Atari ST comes with different keyboards in different countries. This lack of standardization is a major nuisance. Atari solved the problem by providing a different version of their operating system for each country. We have chosen a different strategy: a single version that can be adapted to the various keyboards. This section describes how to set up MINIX for your keyboard.

Unless you have an Atari ST with a United States version of the keyboard, you *must* first adapt MINIX to your particular version of the keyboard. Even with a United States version this procedure can do no harm, so if in doubt proceed. If you skip this procedure, it is assumed that the keys generate the characters that are engraved on the key tops of the United States keyboard, that is, the key below the DEL will generate the ASCII backslash character (\) unshifted, and the ASCII bar (|) if shifted, irrespective of the character engraved on the key tops of your keyboard.

MINIX cannot handle the national characters themselves, like o-umlaut for Germany. The adaptations described below only allow you to enter the ASCII characters in the way you are used to with TOS. In this respect MINIX behaves like most versions of UNIX.

MINIX has its own keyboard translation tables build into the operating system. A special tool is provided to extract the keyboard tables from a running version of TOS and to adapt the tables in the binary version of the MINIX kernel accordingly, without the need to recompile the MINIX kernel. Note that the MINIX keyboard translation tables have exactly the same format as used by TOS.

Some keyboard versions need so many keys for special non-ASCII characters that combinations with the Alternate (ALT) key are used to generate some ASCII characters. For instance, in France the key below the Delete (DEL) generates the ASCII sharp sign (#), SHIFT-# generates the bar (|), ALT-# generates the at-sign (@), and ALT-SHIFT-# generates the tilde. The keyboard tables do not take the ALT key into account, so Atari delivers several national versions of the TOS operating system to cope with these problems, as mentioned above. In order to avoid national versions of MINIX, we have built into the keyboard driver a little table of special ALT and ALT-SHIFT combinations for the limited number of national keyboard versions that we knew of: United States, United Kingdom, Germany, France and Spain. If you happen to have another version you can make a simple modification in the keyboard driver of the kernel, but that takes effect only after recompiling the kernel. Refer to the chapter on kernel recompilation.

To adapt the keyboard tables proceed as follows:

1. Boot TOS and insert 00.TOS in drive 0.
2. Open a window onto drive 0 by double clicking the FLOPPY DISK A icon.
3. Run *COMMAND.TOS* found on diskette 00.TOS by double clicking. *COMMAND.TOS* is a simple line-oriented command interpreter.
4. Run *FIXKEYS.PRG* by typing

   ```
   fixkeys a:
   ```

5. Insert unprotected 01.BOOT when you are asked to do so, and confirm by hitting the RETURN key.
6. Wait for the program to reply with

   ```
   Done
   ```

You are now ready to boot MINIX.

3.3. BOOTING MINIX-ST

This section presents a boot procedure for MINIX-ST that works on all configurations of the Atari ST. Following sections describe how to adapt the set of diskettes so that you can use MINIX effectively on your particular combination of memory and disk drives. For example, if you have more than 512K you can increase the size of the RAM disk from 160K to 300K, if you have 1M of memory, or to 1M or more if you have even more memory. If you have a hard disk, all of the diskettes can be copied onto one or more of its partitions. Finally, some of the options for booting MINIX will be explained. But first the procedure for booting that works on all configurations is described.

Throughout the discussion below, lines printed in the Helvetica typeface are either commands you should type on the keyboard, or are lines that the computer will display for you. In a few of the examples, *italics* characters or words appear in a command. These represent values that you are to fill in.

Booting is a three stage procedure. First the operating system itself is loaded into memory. Then the ROOT file system is copied to a RAM disk allocated in memory. Finally, the script */etc/rc* is executed and a message will be displayed on the screen asking you to log on.

To boot MINIX-ST, proceed as follows:

1. Turn off the ST and then insert diskette 01.BOOT in drive 0. You could push the RESET button as well, but that may not free memory occupied by a crash resistant TOS RAM disk you may have running. Moreover, it fails if you normally boot from the winchester.

2. Wait ten seconds, then turn on the ST. It will read the operating system (about 153K) from diskette in a few seconds. The screen will turn black and it will show on the top two lines the message:

   ```
   Booting MINIX 1.5. Copyright 1990 Prentice-Hall, Inc.
   Insert ROOT diskette and hit RETURN (or specify bootdev)
   ```

3. Replace 01.BOOT by diskette 02.ROOT and hit RETURN. Alternatives will be explained later. The system will respond with:

   ```
   Memory size=992K MINIX=153K RAM disk=160K Available=679K
   ```

   for a system with 1M of RAM (numbers might deviate a little). Adding 32K (the size of the video memory) to the first number should give the amount of memory in your ST.

4. A fourth line will be displayed that reads:

   ```
   RAM disk. To load: 120K        Loaded: 0K
   ```

   (The number 120 may vary a little). In rapid succession the number 0 will be increased in steps of 18K, until the whole line is replaced by:

   ```
   RAM disk loaded.   Please remove root diskette.
   ```

5. When the RAM disk is loaded, the system initialization file, */etc/rc*, is executed. It asks you to remove the root file system and insert the */usr* file system (03.USR1) in drive 0 and type a RETURN. Do so.

6. After */usr* has been mounted, you will next be requested to enter the date (and time). Enter a 12-digit number in the form MMDDYYhhmmss, followed by a RETURN. For example, 9:35 p.m. on June 01, 1990 was 060190213500.

7. You will now get the message:

   ```
   login:
   ```

   on the screen. Type:

   ```
   root
   ```

   and wait for the system to ask for your password. Then type:

   ```
   Geheim
   ```

   being careful to type the first letter in upper case. Lower and upper

case letters are always distinct in MINIX. Alternatively, you could have used the name "ast" together with the password "Wachtwoord". This is much preferred when you use the system normally, but for now it is troublesome.

8. If you have successfully logged in, the shell will display a prompt (sharp sign for root, dollar sign otherwise) on the screen. Try typing:

   ```
   ls –l
   ```

   to see what is in the root directory. Note that you need six keystrokes: "l", "s", space, "–", "l", and a RETURN. Then type

   ```
   ls –l /bin
   ```

   to see what is in the */bin* directory on the root device (RAM disk). After that, try:

   ```
   ls –l /usr/bin
   ```

   to see what is on the drive 0 diskette. To stop the display from scrolling out of view, type CTRL-S; to restart it, type CTRL-Q. (Note that CTRL-S means depress the "Control" key on the keyboard and then hit the *S* key while "control" is still depressed.)

9. You can now edit files, compile programs, or do many other things. The reference manuals given in chapters 8 and 9 of this manual give a brief description of the programs available. However, before rushing off we advise you to adapt the system to your hardware configuration first, as described in the next sections.

10. When you are finished working, and want to log out, type CTRL-D. The

    ```
    login:
    ```

    message will appear, and you or another user can log in again.

11. When you want to shut the computer down, make sure all processes have finished, if need be, by killing them with *kill*. Then type *sync* or just log out. When the disk light goes out, you can turn the computer power off. Never turn the system off without first running *sync* or logging out (which does an implied *sync*). Failure to obey this rule will generally result in a garbled file system and lost data.

3.4. INCREASING THE SIZE OF YOUR RAM DISK

If you have 1M or more of memory, we advise you to increase the size of the RAM disk from 160K to 300K or more. A larger ramdisk allows you to use the RAM disk to copy complete or partial file systems from one diskette to another. It also gives you plenty of space to add a few more utilities to the ROOT file system. Finally, it allows you to compile much larger programs without running out of disk space for the intermediate results. On the other hand it leaves you with less memory to run your MINIX applications. Choosing a RAM disk of 300K leaves you enough memory to recompile most the sources and perform many other tasks.

It is easiest, to use a 360K diskette to carry this enlarged ROOT file system. However, it is a little tricky, to use a 360K diskette to carry a larger ROOT file system. Say you want to make a 512K RAM disk. You may wonder how a 512K RAM disk can be initialized by reading it in from a 360K diskette during the boot procedure. The secret is that the 512K RAM disk is not completely full. Part of it is initially empty so it can be used for scratch files. Only the initialized part (up to 360K) has to be read in. The only problem with this approach is that if you make new root file systems, you should be careful that they do not exceed 360K of data. Failing to do so may damage the file system on your diskette severely.

To install a 512K RAM disk, you must first make a 512K root file system diskette as described below. When MINIX is booted, it looks at the size of the root file system and sets its size accordingly. If you have more than 1 MB, you might even consider making a RAM disk larger than 512K, although only 360K can be initialized at start-up time. To do this, proceed as follows.

1. Take an empty, formatted diskette and label it 10.R512

2. Boot MINIX-ST as described above and login as root. Then type:

   ```
   for i in cpdir mkfs; do cp /usr/bin/$i /bin; done
   /etc/umount /dev/dd0
   ```

3. Insert 10.R512 in drive 0 and type:

   ```
   mkfs -t /dev/fd0 512
   /etc/mount /dev/fd0 /user
   cpdir -msv / /user
   ```

4. Logout by typing CTRL-D.

5. Insert 01.BOOT in drive 0 and type CTRL-ALT-DEL to reboot using 01.BOOT, 10.R512 and 03.USR1.

Do not forget the **–t** option to *mkfs*. It suppresses the check if the new file system fits on the medium. The program *cpdir* will tell you that it skipped the directory */user* to avoid recursion.

By changing the argument *512* to *mkfs* you can adapt the size of the RAM disk. However, if you take a value less than 250 you will run into the problem that *mkfs* allocates not enough inodes to store all the entries of the root file system. If you have 1M of memory and you want to recompile the system a RAM disk of 300K is recommended. Replace the last two occurrences of */dev/fd0* by */dev/dd0* if you prefer to use 720K diskettes, or by */dev/hd3*, or any other hard disk partition, if you want to load the RAM disk from the winchester. Read the section on boot options below if you do.

Note that a copy of the programs *cpdir* and *mkfs* will be present in */bin* on your new ROOT diskette.

3.5. ADAPTING PROGRAMS TO USE EXTRA RAM

As distributed, the C compiler is tuned to work on even the smallest Atari ST configuration. This causes problems if you want to recompile (parts of) MINIX. The first part of the C compiler proper, */usr/lib/cem*, as distributed is configured for a stack size of 40K, but it needs about 70000 bytes more to compile some of the larger source files on the distribution diskettes. It is possible to compile small programs on a 512K machine with the default memory allocation of the compiler.

If you have at least one of the following:

- more than 512K of memory
- two drives, either diskette or hard disk

there are ways to recompile all of MINIX. Note that it is impossible to recompile some parts of MINIX on an ST with only 512K of memory and a single drive.

You are strongly advised to execute the following procedure now if you have more than the minimal 512K of memory.

1. Boot MINIX-ST and login as root.
2. Type:

```
cp /usr/bin/chmem /bin
chmem =35000 /usr/bin/make
/etc/umount /dev/dd0
```

3. Insert 06.ACK in drive 0 and type:

```
/etc/mount /dev/dd0 /usr
chmem =110000 /usr/lib/cem
```

A similar procedure can be executed if you encounter any other program that needs more memory.

3.6. USING SINGLE-SIDED DISKETTES

The distribution contains several 720K diskettes. Most, but not all, Atari ST machines, have a disk drive that can handle 720K diskettes. Only a few older systems can only handle 360K diskettes. If you have one of these systems do not despair. You can split a single 720K diskette into a pair of 360K diskettes on a system with a 720K disk drive. Since you do not have such a system you will have to borrow one from a friend or perhaps your local dealer.

To split 04.USR2 into 13.USR2A and 14.USR2B proceed as follows:

1. Boot MINIX-ST using 01.BOOT, 10.R512 and 03.USR1; login as root.

2. Type:

```
for i in cpdir mkfs rmdir; do cp /usr/bin/$i /bin; done
/etc/umount /dev/dd0
```

3. Insert 04.USR2 in drive 0 and type:

```
/etc/mount /dev/dd0 /user
mkdir /tmp/a
```

4. Now copy files from */user* to */tmp/a*. You should add files to /tmp/a until the command

```
du –s /tmp/a
```

   reports a value just below 355.

5. Unmount using:

```
/etc/umount /dev/dd0
```

6. Remove 04.USR2 and insert an empty, single-side formatted disk labeled 13.USR2A in drive 0 and type:

```
mkfs /dev/fd0 360
/etc/mount /dev/fd0 /user
cpdir –msv /tmp/a /user
/etc/umount /dev/fd0
rm –rf /tmp/a
```

7. Repeat the same process for the second half of the files on 04.USR2, using an empty, single-side formatted disk labeled 14.USR2B.

Be careful about the subtle difference between */usr* and */user*, between */dev/fd0* and */dev/dd0*, and between *13.USR2A*, *14.USR2B* and *04.USR2*. The result is two 360K diskettes that contain all of 04.USR2. Similarly, you can divide others.

It may happen that you need more than two 360K disks to contain all files of one 720K disk, because the file system itself imposes some overhead that is now doubled. Use three 360K diskettes in those cases.

After you have divided all other 720K diskettes and you have verified your work, you should make another copy of your root diskette (02.ROOT or 10.R512) and modify the file */etc/rc* on that new copy, replacing the line

```
/etc/mount /dev/dd0 /usr
```

by

```
/etc/mount /dev/fd0 /usr
```

Now you can use this new 360K version of MINIX just like the original one. However exercise some care when dealing with examples in this chapter or section 7.2, since they assume a 720K version.

3.7. USING A HARD DISK

If you have a hard disk and one or more partitions free for MINIX, you can use it to keep (part of) the distributed diskettes on line. If you have any choice, use a small (512K to 1M) partition 3 (*/dev/hd3*) to hold the ROOT file system that is copied to the RAM disk at boot time. See the section on boot options below. One of the other partitions, for example 4 (*/dev/hd4*), can be as big as 32M and can be mounted on */usr*. It is also possible to keep the root file system on diskette and only use a partition to store the usr file system. In that case you can skip step 6 below. The penalty for keeping the root file system on diskette is an additional disk swap and some additional delay when booting the system. There is no difference in behavior after booting. You could use the whole disk (*/dev/hd5*) (up to 32 MB) as one single MINIX file system, but that would make the disk useless for TOS.

This section describes the steps to set up MINIX on such a system.

3.7.1. Step 1: Backup the Hard Disk

If you are already used your hard disk for TOS, before even *contemplating* installing MINIX, you should make a complete backup of the contents of your hard disk onto diskette or another medium. As a bare minimum, installing MINIX will require erasing one partition of your hard disk, and possibly two. However, to prevent disaster in the event that you make an error during the setup procedure, it is highly desirable that you backup the *entire* disk before you even start. Your files are too valuable to put at risk.

It is worth noting that MINIX has a program, *tos*, that can read TOS diskettes. Thus if you make your backup on diskettes, you will be able to read the files into the MINIX file system after you have completed the hard disk installation.

3.7.2. Step 2: Verify that Your Hard Disk is Atari Compatible

There are a number of different hardware vendors for the Atari ST. Most of their disks work with MINIX. However, some hard disks will not co-operate with MINIX. For example it is known that some of the very first Supra disk controllers will not work with MINIX, due to a bug in the controller. Newer Supra disks (the ones with a SCSI out port) do not have this problem.

To verify that MINIX is indeed able to correctly access your hard disk, boot MINIX as described above, but instead of logging in as *ast*, log in as *root*, using *Geheim* as password (note the upper case *G*). If you are already logged in as *ast*, use CTRL-D to log out, then log in again as *root* (without rebooting). Logging in as *root* makes you the superuser and gives you the sharp sign (#) as prompt instead of the usual dollar sign. The superuser is the system administrator and has special privileges denied ordinary users. To install MINIX on your hard disk, you will need these privileges. Once the installation is complete, you should always log in as *ast*, or create your own login name as described later in this manual.

Once you are successfully logged in as *root*, type:

```
dd if=/dev/hd5 of=/dev/null count=200
```

After a short time, you should get the message:

```
200+0 records in
200+0 records out
```

If you get an error message or no response, MINIX cannot use your hard disk controller.

3.7.3. Step 3: Partition the Hard Disk

Initialize the hard disk (formatting and partitioning) using the tools supplied by Atari, notably the *HDX.PRG* utility. If you have already partitioned your disk before, and you are happy with the partition sizes you can skip this step. Be warned that partitioning the hard disk will destroy all information on that disk. MINIX is not equipped to initialize your disk. The MINIX disk driver requires no special settings of the *pi_flag* and *pi_id* fields (see the Atari hard disk manual), mainly because the Atari hard disk driver code is deficient in properly maintaining the hard disk information found in sector 0. This requires you not to mix up which operating system should operate on which partition, unfortunately. MINIX checks the super block on mounts and it is unlikely that a TOS partition will be accepted. However, writing to a TOS partition by accessing */dev/hd?* directly, although superuser only, is not prevented. Be careful. Similarly, avoid TOS accesses to MINIX partitions. It is a good idea to remove the icons for the MINIX partitions from the TOS desktop.

Another problem is that the *HDX.PRG* seems not to format the last sector on the disk properly, so never use the last sector of the last partition. This is probably a

bug in *HDX.PRG*. So, whenever you make a MINIX file system on the last partition, subtract 1 from the real number of sectors of that partition when calling *mkfs*.

If you have any choice, allocate a small partition 3 of 512K, and a large partition 4 of at least 10M. This setup is assumed in the rest of this section.

3.7.4. Step 4: Make a MINIX File System on Each MINIX Partition

Now that the disk is physically partitioned, it is time to put a MINIX file system on each MINIX partition. To do this, determine the number of sectors in each partition. *HDX.PRG* will have told you the number of sectors when partitioning. The number may not be quite what you had expected due to the use of entire cylinders and rounding effects. Compute the number of 1K blocks in each MINIX partition by dividing the number of sectors by 2 (one block is two 512-byte sectors).

An alternative is to use the command *readall* with the option –t on each partition. For example:

```
readall –t /dev/rhd4
```

will tell you the number of 1k blocks on */dev/hd4*. It is possible that during the execution of readall you get a few error messages about unrecoverable disk errors. These error messages can be ignored safely.

To create a file system of, say, 512 blocks of 1K on partition 3 and 10239 blocks of 1K on partition 4, log in as root and type:

```
mkfs /dev/hd3 512
mkfs /dev/hd4 10239
```

Notice the 10239 (10240 minus 1) due to the bug in *HDX.PRG* mentioned before. For other MINIX partitions (or sizes) type the analogous commands. Do not run *mkfs* on TOS or other partitions. Be very careful not to make a typing error here, as making a new file system destroys all information on the partition specified.

You can verify that the file systems have been made by typing:

```
df /dev/hd3
df /dev/hd4
```

which will report on the i-nodes and blocks present on each file system. The total number of blocks should agree with the number you used in the *mkfs* command.

You can now mount your new file systems. To mount */dev/hd3* (partition 3) on */user*, type:

```
/etc/mount /dev/hd3 /user
```

To change to */dev/hd3*, type:

```
cd /user
```

This puts you in the root directory of the partition 3 file system.

3.7.5. Step 5: Check for Bad Blocks

With current manufacturing technology, it is nearly impossible for disk vendors to deliver perfect drives. Almost every drive has some bad blocks on it. If MINIX were to use a bad block in one of your files, you might lose some valuable data, so it is important to locate all the bad blocks before putting any files on the disk and make sure they do not cause trouble.

The scheme used in MINIX is to put all the bad blocks into dummy files, so that the disk space allocator will think they are in use and leave them alone. This method is more efficient than wasting entire tracks as spares, as is sometimes done. Suppose that you have allocated partitions 3 and 4 for MINIX. To locate the bad blocks on partition 3, first log in as *root*, go to the root directory, and unmount the partition, if mounted, by typing:

```
cd /
/etc/umount /dev/hd3
```

It is important that the next commands be executed on the root device, since they will attempt to mount and unmount */dev/hd3*, which will fail if your working directory is there. To locate all the bad blocks, type:

```
readall –b /dev/rhd3 >bad.3
```

Depending on the size and speed of your disk, this operation may take a substantial fraction of an hour. Please be patient. It is possible that during the execution of *readall* you get a few error messages about unrecoverable disk errors. These error messages can be ignored safely. When it is finished, a prompt will appear on the screen. When it does, you can examine the output files using *cat*, *more*, or an editor, for example, by typing:

```
cat bad.3
```

The output will be a shell script that calls *badblocks* with up to seven arguments, each one the number of a bad block. Bad blocks often cluster together. This is normal.

To mark the blocks as bad, type:

```
sh <bad.3
```

When this command finishes, several files full of bad blocks may have been created in the root directory of the device containing the bad blocks. In the example above these files are created in the top level directory of /dev/hd3. After mounting the disk you can examine them by typing:

```
ls –la
```

They will all have names starting with *.Bad_*, followed by some numbers. Do not examine or remove the files. You can now remove the shell script by typing:

```
rm bad.3
```

If you now type:

```
df /dev/hd3
```

you will notice that the number of blocks used has increased by the number of bad blocks found, and the number of free blocks has decreased by the same amount.

If you have more MINIX partitions, go to the root directory and unmount the current partition. Then mount the next partition and repeat the same process. If the next partition is 4, the sequence is as follows (where the text starting at the # signs are just comments):

```
cd /                           # go to the root directory
/etc/umount /dev/hd3           # if still mounted
readall -b /dev/rdh4 >bad.4    # find the bad blocks on partition 4
sh <bad.4                      # mark the bad blocks on partition 4
rm bad.4                       # remove the shell script
/etc/umount /dev/hd4
```

There is a small chance that a bad block will occur in the i-node list of a new file system. If this occurs, you must go back to Step 3 and repartition the disk with different sizes, trying until all of the i-node blocks are good.

3.7.6. Step 6: Initialize the Root File System

When MINIX boots, it needs a root file system. By default, this root file system is read from a 360K diskette and copied into memory as a RAM disk. If you have a hard disk an easier alternative is to read the root file system from a hard disk partition, preferably */dev/hd3*, and copy it into the RAM disk.

This requires you to make a copy of the root file system onto */dev/hd3*. In the discussion below we will put the root file system on the 512K partition */dev/hd3* on which we have already made an empty file system above. However, you could equally well use another partition, but take care that the size of the file system you make on that partition (the argument to *mkfs*) is used as the size of your RAM disk.

The procedure below is actually rather similar to the procedure described before to increase the size of your RAM disk. Proceed as follows:

1. Boot MINIX-ST with 01.BOOT, any ROOT (02.ROOT or 10.R512) and 03.USR1 and login as root. Then type:

```
for i in cpdir mkfs chmod; do cp /usr/bin/$i /bin; done
/etc/umount /dev/dd0
/etc/mount /dev/hd3 /user
cpdir -msv / /user
```

2. Logout by typing CTRL-D.

You can now test if the new root file system really can be used to boot from. Insert 01.BOOT in drive 0 and type CTRL-ALT-DEL to reboot. You will be confronted again with the message:

```
Insert ROOT diskette and hit RETURN (or specify bootdev)
```

As alternative for the insertion of 02.ROOT or 10.R512 as second step in the boot procedure you now have three option:

1. Keep the 01.BOOT diskette in drive 0, and hit RETURN. MINIX-ST will detect that you have no diskette inserted and will try to load the root file system from hard disk partion 3, precisely where we have created our new root file system.

2. Reply with

   ```
   3,3
   ```

   to override the default by loading the root file system from hard disk partition 3.

3. Reply with any other drive specification, like

   ```
   3,2
   ```

   if you want to load the root file system from partition 2, for instance.

3.7.7. Step 7: Initialize */usr*

The next step is creating all the directories. A shell script called */etc/setup_usr* has been provided to do most of the work. It creates a large number of directories. Next, it copies files from the distribution diskettes to the */usr* tree on the hard disk. It asks for 03.USR1 to 09.SRC3 in sequence. Just follow the instructions that appear on the screen until the "Installation completed" message appears. To perform the installation be sure you are logged in as *root*. We assume that you have

setup */dev/hd4* as described above, and that */dev/hd4* contains at least 10M. Then, proceed as follows:

1. Boot MINIX-ST using 01.BOOT, any ROOT (02.ROOT, 10.R512 or hd3) and 03.USR1 and login as root.
2. Type the commands:

```
for i in cpdir test echo; do cp /usr/bin/$i /bin; done
/etc/umount /dev/dd0
/etc/mount /dev/hd4 /usr
/etc/setup_usr
```

3. Follow the instructions displayed by the *setup_usr* script. If your partition is smaller than 10M, the best thing to do is to install only the binaries onto the hard disk. Type *quit* when the system asks you to insert disk 07 (07.SRC1). Installing only the binaries will require 4M.

Except for the boot diskette and the tos diskette, all the distribution diskettes are normal MINIX file systems that you can mount and inspect if something should go wrong. When this shell script finishes, the entire MINIX file system will be installed on the hard disk. Most of the files on the distribution diskettes are compressed files (with suffix *.Z*) or compressed archives (with suffix *.a.Z*). If, for some reason, installation fails part way through, you may be left with some *.a.Z*, *.a* or *.Z* files on the disk A file *file.a.Z* can be decompressed using :

```
compress -d file.a.Z
```

If the result is an archive (with suffix *.a*), you can extract the files from the archive with the *ar* command, for example:

```
ar x file.a
```

At this point the files *file.a.Z* and *file.a* can be removed. The only archive that you *must* keep as an archive is *libc.a* as the C compiler expects it this way. Do not extract the individual files from it!

From now on you can mount */dev/hd4* at boot time as */usr* by making a small change in */etc/rc* found on the ROOT file system (diskette or winchester). Use *mined* (see chapter 9 on how to use *mined*) to change the first two lines that read:

```
/bin/getlf "Please insert /usr diskette in drive 0. Then hit RETURN."
/etc/mount /dev/dd0 /usr
```

by a single line that reads:

```
/etc/mount /dev/hd4 /usr
```

Inserting diskette 03.USR1 will no longer be necessary at boot time.

3.8. USING A MEGA ST

The Mega ST series is internally quite similar to the Atari 520 ST and 1040 ST machines. It has more memory, which is automatically supported by MINIX-ST. It has a blitter chip, but currently MINIX-ST does not support it. Another standard feature is the battery powered real time clock. To eliminate the need to type the date each time the system is boot, a small program that reads out the current date and time from the real time clock, and sets the MINIX time accordingly has been provided. If you have a Mega ST you are advised to adapt the file */etc/rc* so that it will use that program *megartc* whenever you boot. Replace the line that reads:

```
/usr/bin/date -q </dev/tty
```

by the following two lines:

```
/usr/bin/megartc
/usr/bin/date
```

Note that *megartc* is found on 03.USR1. This change has the following effect. The program *date* queries the terminal for the date and then installs the date. The program *megartc* takes the date from the real time clock instead of asking for it from the terminal. The second line causes the date to be printed.

As an aside, please note that any other commands inserted in the file */etc/rc* will be executed before the system is booted. However, when inserting commands there, be sure that they do not require programs or files that are on diskettes that have not yet been mounted.

3.9. USING A DISK CONTROLLER BASED CLOCK

Since the original Atari ST did not contain a battery powered real time clock, quite a number of add-on clocks have appeared on the market. MINIX-ST supports the real time clock from Weide. It also supports the clocks available on various third party disk controller boards, but only if you recompile your kernel with the **-DCLOCKS** option in the kernel Makefile turned on. See chapter 7 for an explanation of rebuilding the kernel. For both types of clocks a small program that reads out the current date and time from the real time clock, and sets the MINIX time accordingly has been provided. In both cases you are advised to adapt the file */etc/rc* so that it will read the real time clock whenever you boot. If you have a Weide real time clock replace the line that reads:

```
/usr/bin/date -q </dev/tty
```

by the following two lines:

```
/usr/bin/weidertc
/usr/bin/date
```

If you have a disk controller with a real time clock and have a modified operating system (*MINIX.IMG*) on your 01.BOOT diskette, replace the same line by:

```
/usr/bin/diskrtc controller
/usr/bin/date
```

where *controller* is one of *supra*, *icd*, *bms1* (for a BMS 100 controller) or *bms2* (for a BMS 200 controller). Note that also these programs are found on 03.USR1.

3.10. BOOT PROCEDURE OPTIONS

The boot sequence we have described so far always starts with a 360K BOOT diskette in drive 0, followed by a 360K ROOT diskette in drive 0. Between the BOOT and ROOT diskette you have always answered the question:

```
Insert ROOT diskette and hit RETURN (or specify bootdev)
```

by hitting RETURN. If the ROOT file system is found on another device you may specify that device as:

major,minor

where *major* is a decimal number specifying the device type and *minor* is a decimal number specifying the drive and/or partition. These *major,minor* pairs correspond with the numbers you see in the output of:

```
ls -l /dev
```

Some of the useful combinations are:

Major	Minor	Device	Description
2	0	fd0	360K diskette in drive 0
2	1	fd1	360K diskette in drive 1
2	8	dd0	720K diskette in drive 0
2	9	dd1	720K diskette in drive 1
3	1	hd1	partition 1 of hard disk 0
3	2	hd2	partition 2 of hard disk 0
3	3	hd3	partition 3 of hard disk 0
3	4	hd4	partition 4 of hard disk 0
3	5	hd5	complete hard disk 0

So, if ROOT is found on a 720K diskette in drive 1 the second line of your screen will look like:

```
Insert ROOT diskette and hit RETURN (or specify bootdev) 2,9
```

If you specify nothing or anything illegal, MINIX will check two default devices in sequence. First it tries to read the super block of the ROOT file system on 2,0 (360K diskette in drive 0). Only if that fails (read error or illegal super block) it tries 3,3 (partition 3 of hard disk 0). That is why we advised you to use */dev/hd3* as copy of the RAM disk.

One of the more exotic options of the boot sequence is to read the MINIX operating system itself from a TOS file, not using the BOOT diskette. On the diskette 00.TOS you find a TOS program *MINIX.PRG* that takes as first argument the name of a TOS file, default *MINIX.IMG*, that contains the operating system. You can create the file *MINIX.IMG* yourself by reading enough sectors from the BOOT diskette, starting with sector 0, but it requires at least one other diskette, hard or RAM disk besides a:. The procedure below assumes that you have a TOS RAM disk named *m:*. Proceed as follows:

1. Start TOS.
2. Insert a copy of 00.TOS, in drive 0.
3. Double click icon FLOPPY DISK A.
4. Double click *COMMAND.TOS* on A:

   ```
   rflop a: m:\minix.img 100000
   ```

5. Insert protected 01.BOOT if you are asked and hit RETURN.
6. When done, put *MINIX.PRG* and *MINIX.IMG* onto a TOS diskette

The third argument to *RFLOP* is the number of bytes to read. 100000 is more than sufficient for the operating system as distributed. You can now copy *MINIX.PRG* and *MINIX.IMG* to a TOS partition of the hard disk. Assuming that you normally boot TOS from the hard disk, you can subsequently switch to MINIX by double clicking *MINIX.PRG*. If you want to switch back to TOS you logout by typing CTRL-D. If you see the prompt *login:* again, type CTRL-ALT-DEL.

3.11. UNPACKING THE SOURCES

The sources, except the compiler and *elle*, are on the SRC diskettes. These diskettes are normal MINIX file systems, which you can mount using the command:

```
mount /dev/dd0 /user
```

The files on the distribution diskettes are compressed archives (with suffix *.a.Z*). If you want to extract the sources from a file *file.a.Z* you should first copy this file to either an empty diskette, or to the RAM disk, if the latter is large enough. Typically about 4 times the size of the compressed file is required when extracting the sources. If later on you want to recompile the sources even more space may be required. That is why you first should copy the compressed source file to an empty diskette. Your copy of *file.a.Z* can be decompressed using:

```
compress -d file.a.Z
```

After decompressing you can remove your copy of *file.a.Z*. Now you can extract the files from the archive with the *ar* command, for example:

```
ar x file.a
```

At this point all files from the archive are extracted. You can now remove *file.a* since it is no longer needed.

3.12. THE TOS TOOLS

Several tools have been developed for TOS. In the early stages of the MINIX-ST port TOS was used as the development environment. That forced us to port tools like *mkfs* and *build*, and to develop the programs *minix* and *relmix*. Later, when the native MINIX-ST C compiler became available, we could use MINIX-ST itself for further development. Rather than simply discarding the TOS tools, we have included them in the distribution for the benefit of people wishing to do further MINIX-ST developments using TOS. Below we describe these tools in the same style as the MINIX commands.

Command: **BUILD.PRG – build *MINIX.IMG*** out of its constituent parts
Syntax: **build** *bootblok kernel mm fs init menu minix.img*
Flags: (none)

Build takes the six constituent parts and produces the MINIX-ST operating system image. That image, if written onto a diskette starting at sector 0, is bootable on the Atari ST. Alternatively, the program *MINIX.PRG* can be used once TOS is up and running.

Command: **FIXKEYS.PRG – patch BOOT diskette for TOS keyboard table**
Syntax: **fixkeys** [**–d**] [**–o**] *drive*
Flags: **–d** Double-sided diskette
–o Accept not only a: and b:
Example: fixkeys a: # Modify BOOT diskette in drive a:

Fixkeys patches the keyboard tables of the currently active version of TOS into the MINIX-ST operating system image as normally found on the BOOT diskettes. It can only operate on diskettes, not on file images.

Command: **KEYTBL.TTP – display the keyboard tables**
Syntax: **keytbl.ttp** [*file*]
Flags: (none)

Keytbl writes the keyboard tables to the file whose name is gives as a parameter (or to standard output if no parameter is present). This file can be used when recompiling the kernel. Refer to chapter 7 for details on how to recompile the kernel.

Command: **MINIX.PRG – boot MINIX-ST from an image on file**
Syntax: **minix** [*image*]
Flags: **(none)**
Example: minix minix.img # boot MINIX-ST from minix.img

Minix allows you to boot MINIX-ST if TOS is already up and running. It reads the operating system image from a TOS file into memory, copies the image to address 0 and jumps to the address found at location 4. There is no way back to TOS, except by rebooting the machine.

Command: **MKFS.PRG – make a MINIX-ST file system**
Syntax: **mkfs** [**–dol**] *drive prototype*
Flags: **–d** Double-sided diskette
–o Overwrite: accept not only a: and b:
–l Make a listing on standard output
Examples: mkfs a: proto # Make a file system on drive a:
mkfs -d b: 360 # Make empty 360 block file system

Mkfs builds a file system and copies specified files to it. See chapter 8 for a description of the proto file syntax. The files used to initialize the new file system should conform to the TOS syntax, including backslashes and drive specifications.

Command: RELMIX.PRG – change loadfile from .68K to .MIX format
Syntax: **relmix** [+*amount*] [–*amount*] [=*amount*] *prog.68k prog.mix*
Flags:
+ Increase memory allocation
– Decrease memory allocation
= Set memory allocation

Example: relmix =2000 x.68k x.mix # Make MINIX-ST style loadfile

The Alcyon 4.14 C compiler, part of the Atari ST developers kit, produces a loadfile in .68K format. A simple transformation of the header, removal of the symbol table, and a transformation of the relocation information as performed by the RELMOD.PRG program (also part of the developers kit) does the trick.

Command: RFLOP.PRG – read bytes from diskette
Syntax: **rflop** [**–d**][**–o**] *drive file bytes*
Flags:
–d Double-sided diskette
–o Accept not only a: and b:

Example: rflop a: minix.img 100000 # Read *minix.img* from BOOT diskette

An arbitrary number of bytes is read from the diskette and written to a TOS file. Reading always starts at sector 0.

Command: WFLOP.PRG – write bytes to diskette
Syntax: **wflop** [**–d**] [**–o**] *drive file*
Flags:
–d Double-sided diskette
–o Accept not only a: and b:

Examples: wflop a: minix.img # Make BOOT diskette

A TOS file is written to a diskette, starting at sector 0. This overwrites the information in sector 0 used by TOS to determine the type of the diskette.

3.13. TROUBLESHOOTING

As a user of MINIX-ST you may be confronted with some of the error messages the system can produce. The following subsections give guidelines on you how to react. It also explains how you can use the built-in debugging aids.

If you have problems booting the system, try the following steps: power down the machine, wait 10 seconds, insert the BOOT diskette in drive 0, and power up the machine. If you have a hard disk and normally boot from the hard disk directly, you may either force booting from the diskette as described in the Atari hard disk manual, or start MINIX-ST using the supplied TOS program *MINIX.PRG*, as described in section 3.9 of this manual.

Either way, the screen should turn black and you will see two lines printed on the top of the screen, asking you to insert the ROOT diskette. If these lines do not appear, the BOOT diskette is probably damaged.

Hitting RETURN at this point should give one more line. If not, you might suspect the keyboard or the BOOT diskette. Normally, when the root file system is being read in, regular progress reports appear on the screen. If not, the diskette drive may not be working correctly with the MINIX-ST diskette driver (e.g., because your diskette controller does not generate interrupts as it should). This should not be a problem with all known Atari ST production models, but we have heard about some problems with very old development machines. If disk error messages appear on the screen, your drive may need slower step rates than usual. Official Atari diskette drives should work correctly.

If the system behaves funny or even crashes while loading the root file system, the ROOT diskette is suspect. It might be corrupted or too big for this machine. If so, try it with (a copy of) your 02.ROOT diskette.

If you have gone through these critical initial steps you should not have any problems getting MINIX-ST booted, since the essential resources, the diskette, the keyboard and the screen are probably all right.

3.13.1. Error Messages

Many of the error messages are also found in MINIX-PC. Here we list the MINIX-ST specific ones. In all cases % followed by a letter gives the *printf* format of the number.

Three messages are printed by the kernel if commands running on top of the operating system itself encounter problems, such as unsolicited hardware traps and stack overflow. These three are:

- **sig=%d to pid=%d at pc=%X**
  Generated if bus errors, segmentation faults, illegal instructions or funny traps are encountered,

- **Stack low (pid=%d,pc=%X,sp=%X,end=%X)**
  If a stack overflow has happened or is about to happen, and

- **Unexpected trap. Vector = %d**
  This may be due to accidentally including
  a non-MINIX library routine that is trying to make a system call.
  If any of the trap instructions is executed that is not used by MINIX-ST. In both cases, the system will continue, but the program is likely to be aborted with a core dump generated on the file *core*.

A number of messages announce unexpected hardware events, sometimes only a warning, sometimes more serious, but not immediately fatal. In this category fall:

- **fd%d: timeout**
  No diskette in drive (you have 15 seconds to insert one)

- **fd%d: read: dma status = 0x%x**
  DMA error on diskette read request

- **fd%d: read sector %d: fdc status = 0x%x**
  Diskette controller error on read request

- **fd%d: write protected**
  Writing to write-protected diskette

- **fd%d: write sector %d: fdc status = 0x%x**
  Diskette controller error on write request

- **fd%d: recalibrate failed. status = 0x%x**
  Cannot find track 0

- **hd: read: drive=%d sector=%D status=0x%x**
  Hard disk error on read request

- **hd: write: drive=%d sector=%D status=0x%x**
  Hard disk error on write request

- **DMA interrupt discarded**
  Unsolicited interrupt from device on DMA bus

- **midi interrupt: status=%x, data=%x**
  Unsolicited interrupt from MIDI interface

- **Fake interrupt handler for %s. trap = %02x**
  Unsolicited interrupt from:
  timint,00: timer A of MFP chip
  timint,01: timer B of MFP chip
  timint,03: timer D of MFP chip
  siaint,00: MFP RS232: char received
  siaint,01: MFP RS232: receive error
  siaint,02: MFP RS232: char transmitted
  siaint,03: MFP RS232: transmit error
  iob,01: MFP RS232 Data Carrier Detect

iob,02: MFP RS232 Clear To Send
iob,03: unused
iob,06: MFP RS232 Ring Indicator
iob,07: Monochrome Monitor Detect

- **Printer is not available**
  Ready bit off: not connected or off line

- **printer: still busy**
  Interrupt received, but not ready

More serious conditions cause a system panic. A message is printed and an infinite loop is entered. Only a reset helps: push the RESET button (sometimes CTRL-ALT-DEL works as well). The most important MINIX-ST specific kernel panics are:

- **dma:ASSERT(%s) failed**
  Consistency checking in stdma.c

- **fd:ASSERT(%s) failed**
  Consistency checking in stfloppy.c

- **Nonexisting interrupt. Vector = %d**
  A trap via one of the vectors that is unassigned

- **Unexpected interrupt. Vector = %d**
  A trap via one of the autovectors not used by the ST

- **trap via vector %d**
  A synchronous trap in kernel mode

- **no shadow?**
  Two processes share an ORIGINAL, but neither points to a SHADOW

- **rmshadow: cannot handle physio shadows**
  SHADOW with p_physio set must be copied to ORIGINAL.

- **only shadow(s)**
  All that share ORIGINAL have SHADOW set

- **tty_init: unknown terminal %d**
  For all NR_TTYS an initialization routine must be called

The important file system panics are:

- **Invalid root file system**
- **RAM disk is too big. # blocks = %d**
- **Root file system corrupted. Possibly wrong diskette.**
- **init: can't load root bit maps**

For all these errors retry booting with (a copy of) 02.ROOT.

3.13.2. Debugging Aids

Some of the internal tables can be inspected by special key combinations. The keyboard driver recognizes the following key combinations and calls debugging routines in the kernel:

CTRL-ALT-F1	dump of the process table
CTRL-ALT-F2	dump of the memory map
CTRL-ALT-F3	dump of the status of the current process

The process table shows the current and lowest stack pointer detected, the CPU time spend in user mode and system mode, and the memory slot occupied for each process, including kernel tasks, as well as other information.

The memory map shows (for user processes only) the location and length of the text, data and stack segments, and the shadowing fields *p_shadow*, *p_nflips* and *p_physio*.

The status of the current process shows the register values most recently saved, a memory dump around the location of the program counter, and a memory dump around the location of the stack pointer. The memory dumps can be used to give a stack trace and, by manual disassembly, the instructions executed most recently.

The CTRL-ALT-F6 key combination toggles an option to dump the same tables whenever the message

sig=%d to pid=%d at pc=%X

is printed and whenever the system panics. By default the "automatic table dump" option is OFF.

The CTRL-ALT-F5 key combination toggles an option to send all kernel generated output not only to the screen but also to the line printer. By default the "kernel output to printer" option is OFF. If the printer is offline, the printing is temporarily suppressed. If the "kernel output to printer" option is ON, and the printer switches from online to offline, it may take a few seconds to detect this, since initially it looks similar to a printer buffer full condition. Be patient.

The CTRL-ALT-F4 key combination toggles an option to send all kernel generated output to the screen. By default the "kernel output to screen" option is ON.

A good procedure, if you encounter a problem and you want to spot it, is to isolate the problem such that it is reproducible. Then, insert paper in the printer and toggle CTRL-ALT-F5 and CTRL-ALT-F6 to capture the debugging information on paper while you reproduce the error situation.

If the problem is in a command the *core* file contains a memory dump at the time of the crash. These post mortem dumps can be analyzed using the mdb debugger. Refer to chapter 9 for a description of *mdb*.

If problems are encountered in the MINIX-ST driver, you have a chance that that driver has debugging statements coded in. By changing either the #DEBUG or #TRACE definitions, you can effectuate these statements, but only after recompilation. Refer to chapter 7 on how to recompile MINIX

4

MINIX ON THE COMMODORE AMIGA

This chapter tells you how to install and run MINIX on a Commodore Amiga. Four sections are present in this chapter. The first section discusses the kind of hardware you need to run MINIX. The second section gives an overview of how to get MINIX running. The third one goes into more details. The fourth one is about troubleshooting. If during the installation you have problems, please check the troubleshooting section. You may have run into a common problem whose solution is well known and described there. When you have finished reading this chapter and have successfully installed MINIX, please skip to Chap. 6 to learn about using your newly installed MINIX system.

4.1. MINIX HARDWARE REQUIREMENTS

MINIX should run on any Amiga 500 or Amiga 2000 that has at least 1M of memory. The most common peripherals are supported, except for the hard disk. Extra memory or a second drive makes programming much more pleasant. While it is possible to boot MINIX with a 512K 1 drive system, it is difficult to do anything serious, certainly not recompiling the operating system. To do that, you really should buy an additional 512K.

4.2. HOW TO START MINIX

Throughout the discussion below, lines printed in the Helvetica typeface are either commands you should type on the keyboard, or are lines that the computer will display for you.

Before running MINIX for the first time, make a backup of all the diskettes, to prevent disaster if one of them should be subsequently damaged. They are not copy protected. However, all of them, except the first one (called: "BOOT"), are not AmigaDOS disks, so do not use any of the usual AmigaDOS disk copy programs. Instead use either a copy program that is able to copy IBM-PC disks, or use the supplied *diskcopy* utility. Since MINIX does not come with a format program, use the *transfer* utility, which can be found in the C directory of the boot disk and can be invoked by typing:

```
BOOT:C/transfer –f
```

under AmigaDOS. If you do not have a program to copy IBM-PC disks under AmigaDOS you can not yet backup your original disks. Please remember to do so once you have got MINIX working.

To boot MINIX, proceed as follows.

1. Turn on the Amiga, and insert the boot diskette (BOOT) in any drive. If the Amiga was already powered on, you may also press both Amiga keys while holding down the control key to reset the Amiga.

2. Because the AmigaDOS diskette does not contain any of the usual utilities such as *setmap, rename*, etc., you have to copy them from your original Workbench disk onto the AmigaDOS disk yourself. When you boot MINIX for the very first time, a little program will show you exactly how to do so. When you have successfully copied the required utilities onto the AmigaDOS disk you should re-boot the Amiga. From now on the Amiga will automatically load MINIX whenever your boot from the BOOT disk.

3. About 15 seconds later, MINIX will ask you to either specify a root device or press return. Insert the root disk (ROOT) and press return. The RAM-disk will now be loaded into memory.

4. Another 10 seconds later MINIX will display a line telling how much memory the machine has, how large the operating system (including all its tables and buffers) is, how large the RAM disk is, and how much memory is available for user programs (the first number minus the next two). Check to see that the available memory is at least positive. MINIX will not run with negative memory. To do anything useful, however, at least 200K of available memory is needed.

5. When the RAM disk has been loaded, the system initialization file, */etc/rc*, is executed. It asks you to remove the root file system and then insert the */usr* file system ("USR") in drive 0 and type a carriage return. Do so.

6. After */usr* has been mounted, you will next be requested to enter the date (and time). Enter a 12-digit number in the form MMDDYYhhmmss, followed by a carriage return. For example, 3:35 p.m. on July 4, 1976 was 070476153500.

7. You will now get the message:

   ```
   login:
   ```

   on the screen. Type:

   ```
   ast
   ```

   and wait for the system to ask for your password. Then type:

   ```
   Wachtwoord
   ```

   being careful to type the first letter in upper case. Lower and upper case letters are always distinct in MINIX. Do not use an upper case letter when a lower case one is called for or vice versa. Like UNIX, MINIX regards "a" and "A" as two distinct characters. Please do not type "a" when you mean "A". It matters.

8. If you have successfully logged in, the shell will display a prompt (dollar sign) on the screen. Try typing:

   ```
   ls –l /bin
   ```

   to see what is in the */bin* directory on the root device. After that, try:

   ```
   ls –l /usr/bin
   ```

   to see what is on the drive 0 diskette. To stop the display from scrolling out of view, type CTRL-S; to restart it, type CTRL-Q. (Note that CTRL-S means depress the "control" key on the keyboard and then hit the S key while "control" is still depressed.)

9. If you have more than one diskette drive, you can mount one of the other diskettes by inserting it into drive 1 and typing:

   ```
   /etc/mount /dev/dd1 /user
   ```

   If you want to use drive 2 or 3, replace */dev/dd1* by */dev/dd2* or */dev/dd3* respectively. Use *ls* to inspect it. You can now try out other commands.

10. When you are finished and want to log out, type: CTRL-D. The

```
login:
```

message will appear, and you or another user can log in again.

11. When you want to shut the computer down, make sure all processes have finished, if need be, by killing them with *kill*. Then type:

```
sync
```

or just log out. When the disk light goes out, you can turn the computer power off. *Never*, ever turn the system off without first running *sync* or logging out (which does an implied *sync*). Failure to obey this rule might result in a garbled file system and lost data. If you forget and just turn off the computer, next time you boot, be sure to run *fsck* to repair the file system.

4.3. A MORE DETAILED LOOK

In this chapter we will describe some of the details of MINIX. Note: some programming examples will be presented in the rest of this chapter. You can recognize them by the prompts: The 1> prompt indicates that you should type the command in an AmigaDOS CLI window, the $ indicates a normal MINIX commando and the # indicates commands that should be run by the superuser (logged in as *root*). You *must not* type the prompts themselves, just type the commands following them.

The MINIX distribution consists of one disk in the normal 880K AmigaDOS format (which contains some tools and a binary of the operating system and is used for booting MINIX) plus a number of double-sided 720K MINIX disks. We will refer to these diskettes in the rest of this manual by their name in the first column of the following table. Here is the list of diskettes:

Name	Size	File system	Description
01 BOOT	880K	AmigaDOS	Used for booting MINIX
02 ROOT	720K	MINIX	160K Root file system copied to RAM disk
03 USR1	720K	MINIX	System Binaries 1 (/usr)
04 USR2	720K	MINIX	System Binaries 2
05 USR3	720K	MINIX	System Binaries 3
06 ACK	720K	MINIX	C compiler
07 SRC1	720K	MINIX	Operating System Sources
08 SRC2	720K	MINIX	Commands Sources 1
09 SRC3	720K	MINIX	Commands Sources 2

If you have not already made backups, now is the time to do so. You can use the normal AmigaDOS procedure to copy BOOT, as is described in the AmigaDOS

manuals, or you can use any of the available disk copiers. To copy the MINIX disks you will have to use MINIX itself. Be sure you have 8 formatted (see section 4.2) disks ready, to copy the original onto. Be sure to follow *diskcopy's* instructions and repeat this 7 more times. You can also use other means, e.g., *dos2dos's* format command or a real PC or Atari-ST to format the disks. We will refer to the copies as BOOT, ROOT, USR, ACK and SRC. Keep the original disks write protected under all circumstances to prevent accidental loss of the original source.

4.3.1. Keyboards

The Amiga comes with different keyboards in different countries. MINIX solves this in the normal Amiga-way: keymaps. This section describes how to set up your keyboard for MINIX.

If you have one of the European keyboards, you must first install a keymap for your particular version of the keyboard (unless you are willing to live with the US key bindings, meaning that the character engraved on the keytop will not always correctly describe which key it is). Life would have been a lot simpler if typewriter manufacturers had devised an international standard keyboard 100 years ago.

There are several methods for installing a keymap, increasing in complexity. If the one you use fails, please try one of the other methods. For all of the methods, we assume that you are a bit familiar with the CLI. If you are not, please read that part of the manual that came with your Amiga. Start up your Amiga and boot from your favorite Workbench disk. Now put BOOT: in a drive.

Using One of the Prefab Keymap Files.

Find out which one you normally use by typing:

```
1> type S:startup-sequence
```

You should see a line like:

```
setmap nl
```

which means you normally use the Dutch (nl) map:

```
1> cd BOOT:
1> dir devs/keymaps
```

You will see several files, all starting with *m_*, such as *m_usa0*. These are keymap files. You should specify your keymap by editing *BOOT:S/startup-sequence*. To specify the *nl* keymap, change *BOOT:c/setmap m_usa1* to *BOOT:c/setmap m_nl*.

If you cannot find your favorite map among the *m_** files, or it fails for some other reason proceed with step 2.

Converting Your Keymap.

Patch a keymap using one of the keymap editors available. We'll assume that you are using *KeyMapEd* because it is public domain and quite good. If you have another keymap editor it will probably do just fine.

The only changes necessary are: help, up, left, right, down to \x00, del to \x7f, f1 to \x1bOP, f2 to \x1bOQ, f3 to \x1bOR, f4 to \x1bOS, f5 to \x1bOT, f1 to \x1bOU, f7 to \x1bOV, f8 to \x1bOW, f9 to \x1bOX, f10 to \x1bOY.

You might change the (shifted) function key definitions: these are the only ones where it makes sense to select "string." If you do so you can map any of them to an arbitrary string. Do not to exceed an average length of 20 characters per key because in the kernel there are only 400 bytes to store their definitions. When redefining the keys, do not change the definitions of the (unshifted) function keys, since they are used by the *mined* editor.

Give up

If you think that this is all quite complicated or you are not so sure about really doing any of it, you can skip it for now and find out how the default map (*m_usal*) works for you. If worst comes to worst, experimentally determine what all the keys do, and paste paper stickers on the key tops giving their new functions.

4.3.2. The Preferences

When MINIX boots it copies all sorts of information from AmigaDOS such as the mouse pointer, which will be used as a cursor under MINIX, border and character colors, the keymap, the memory map, etc. To change the default settings you can boot from the BOOT disk, hit CTRL-D before MINIX has actually booted and then run *preferences* from your Workbench.

4.3.3. Exchanging Files between AmigaDOS and MINIX

Just as in the other versions of MINIX you can exchange files between AmigaDOS and MINIX. One problem, however, is that AmigaDOS uses a nonstandard diskette format; not just a different file system, but also a different encoding scheme for the data. To overcome this problem, we have provided a *transfer* utility to read, write and format MINIX diskettes under AmigaDOS. For more information on *transfer* consult the manual pages in Chap. 8.

The 720K diskettes used by MINIX on the Amiga conform to the industry standard for 3.5 inch diskettes, and can be read on the Atari ST using MINIX there. Thus you can make MINIX file systems on your Amiga and then use them on an Atari ST and vice versa. In fact, binary programs compiled on any of these systems can be run on any of the others without modification. This makes it easier for you to share

software with other MINIX users. People who do not believe in standardization are requested to read Sec. 4.3.1 again.

4.3.4. Making Backups of MINIX

Ok, how about *your first good deed as a MINIX user?* Boot MINIX login, and type:

```
# cd /
# diskcopy
```

diskcopy asks you to insert the source disk, insert ROOT, the first of the disks, do so and hit return. After a while *diskcopy* will ask you for the (formatted) destination disk, insert one of the disks you've just formatted. repeat this process for the others and then store the original disks together with the original BOOT in a safe place. You will not need them again unless you accidentally damage one of the new copies. *Diskcopy* unmounts the */usr* disk so you'll have to remount it when it is done. First insert the the USR diskette, then type:

```
# /etc/mount /dev/dd0 /usr
```

You can check if MINIX is working 100% as follows. Type:

```
# cd /usr/test
# run
```

These elaborate tests take over 15 minutes. If no error messages appear, the system is working properly. Be sure the diskette is write enabled.

You can now edit files, compile programs, or do many other things. The reference manuals given in Chap 8. and the extended ones in Chap 9. tell you about the programs available and what they do. The descriptions are for reference purposes, however. They are not tutorials. If you are unfamiliar with UNIX, it is suggested that you first read one of the many books available on this subject. Any good computer bookstore will have a wide selection of them.

4.3.5. Boot Procedure Options

While the default boot sequence will probably be just fine most of the time, you can change the behavior of the MINIX kernel with some useful options. The MINIX kernel, fs, mm and init, packed together in *BOOT:minix.img*, contain the MINIX code. This data file is read in by a little utility program called *minix*. This is an ordinary AmigaDOS program, it finds out some things about your Amiga (how much RAM you have, what keymap you use, if you have a NTSC 60Hz or PAL 50Hz machine etc.), then it loads *minix.img*, passes the information to it and starts it off. For the exact usage of *minix*, see the manual pages.

4.3.6. Even More Details

If you want to know more about the exact differences between the Amiga and the IBM versions of MINIX, you might also read the Atari specific chapter, since the Amiga version was derived from the Atari version, which was derived from the IBM version. The IBM version was not derived from anything. Details about exactly what devices are available, how the tty driver works, etc. can be found there.

4.4. TROUBLESHOOTING

Sometimes things can go wrong. If you are having trouble getting started, you should try to find a friend's machine and try MINIX there. If it works, then the problem is clearly due to incompatible hardware. To verify that this is indeed the problem, remove (or at least disconnect) all optional equipment from your Amiga and try again. If this works, insert the optional equipment one device at a time, rebooting MINIX after each one is installed until the guilty party is located.

If problems arise after you have gained enough experience to recompile the kernel, you might compile (parts of) the kernel with the **–DDEBUG** flag as to allow extra debugging output to appear when MINIX is running. This option is not very useful for inexperienced users, however.

5

MINIX ON THE APPLE MACINTOSH

In this chapter we will describe how to boot and install MINIX on the Apple Macintosh. It is assumed that the reader is already familiar with MINIX in general, and has at least some knowledge of UNIX. Readers not at all familiar with UNIX should probably begin by looking at one of the many introductory articles and books about it, as this manual does not contain any tutorial material on UNIX.

If you plan on running multifinder with MACMINIX, be sure to read the multifinder section below.

5.1. MACMINIX HARDWARE REQUIREMENTS

MINIX will run on any Macintosh with at least one megabyte of memory and 128K (or larger) ROMs. MACMINIX has been tested most extensively with system software version 6.0 or latter. Earlier versions may present some problems.

5.2. THE MACMINIX DISTRIBUTION

The MACMINIX distribution consists of eight double-sided 800K diskettes. One of them contains the boot application and the root file system, and is used for booting MACMINIX. Since MACMINIX file systems are also ordinary Macintosh operating

system files, all of the diskettes are readable by the normal Macintosh operating system. Here is the list of the diskettes:

Name	Size	File system	Description
00.BOOT	800K	MAC OS/MINIX	Used for booting MINIX
01.USR1	800K	MAC OS/MINIX	Commands
02.USR2	800K	MAC OS/MINIX	Commands
03.ACK	800K	MAC OS/MINIX	C compiler
04.SRC1	800K	MAC OS/MINIX	Sources of MINIX
05.SRC2	800K	MAC OS/MINIX	Sources of commands
06.SRC3	800K	MAC OS/MINIX	Sources of commands
07.SRC4	800K	MAC OS/MINIX	Editors

We will refer to these diskettes in the rest of this chapter by their name in the first column of this table, for example, 00.BOOT.

Before you start working with these diskettes we strongly advise you to make copies of them. You can use normal Macintosh procedures to make these copies. If you are not familiar with how to copy a diskette, refer to your Macintosh owner's guide. Keep the original disks write protected under all circumstances. *Make sure that the copies are named identically to the originals*, since the following procedures depend on this. Once you have made the copies, place the originals in a safe place and use the copies.

5.3. NATIONAL KEYBOARDS

The Macintosh has different keyboards for different countries. When booting, MACMINIX uses the Macintosh Toolbox to assist in the creation of the virtual key code to ASCII translation table, so assuming that the international resources have been properly configured for your machine, the table will be correct for your country.

5.4. BOOTING MACMINIX

This section presents a boot procedure for MACMINIX that works on all Macintosh configurations. Following sections describe how to adapt the set of diskettes so that you can use MINIX effectively on your particular combination of memory and disk drives. For example, if you have more than 1 megabyte of memory but no hard disk, you may wish to increase the size of the RAM disk to 512K. If you have a hard disk, all of the diskettes can be copied onto one or more of its partitions.

Finally, some of the options for booting MINIX will be explained. But first the procedure for booting that works on all configurations.

Throughout the discussion below, lines printed in the Helvetica typeface are either commands you should type on the keyboard, or are lines that the computer will display for you. In a few of the examples, *italics* characters or words appear in a command. These represent values that you are to fill in.

Booting is a three stage procedure. First the operating system itself is loaded into memory. Then the ROOT file system is copied to a RAM disk allocated in memory. Finally, the script */etc/rc* is executed and you are asked to log on.

To boot MACMINIX, proceed as follows:

1. Follow your normal booting conventions to boot your Macintosh.

2. Place the boot diskette, 00.BOOT in drive 0, and launch the boot boot application (named boot) by double clicking on it. A window will appear, and the MACMINIX boot application will exhibit a status line as it loads each of the initial software components of MINIX: the *kernel*, memory management (*mm*), the file system (*fs*), and process 0 (init). When loading is complete, this window will disappear.

3. The main console window will then display (approximately):

   ```
   Booting MACMINIX 1.5. Copyright 1990 Prentice-Hall, Inc.
   Memory size=775 MINIX=165 RAM disk=160K Available=475K
   ```

   for a system with 1M of RAM. The memory not accounted for is left for the Macintosh operating system to use. The amount of memory left can be configured by you. See the Configuration section below.

4. A fourth line will be displayed that reads:

   ```
   RAM disk. To load: 160K     Loaded: 0K
   ```

   Again, the number may vary. In rapid succession the number 0 will be increased in steps of 5K, until the whole line is replaced by:

   ```
   RAM disk loaded.
   ```

5. When the RAM disk is loaded, the system initialization file, */etc/rc*, is executed. It ejects the boot diskette (00.BOOT) and asks you to insert the */usr* file system (01.USR1) in a drive and type a RETURN. Do so.

6. After */usr* has been mounted, you will now get the message:

   ```
   login:
   ```

   on the screen. Type:

   ```
   root
   ```

and wait for the system to ask for your password. When it does, please type:

```
Geheim
```

being careful to type the first letter in upper case. Lower and upper case letters are always distinct in MINIX. Alternatively, you could have used the name "ast" together with the password "Wachtwoord". This is much preferred when you use the system normally, but for now it is troublesome.

7. If you have successfully logged in, the shell will display a prompt (sharp sign for root, dollar sign otherwise) on the screen. Try typing:

   ```
   ls –l
   ```

   to see what is in the root directory. Note that you need six keystrokes: "l", "s", space, "–", "l", and a RETURN. Then type:

   ```
   ls –l /bin
   ```

   to see what is in the */bin* directory on the root device (RAM disk). After that, try:

   ```
   ls –l /usr/bin
   ```

   to see what is on the drive 0 diskette. To stop the display from scrolling out of view, type CTRL-S; to restart it, type CTRL-Q. (Note that CTRL-S means depress the "Control" key on the keyboard and then hit the *S* key while "control" is still depressed. If your keyboard does not have a control key, as with the normal Macintosh Plus keyboard, you may use the option key instead.)

8. You can now edit files, compile programs, or do many other things. The reference manuals given in chapters 8 and 9 of this manual give a brief description of the programs available. However, before rushing off we advise you to adapt the system to your hardware configuration first, as described in the next section.

9. When you are finished working, and want to log out, type CTRL-D. The

   ```
   login:
   ```

   message will appear, and you or another user can log in again.

10. When you want to leave MINIX, make sure all processes have finished, if need be, by killing them with *kill*. Then type: *sync* or just log out. Your can then select the "Quit" menu item from the "File" menu, and this will return you to your familiar Macintosh desktop. Never quit

without first running *sync* or logging out (which does an implied *sync*). Failure to obey this rule will generally result in a garbled file system and lost data.

5.5. INCREASING THE SIZE OF YOUR RAM DISK

If you have more then 1M of memory, and are not planning on using a hard disk, we advise you to increase the size of the RAM disk from 160K to 512K. This allows you to use the RAM disk to copy complete or partial file systems from one diskette to another. It also gives you plenty of space to add a few more utilities to the ROOT file system. Finally, it allows you to compile much larger programs without running out of disk space for the intermediate results. On the other hand it leaves you with somewhat less memory to run your MINIX applications. But that is more than sufficient to recompile most the sources and perform many other complicated tasks.

To install a 512K RAM disk, you must first make a 512K root file system diskette as described below. When MINIX is booted, it looks at the size of the root file system and sets its size accordingly. If you have significantly more than 1 MB, you might even consider making a RAM disk larger than 512K. To do this, proceed as follows.

1. Take an empty, formatted, 800K diskette, name it 00.BOOT, and copy the boot application from your original 00.BOOT onto the new diskette (in the normal Macintosh way).

2. Boot MACMINIX as above and login as root. Then type:

```
for i in cpdir mkfs mknod chmod; do cp /usr/bin/$i /bin; done
/etc/umount /dev/hd0
/etc/hdclose /dev/hd0
/etc/eject
```

3. Insert the *new* 00.BOOT in drive 0 and type:

```
/etc/maccreate 512 00.BOOT:ROOT
/etc/hdopen 00.BOOT:ROOT /dev/hd0
mkfs /dev/hd0 512
/etc/mount /dev/hd0 /user
cpdir -msv / /user
/etc/setup_root
```

4. Logout by typing CTRL-D.

5. Quit from MACMINIX by selecting "Quit" from the "File" menu. Restart MACMINIX using the above booting procedure, but use your newly created 00.BOOT diskette in place of the original 00.BOOT diskette.

The program *cpdir* is able to copy the devices in */dev*, so that will not be a problem when executing */etc/setup_root*. *Cpdir* also will tell you that it skipped the directory */user* to avoid recursion.

By changing the argument *512* to *mkfs* you can adapt the size of the RAM disk. Note that a copy of the programs *cpdir*, *mkfs*, *mknod* and *chmod* will be present in */bin* on the new 00.BOOT.

5.6. ADAPTING PROGRAMS TO USE EXTRA RAM

As distributed, the C compiler is tuned to work on even with the smallest Macintosh memory configuration. This may cause problems if you want to compile large source files. The first part of the C compiler proper, */usr/lib/cem*, as distributed, will compile most source files, but you may need to increase its memory allocation for larger source files.

You are strongly advised to execute the following procedure now if you have more than the minimal 1M of memory.

1. Boot MACMINIX and login as root.

2. Type:

```
cp /usr/bin/chmem /bin
/etc/umount /dev/hd0
/etc/hdclose /dev/hd0
/etc/eject
```

3. Insert 03.ACK in drive 0 and type:

```
/etc/hdopen 03.ACK:ACK  /dev/hd0
/etc/mount /dev/hd0 /usr
chmem +50000 /usr/lib/cem
```

A similar procedure can be executed if you encounter any other program that needs more memory. *chmem* takes a little getting use to, but it is difficult to avoid in a general-purpose multiprogramming system for a machine without a proper memory management unit.

5.7. USING A HARD DISK

The Macintosh version of MINIX is quite different from the other version in that it is not a *stand-alone* operating system. That is, the IBM and other versions completely take over the hardware once they begin execution, while MACMINIX runs in tandem with the normal Macintosh operating system, even depending on it for certain services, like accessing the hard disk, drawing and manipulating the menus, and drawing the tty windows. This has some drawbacks, especially with regard to speed, but it has the attraction that you can still have some of the things that Macintosh owner's like about their machine, such as menus and windows. In addition, if you have enough memory, you can run multifinder and simultaneously still use all your other Macintosh software.

However, the Macintosh file system is completely incompatible with the MINIX file system for a number of reasons, and therefore they do not share the same file name space. Instead, MACMINIX uses the Macintosh operating system to request it to set aside some number of (if possible, contiguous) disk blocks into a Macintosh file. The logical blocks of this file are then used as a MACMINIX disk partition.

You can have up to 9 of these Macintosh files (as distributed, that is, you can recompile the system to get more), and together they make up a logical MACMINIX disk. A MINIX partition can then mapped onto the file by means of the *hdopen* MINIX utility program.

If you so desire, the Macintosh files that make up the disk can also be backed up onto your tape or diskette with the rest of your Macintosh files, using your normal Macintosh backup software and later restored, if necessary. However, if you choose to do your backups in this way, you must remember that the *entire* Macintosh file will be backed up when any new information is written, so you may want to carefully consider where you put things.

Therefore, if you have a hard disk and have some available disk space, you can use it to keep (part of) the distributed diskettes on line. This section describes the steps to set up MINIX on such a system.

5.7.1. Step 1: Decide How Much of Your Disk Space to Devote to MINIX

The first decision you must make is how much of your disk you want to give over to MACMINIX. It is really not all that crucial that you be right the first time, since you can reclaim file space for the Macintosh operating system by using the finder to remove one or more of the files that correspond your MACMINIX "partitions." (Of course, you also lose the information on that "partition").

You can also create a new "partition" at any time (assuming you have the free disk space), make a MINIX file system on it, and then mount it for use by MACMINIX (see the *hdcreate*, lhdopen, *mkfs*, and *mount* manual pages). Keep in mind, however, that as distributed MACMINIX allows you to mount a maximum of five such partitions simultaneously.

5.7.2. Step 2: Decide How to Logically Partition Your Disk Space

Once you have decided how much disk space you want to use, you must decide how to split the space into logical disk partitions. This is entirely up to you, but you should probably create at least one small partition to hold the ROOT file system that is copied to the RAM disk at boot time.

Remember that how you logically partition your MACMINIX disk, and what you put on each partition, potentially has great impact on backing up your disk if you plan on doing so with ordinary Macintosh backup software. (You do back up your disk, don't you?) Also remember that as distributed MACMINIX will allow a maximum of 5 of these partitions to be mounted simultaneously.

5.7.3. Step 3: Build a Macintosh File for Each Partition

For each partition that you want, you must create the Macintosh file that will correspond to that partition. If, for example, you want a MINIX partition that is 160 blocks, boot MACMINIX as above, and type:

```
maccreate 160 harddisk:file
```

This will set aside a Macintosh file of 160 blocks (assuming you have 160 free blocks) that can be used a MINIX disk partition. When you are running the finder, with the normal Macintosh operating system, these 160 blocks will belong to the Macintosh file called *harddisk:file* and will have a MACMINIX file system icon on the desktop.

Follow this procedure for each logical MINIX partition you wish to create, changing the second and third parameters to *maccreate* as appropriate. You **must** substitute a complete, legal Macintosh file name for the second parameter. Remember or write down the size and file names for each partition, as you will need them in the next step.

5.7.4. Step 4: Make a MINIX File System on Each MINIX Partition

Now that your MINIX disk is logically partitioned, it is time to put a MINIX file system on each partition.

Let us assume, for example, that you have made a partition of 5000 blocks on the Macintosh file **harddisk:file1** and a partition of 20000 blocks on Macintosh file **harddisk:file2**. Then to create a file system on each, log in as root and type:

```
hdopen harddisk:file1 /dev/hd2
mkfs /dev/hd2 5000
hdopen harddisk:file2 /dev/hd3
mkfs /dev/hd3 20000
```

For the other MINIX partitions type the analogous commands.

You can verify that the file systems have been made by typing:

```
df /dev/hd2
df /dev/hd3
```

which will report on the i-nodes and blocks present on each file system. The total number of blocks should agree with the number you used in the *mkfs* command.

You can now mount your new file systems. To mount */dev/hd2* (partition 2) on */user*, type:

```
/etc/mount /dev/hd2 /user
```

To change to */dev/hd2*, type:

```
cd /user
```

This puts you in the root directory of the partition 2 file system.

5.7.5. Step 5: Initialize the Root File System

When MINIX boots, it needs a root file system. This file system can be a disk partition 0 or a RAM disk. If it is on a hard disk partition 0, then certain directories and special files must be created on that partition. If it is on RAM disk, then an image of the RAM disk must be created on partition 0. Either way, a root file system is needed on disk partition 0 (unless you have a diskette-only system). In the discussion below, we will assume that a RAM disk is being used.

The root file system normally has certain standard directories in it, to be described later. One of these, */dev*, contains all the character and block special files. To create the directories and special files, first change to the root directory, unmount all hard disk partitions that are currently mounted using */etc/umount*, then type:

/etc/setup_root root_file *ram hd1 hd2 hd3 hd4*

where *root_file* is the name of the Macintosh file that was used in the *hdcreate* command in the last step, *ram* is the size of the RAM disk in blocks (1K), and the next four numbers are the sizes of the four hard disk partitions, also in blocks. You must be logged in as root to run */etc/setup_root*. As an example, with *harddisk:file1* as the 2M Macintosh file name used in the *hdcreate* command, */dev/hd0* as the root device, and the four hard disk partitions being 32000, 32000, 2048, and 14000 blocks, respectively, you should type:

```
/etc/setup_root harddisk:file1 2048 32000 32000 2048 14000
```

You *must* specify all four partition sizes. If a partition has size zero, use 0.

At this point, the new hard disk root will contain the same files as the root file

system diskette. To try it out, type *sync*, select "Quit" from the "File" menu, copy the boot application to the hard disk, and reboot.

Having booted from the hard disk, you should type:

```
df
```

to see how much space is still available on the new root device. You need 100K to 200K free for scratch files in */tmp*, but if there is more than that available, you may wish to copy other files from */bin* on one of the other diskettes to the root. Files can also be copied from */lib*, but note that the C compiler expects to find all its passes in */usr/lib* rather than */lib*. This expectation can easily be changed by editing and recompiling *commands/cc.c*.

If the initial setup has copied files to the RAM image that you do not want there, you should remove them. After modifying the RAM image, do a *sync* and reboot to computer to see if all is well.

5.7.6. Step 6: Initialize */usr*

The next step is creating all the directories. A shell script called */etc/setup_usr* has been provided to do most of the work. It mounts the main hard disk partition and creates a large number of directories. Next, it copies files from the root file system and from diskettes to the */usr* tree on the hard disk. When it is finished, it asks for more diskettes to be inserted so it can copy files from them to the hard disk. Just follow the instructions that appear on the screen until the "Installation completed" message appears. To perform the installation be sure you are logged in as *root*. For instance, let's assume you created a 10M disk partition on Macintosh file *harddisk:file1*, and you now want to mount it on */dev/hd1*, and copy all the diskettes to it. To do this, proceed as follows:

1. Boot MACMINIX using 00.BOOT and 01.USR1 and login as root.
2. Type the following line:

   ```
   hdopen harddisk:file1 /dev/hd1
   mkfs /dev/hd1 10240
   /etc/mount /dev/hd1 /usr
   /etc/setup_usr /dev/hd1
   ```

3. Follow the instructions displayed by the *setup_usr* script.

Loading all the diskettes requires 9M. You can have a smaller partition if you install only the binaries (4M) onto the hard disk. In order to do so you should change the value of STOP on the third line of the */etc/setup_usr* script to 4, before issuing the commands above.

When this shell script finishes, the entire MINIX file system will be installed on the hard disk. Most of the files on the distribution diskettes are compressed files (with suffix *.Z*) or compressed archives (with suffix *.a.Z*). If, for some reason, installation fails part way through, you may be left with some *.a.Z*, *.a* or *.Z* files on the disk. A file *file.a.Z* can be decompressed using

```
compress –d file.a.Z
```

If the result is an archive (with suffix *.a*), you can extract the files from the archive with the *ar* command, for example:

```
ar x file.a
```

At this point the files *file.a.Z* and *file.a* can be removed. The only archive that you *must* keep as an archive is *libc.a* as the C compiler expects it this way. Do not extract the individual files from it!

From now on you can mount */dev/hd1* at boot time as */usr* by making a small change in */etc/rc* found on the ROOT file system. Use *mined* (see the section on editing below) to change the first three lines that read:

```
/bin/getlf "  Please insert /usr diskette in drive 0.  Then hit RETURN."
/etc/hdopen  01.USR1:USR1 /dev/hd0
/etc/mount /dev/hd0 /usr
```

by two lines that read:

```
/etc/hdopen  harddisk:file1  /dev/hd0
/etc/mount /dev/hd0 /usr
```

Inserting diskette 01.USR1 will no longer be necessary at boot time.

5.8. UNPACKING THE SOURCES

All MINIX sources, except the sources for the compiler and the editor, can be found on the SRC disks. These disks are normal MINIX file systems, which you can mount using the command

```
openfs 04.SRC1:SRC1 /dev/hd0
mount /dev/hd0 /user
```

The files on the distribution diskettes are compressed archives (with suffix *.a.Z*). If you want to extract the sources from a file *file.a.Z* you should first copy this file to either an empty floppy, or to the ram disk, if the latter is large enough. Your copy of *file.a.Z* can be decompressed using:

```
compress –d file.a.Z
```

After decompressing you can remove your copy of *file.a.Z*. Now you can extract the files from the archive with the *ar* command, for example:

```
ar x file.a
```

At this point all files from the archive are extracted, and the file *file.a* can be removed.

5.9. THE MENUS

There are four menus available to MACMINIX user. They are generally self-explanatory, but this section gives a brief description of each.

5.9.1. The Apple Menu

The Apple menu is similar to every other apple menu you have seen. Selecting the first item on the menu will bring up a simple dialog box, describing MACMINIX, while the rest of the items are for your desk accessories.

5.9.2. The Edit Menu

The Edit menu exists for the benefit of those desk accessories that can use it. None of the menu items are used by MACMINIX.

5.9.3. The File Menu

The File menu is used primarily to quit from MACMINIX or to configure various aspects of MACMINIX operation. Remember to sync the disks before quitting. A detailed explanation of your configuration options is given below.

5.9.4. The Windows Menu

The Windows menu is used to manipulate your windows. For example, there are menu items to enable you to rotate the windows, reopen them once you have closed them, or selecting individual windows to bring to the front.

5.9.5. The Debug Menu

Once you have MACMINIX running, some of the internal MINIX tables can be inspected by selecting items from this menu. Below is a table describing the name of the menu item and what it does.

PROCESS	dump of the process table
MEM MAP	dump of the memory map
CURRENT	dump of the status of the current process

The process table shows the current stack pointer, the CPU time spend in user mode and system mode, and the memory slot occupied for each process, including kernel tasks, as well as some other information.

The memory map shows (for user processes only) the location and length of the text, data and stack segments, as well as some other fields.

The status of the current process shows the register values most recently saved, a memory dump around the location of the program counter, and a memory dump around the location of the stack pointer. The memory dumps can be used to give a stack trace and, by manual disassembly, the instructions executed most recently.

5.10. SETTING CONFIGURATION OPTIONS

Selecting the "Configuration" menu item in the "File" menu will bring up a dialog box that allows you to set various operating parameters of MACMINIX. This section describes each of your options.

5.10.1. Heap Space

The dialog item entitled "Heap Space" allows you to specify how much of the application heap should be left by MACMINIX to support normal Macintosh operation, such as desk accessories and dialog boxes. The smaller you make this number, the more memory you will have available for MACMINIX processes. On the other hand, if this number is too small, the Macintosh operating system may run out of heap space, in which case the machine will crash. As distributed, this number is fairly generous so that it has the best chance of working with your configuration. You may want to experiment to see what is best for you. If you plan on using multifinder with MACMINIX, you can get away with making this number somewhat smaller, since desk accessories are not generally loaded into the currently running application's heap under multifinder.

5.10.2. Keyboard Mappings

There is also a check box labeled "Use Builtin Keyboard Mappings." As noted earlier, MACMINIX uses the Macintosh ROMs to set up the initial virtual keycode to ASCII mapping. If you prefer, however, you can select this check box and MACMINIX will use the mapping that has been compiled into the kernel. You may want to try this if you experience problems with your keyboard. As distributed, the US keyboard mapping has been compiled into the kernel.

5.10.3. The ROOT Partition

The final option you may set in this dialog box have to do with what Macintosh file is initially mapped to the MACMINIX hd0 disk partition. This is the partition used to initially read the ROOT file system. As distributed, this is set to "00.BOOT:ROOT" meaning that it will attempt to use the Macintosh file called ROOT on the volume named 00.BOOT. Clicking the mouse button on top of the box that displays the name will bring up a standard file dialog box, and you can select a new Macintosh file to use as the ROOT partition.

5.10.4. Effecting The Changes

Once you have made the desired changes to he configuration, the new configuration take effect the next time you boot MACMINIX. The "Cancel" button will cause any changes you made to be ignored.

You may also configure MACMINIX previous to booting by holding down the mouse button as you launch the boot application. If you do this, the configuration dialog box appears immediately, and you can set the various items as above. In this case, the new configuration is used immediately.

5.11. MACINTOSH SYSTEM CALLS

Supplied with MACMINIX is a partial interface with the Macintosh ROMs. You can find the include files in */usr/include/mac*, and the interface routines in */usr/lib/mac*. The library routines were built automatically from a program that uses the include file prototypes as a guide. A complete interface will be available sometime in the future.

A MACMINIX process may make calls to the ROMs, but please keep in mind that MACMINIX will not preempt a MINIX process when it has made a ROM call, since the ROMs are non-reentrant, and preempting may cause major problems.

5.12. RUNNING MACMINIX WITH MULTIFINDER

Before multifinder, there was not much of a distinction between the currently running application and the operating system. Since only one application could run at a time, the Macintosh operating system could be viewed simply as a set of support routines for the application. With the introduction of multfinder, this view changed somewhat, since now a Macintosh could have several applications in memory at any one time. The Macintosh operating system would transparently switch between them, although it could only do so at times when the application agreed to "give up" the processor.

MACMINIX will work with multifinder, giving up the processor at various times so that your other applications may run. In order to give MACMINIX a larger multifinder memory partition, set the memory size of the boot application the same way you do for any other application (see your owner's guide for a more complete description). There is one thing to remember here however and that is that MACMINIX does not run *at all* while another Macintosh application is running, so you may find that you have inconsistent results when running MINIX programs if they are time dependent.

5.13. TROUBLESHOOTING

As a user of MACMINIX you may be confronted with some of the error messages the system can produce. The following sections gives guidelines on you how to react. It also explains how you can use the built-in debugging aids.

5.13.1. Booting Problems

If you have problems there can be many causes, and listed here are some of possibilities.

5.13.2. Exhausted Heap Space

If you experience some unexplainable crashes, especially when you do simple things like selecting a menu, the Macintosh operating system may be running out of usable heap space. You can increase this with the "Configuration" menu item, described above. Since you might be having a problem getting running in the first place, you can bring up the configuration dialog box before anything else happens if you hold down the mouse button when you initially launch MACMINIX.

5.13.3. System Software Incompatibilities

If you are running an old version of the Macintosh system software, you may want to bring it up to date in order to minimize incompatibility problems.

5.13.4. Init Incompatibilities

If you experience unexplained crashes and are using some *inits* on your system, you may wish to temporarily remove them to see if it solves your problem. One or more of them may be incompatible with MACMINIX.

6

USING MINIX

By now you should have installed MINIX and gotten to the point where you can use it. In this chapter we will discuss some basic and some less basic points about using it. Once again, if you are not already reasonably familiar with using UNIX, you should first read one of the many books about it.

As a general rule, most aspects of MINIX work the same way as they do in UNIX. When you log in, you get a shell, which is functionally similar to the standard V7 shell (Bourne shell). Most programs are called the same way as in UNIX, have the same flags, and perform the same functions as their UNIX counterparts. The (default) keyboard editing conventions are also similar to V7 UNIX: the backspace key (CTRL-H) is used to correct typing errors, the @ symbol is used to erase the current input line, CTRL-S is used to stop the screen from scrolling out of view, CTRL-Q is used to start the screen moving again, and CTRL-D is used to indicate end-of-file from the keyboard., for example, to log out. These key bindings can be changed using the IOCTL system call and *stty* program, the same way as in UNIX.

6.1. MAJOR COMPONENTS OF MINIX

Although MINIX consists of hundreds of files, programs, and procedures, from the user's perspective, a few of them stand out as being especially important. In this section we will take a quick look at a few of the most important ones.

6.1.1. The Shell

The MINIX command interpreter is functionally identical to the Version 7 command interpreter, known as the **shell** (or the **Bourne shell** in honor of its inventor, S. R. Bourne). When a user logs in, the shell starts out by displaying the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types:

```
date
```

for example, the shell sees to it that the *date* program is run. When *date* finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file by typing, for example:

```
date >file
```

Similarly, standard input can be redirected, as in:

```
sort <file1 >file2
```

which invokes the sort program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

```
cat file1 file2 file3 | sort >outfile
```

invokes the *cat* program to con*cat*enate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file *outfile*.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently:

```
cat file1 file2 file3 | sort >outfile &
```

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on.

It is possible to collect several commands together in a file called a **shell script** and have them executed by just typing the name of the shell script. The shell also recognizes some programming constructs, such as *if*, *for*, *while*, and *case*, so it is possible to write shell scripts that act like programs. For more information about the MINIX shell, consult any book about the UNIX system because the MINIX and Bourne shells are practically indistinguishable to the user (although they are very different internally).

6.1.2. Editors

MINIX comes with several editors, among them a line-oriented editor called *ed*, a simple full-screen editor called *mined*, a powerful multifile, multiwindow editor called *elle*, and a clone of the well-known Berkeley *vi* editor, called *elvis*. People often have very strong, almost emotional, attachments to particular editors, so we have provided a wide choice. Try them all and see which you like best.

The *ed* editor is based on the V7 editor used on old mechanical teletypes. Although it still useful under certain circumstances, for daily use, it is rarely used nowadays.

In contrast, *mined* is a simple, but modern full-screen editor. Its greatest virtue is that it can be learned in about 10 minutes. When you type an ordinary ASCII character, that character is inserted on the screen (and in the file being edited) at the position of the cursor. This may sound obvious, but many editors require you to first enter a special "insert mode," enter the text, and then leave insert mode.

Commands to *mined*, such as moving the cursor or terminating the edit session, are handled by control characters, such as CTRL-F (go forward one word) or by the keys such as the four arrows on the numeric keypad at the right-hand side of the IBM PC keyboard. There are about three dozen commands in all, mostly chosen for their mnemonic value (e.g., CTRL-A moves the cursor to the start of the current line; CTRL-Z moves it to the end of the line).

Some of the commands move the cursor around the screen, scroll the screen forward or backward, or position it at the beginning or end of the file. Other commands delete text around the cursor (e.g., delete the word to the left or right of the cursor, or delete the tail of the current line). There are also commands available to manipulate blocks of text, such as deleting a block of text or saving it in a buffer to be copied to another part of the file. Finally, there are commands for searching forward or backward for a given text pattern, where the text pattern may contain a mixture of ordinary ASCII characters and "wild card" characters for matching sets of characters, end-of-line, and so on.

Another editor is *elle*, which can be thought of as a fast, simplified version of the famous *emacs* editor. It has about 100 commands, and can edit multiple files in full-screen or split-screen mode. Many sophisticated users regard *emacs* as the last word in editing.

Finally, there is *elvis*, an editor that has nearly all the features of the Berkeley *vi* and *ex* editors. All four editors have different properties. If you do not already have a preference, try them all until you find one you like.

6.1.3. The C Compiler

MINIX comes with a C compiler that accepts programs written in C as described in the Kernighan and Ritchie book. It also accepts many nonstandard features that are commonly used, but gives a warning message about each of them when asked

to. It also provides the standard header files normally provided with C compilers. The command:

```
cc prog.c
```

compiles the program on the file *prog.c* and leaves the executable binary program on a file called *a.out*.

The compiler knows about most of the standard C compiler flags, including **–c** (compile but do not link), **–o** (put the compiler output on a specific file instead of *a.out*), **–D** (define a macro), and **–I** (search a given directory for include files). Like the Version 7 compiler, this one also has a preprocessor for *#define*, *#include*, and *#ifdef* statements.

One minor difference between the MINIX compiler for the IBM PC and most other C compilers is that this one produces *.s* rather than *.o* files as a result of the **–c** flag. Furthermore, the assembler and linker are combined into a single program, *asld*, that reads a list of *.s* files and possibly some library archives, and produces an executable file. The members of the library archives are also *.s* files, although both they and the compiler output are compacted to save time and space. The programs *libpack* and *libupack* are provided to convert assembly language files from ASCII to compact format and back. The C compilers for the 68000 machines produce normal *.o* files.

6.1.4. The Utility Programs

MINIX comes with more than 175 utility programs. One rough grouping is to classify them into five categories as follows:

1. Compiler utilities.
2. File and directory manipulation.
3. Text file processing.
4. System administration.
5. Miscellaneous.

The compiler utilities are programs such as *make*, for keeping track of interdependent source and object files; *ar*, for maintaining libraries; and *size*, for determining the size of the various segments in a binary program.

The file and directory manipulation programs include *cat*, *cp*, *dd*, *mv*, and *pr*, for moving files around; *mkdir*, *rmdir*, and *ls*, for managing directories; and *chmod*, and *chown*, for dealing with protection.

A variety of programs are present for working with text files in addition to the editors, including the well-known filters *grep*, *rev*, *sort*, *tr*, *uniq*, and *wc*. The program *gres* searches a set of files for a pattern, and replaces occurrences with a given

pattern. The MINIX text justifiers are *roff*, and *nroff*, which have a wide variety of commands for controlling page layout.

Some utility programs deal with system administration. These include *df*, for determining how much space a file system has, *mkfs*, for making new file systems, *mount* and *umount* for attaching and detaching file systems to the main file tree, *passwd* for changing passwords, and *su* for becoming superuser.

The last category is for programs that do not fit in anywhere else. Among these are *date*, for setting and displaying the date and time, *pwd*, for printing the working directory, and *stty*, for setting the terminal parameters. There are many more.

6.1.5. The Library Procedures

MINIX also comes with over 200 library procedures that can be called from C programs. Like the utilities, these can also be divided into several rough groups:

1. System calls.
2. ANSI C procedures.
3. Miscellaneous.

The system call procedures allow C programs to issue system calls. There are more than 40 system calls available, including OPEN, READ, WRITE, CLOSE, LSEEK, PIPE, FORK, and EXEC. For almost every system call, there is a library procedure with exactly the same parameters and results as in Version 7. It should be possible to take almost any portable C program that runs under Version 7 and compile and run it on MINIX. Furthermore, most reasonable C programs written for other versions of UNIX should also work on MINIX, provided that they do not use any of the more bizarre system calls available in other versions and do not make any implicit assumptions about the sizes of integers and pointers (which are not the same on the 68000 versions of MINIX).

The second category is the set of procedures defined ANSI C. The collection is not yet complete, but most of the more common procedures are present, including standard I/O and string handling. Calls such as *fopen*, *fread*, *fwrite*, *fclose*, and *fprintf* are all present, as are *strcat*, *strcmp*, *strcpy*, and *strlen*, to name just a few.

The last category consists of a mixture of other procedures, which span a wide range, from encryption (*crypt*) to temporary file creation (*mktemp*).

6.1.6. Relation with Other Operating Systems

Like UNIX, MINIX is a complete operating system. It does not require any other operating system to help it. On the IBM PC, Atari, and Amiga, when MINIX is running it takes over the entire computer and runs on the bare hardware. For the Macintosh version, this is not true. There MINIX runs as a user program on top on

the Macintosh operating system. Since MINIX has different system calls than MS-DOS, TOS, and AMIGA-DOS, it is not possible to run programs written for other operating systems on MINIX.

Nevertheless, it is possible to partition your hard disk with one or more partitions for MINIX and one or more partitions for other operating systems. Furthermore, you can transport files back and forth between MINIX and MS-DOS, TOS, or AMIGA-DOS, using utility programs have been provided for this purpose. The utilities reside in */usr/bin* and are invoked in the usual way, by just typing their names and arguments. The first program for the IBM PC, *dosdir*, reads an MS-DOS diskette and tells what is on it. The program can also be told to list a specific directory on the diskette.

The second program, *dosread*, reads a file from an MS-DOS diskette and copies it to standard output, which, of course, can be redirected to a file. When the **–a** flag is given, the MS-DOS conventions for ASCII files are converted to the MINIX conventions, so the resulting file appears to be a normal text file.

The third program, *doswrite*, copies its standard input to a diskette containing an MS-DOS file system, again doing format conversion if requested. It does not create directories, however, so all the necessary directories must be in place on the diskette when it is inserted into the drive. As an aside, these three programs are all links to the same file, which checks to see how it was called to see what it should do.

For the 68000-based systems, analogous programs are provided to get files back and forth.

6.2. PROCESSES AND FILES IN MINIX

Two key concepts in MINIX are processes and files. Since these may not be immediately familiar to people who are accustomed to other operating systems, below we give a brief introduction to them. Processes and files are accessed by **system calls**, services provided by the operating system. Some of the more important process and file system calls will be discussed below.

6.2.1. Processes

A **process** is basically a program in execution. It consists of the executable program, the program's data and stack, its program counter, stack pointer, and other registers, and all the other information needed to run the program.

The easiest way to get a good intuitive feel for a process is to think about a timesharing system. Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

When a process is temporarily suspended like this, it must later be restarted in

exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, if the process has several files open, the exact position in the files where the process was must be recorded somewhere, so that a subsequent READ given after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among other things.

The key process management system calls are those dealing with the creation and termination of processes. Consider a typical example. The shell reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 6-1.

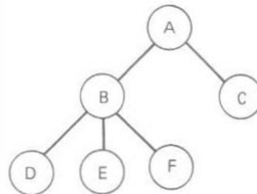

Fig. 6-1. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

Other process system calls are available to request more memory (or release unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for it. For example, a process that is communicating with another process on a different computer does so by sending messages over a network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends a **signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal

handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was just before the signal. Signals are the software analog of hardware interrupts, and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use MINIX is assigned a **uid** (user identification) by the system administrator. Every process started in MINIX has the uid of the person who started it (except for so-called setuid programs). A child process has the same uid as its parent. One uid, called the **superuser**, has special power, and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

6.2.2. Files

A major function of the operating system is to hide the peculiarities of the disks and other I/O devices, and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be opened, and after it has been read it should be closed, so calls are provided to do these things.

In order to provide a place to keep files, MINIX has the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he was taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his computer games. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file system, as shown in Fig. 6-2.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than three levels is unusual), whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally a few minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 6-2, the

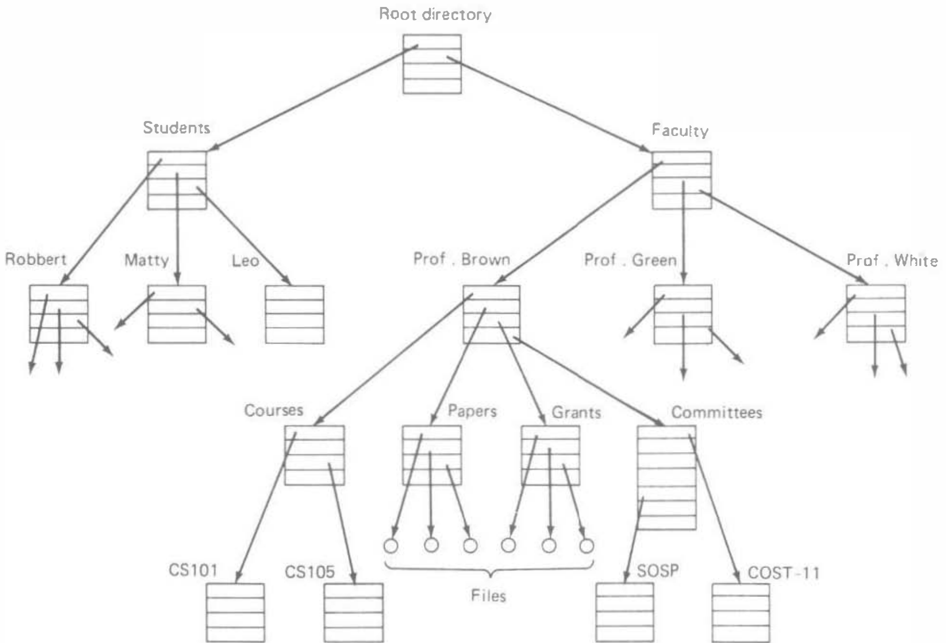

Fig. 6-2. A file system for a university department.

path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory (as opposed to a relative path starting at the working directory).

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. In Fig. 6-2, if */Faculty/Prof.Brown* were the working directory, then use of the path name *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Files and directories in MINIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), and one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x--x* means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer

called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in MINIX is the **mounted file system**. To provide a clean way to deal with removable media (e.g. diskettes), MINIX allows the file system on a diskette to be attached to the main tree. Consider the situation of Fig. 6-3(a). Before the MOUNT call, the RAM disk (simulated disk in main memory) contains the primary, or **root file system**, and drive 0 contains a diskette containing another file system.

However, the file system on drive 0 cannot be used, because there is no way to specify path names on it. MINIX does not allow path names to be prefixed by a drive letter or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the MOUNT system call allows the file system on drive 0 to be attached to the root file system wherever the program wants it to be. In Fig. 6-3(b) the file system on drive 0 has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had contained any files they would not be accessible while drive 0 was mounted, since */b* would refer to the root directory of drive 0. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.)

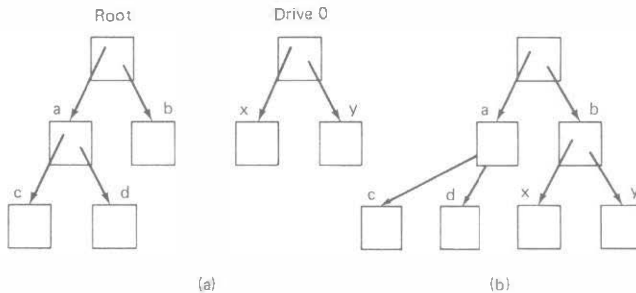

Fig. 6-3. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

Another important concept in MINIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Programs that do system maintenance often need this facility. Access to special files is controlled by the same *rwx* bits used to protect all files, so the power to directly access I/O devices can be restricted to the system administrator, for example.

Character special files are used to model devices that consist of character streams, rather than fixed-size randomly addressable blocks. Terminals, line

printers, and network interfaces are typical examples of character special devices. The normal way for a program to read and write on the user's terminal is to read and write the corresponding character special file. When a process is started up, file descriptor 0, called **standard input**, is normally arranged to refer to the terminal for the purpose of reading. File descriptor 1, called **standard output**, refers to the terminal for writing. File descriptor 2, called **standard error**, also refers to the terminal for output, but normally is used only for writing error messages.

All special files have a **major device number** and a **minor device number**. The major device number specifies the device class, such as diskette, hard disk, or terminal. The minor device number specifies which of the devices in the class is being addressed, for example, which diskette drive. All devices with the same major device number share the same device driver code within the operating system. The minor device number is passed as a parameter to the device driver to tell it which device to read or write. The device numbers can be seen by listing */dev* with the *ls –l* command.

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudo-file that can be used to connect two processes together, as shown in Fig. 6-4. When process *A* wants to send data to process *B*, it writes on the pipe as though it were an output file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in MINIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call.

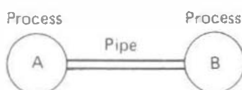

Fig. 6-4. Two processes connected by a pipe.

6.3. A TOUR THROUGH THE MINIX FILE SYSTEM

The MINIX file tree is organized the same way as the standard UNIX file tree. The standard MINIX file system contains the following directories:

Name	- Description
/bin	- Most common system binaries can be copied here from */usr/bin*
/dev	- Special files for I/O devices
/etc	- Miscellaneous system administration
/doc	- Place to put (user-supplied) online documentation
/lib	- Most common libraries can be copied here from */usr/lib*
/tmp	- Some utilities generate their temporary files here

/user	- Empty; can be used for mounting file systems
/usr	- Root of the user file system (usually mounted file system)
/usr/adm	- The */usr/adm/wtmp* file records logins
/usr/ast	- Home directory for user *ast*
/usr/bin	- System binaries are kept here
/usr/etc	- Main system administration directory
/usr/include	- System header files
/usr/include/minix	- MINIX-specific header files
/usr/include/sys	- More header files
/usr/lib	- Libraries, compiler passes, miscellanea
/usr/lib/tmac	- Holds macro packages for *nroff*
/usr/man	- Place to put user-written manual pages for *man* (if any)
/usr/spool	- Holds specialized spooling directories
/usr/spool/at	- Spooling directory for the *at* program
/usr/spool/lpd	- Spooling directory for line printer daemons (future)
/usr/spool/mail	- Spooling directory for local mail
/usr/spool/uucp	- Spooling directory for *kermit* and *uucp* (future)
/usr/src	- Start of the source tree
/usr/src/commands	- Sources for the utility programs (has many subdirectories)
/usr/src/fs	- Sources for MINIX file system
/usr/src/lib	- Holds library directories
/usr/src/lib/amiga	- Sources for Amiga-specific procedures
/usr/src/lib/ansi	- Sources for ANSI C procedures
/usr/src/lib/atari	- Sources for Atari-specific procedures
/usr/src/lib/ibm	- Sources for IBM PC-specific procedures
/usr/src/lib/mac	- Sources for Macintosh-specific procedures
/usr/src/lib/other	- Sources for other library procedures
/usr/src/lib/posix	- Sources for procedures required by POSIX
/usr/src/lib/string	- Sources for IBM assembly code string procedures
/usr/src/kernel	- Sources for MINIX kernel
/usr/src/mm	- Sources for MINIX memory manager
/usr/src/test	- Sources and binaries for testing MINIX
/usr/src/tools	- Utilities for building MINIX boot diskettes
/usr/tmp	- Alternative directory for temporary files

Let us briefly examine some of these directories. In */bin* we find the most heavily used programs such as *cat*, *cp*, and *ls* as well as some programs such as *login* and *sh* needed to bring the system up. If */bin* is being kept on RAM disk, it will normally contain a subset of */usr/bin*. The idea of putting it on the RAM disk is to speed up access, of course. If a RAM disk is not being used, it is not necessary to put any files in *bin* other than the ones it comes with.

The directory */dev* contains the special files for the I/O devices, including most of the following, although not every one is present in each version. Ethernet is not

supported on the 68000, for example. Also, */dev/hd5-9* are for an (optional) second hard disk.

Name	#	Description
/dev/ram	1, 0	- RAM disk
/dev/mem	1, 1	- Absolute memory
/dev/kmem	1, 2	- Kernel memory
/dev/null	1, 3	- Data written here vanishes; reads yield end of file
/dev/port	1, 4	- Access to I/O ports
/dev/fd0	2, 0	- Diskette drive 0
/dev/fd1	2, 1	- Diskette drive 1
/dev/hd0	3, 0	- IBM: Entire hard disk 0; Atari: boot block
/dev/hd1	3, 1	- Hard disk 0, partition 1
/dev/hd2	3, 2	- Hard disk 0, partition 2
/dev/hd3	3, 3	- Hard disk 0, partition 3
/dev/hd4	3, 4	- Hard disk 0, partition 4
/dev/hd5	3, 5	- IBM: Entire hard disk 1; Atari entire hard disk 0
/dev/hd6	3, 6	- Hard disk 1, partition 1
/dev/hd7	3, 7	- Hard disk 1, partition 2
/dev/hd8	3, 8	- Hard disk 1, partition 3
/dev/hd9	3, 9	- Hard disk 1, partition 4
/dev/console	4, 0	- Terminal 0 (main keyboard and screen)
/dev/tty0	4, 0	- Same as */dev/console*
/dev/tty1	4, 1	- RS232-C port 1
/dev/tty2	4, 2	- RS232-C port 2
/dev/tty	5, 0	- Current terminal
/dev/lp	6, 0	- Line printer (Centronics port)
/dev/net0	7, 0	- Ethernet

(The IBM diskette combinations are given in Chap. 2.) When */dev/ram* is opened and read, for example, by the command

```
od –x /dev/ram
```

the contents of the RAM disk are read out, byte by byte, starting at byte 0. Similarly, reading */dev/mem* reads out absolute memory, starting at address 0 (the interrupt vectors). The file */dev/kmem* is similar to */dev/mem*, except that it starts at the address in memory where the kernel is located. The next file, */dev/null*, is the null device. It is used as a place for redirecting program output that is not needed. Data copied to */dev/null* are lost forever. The final file in this group, */dev/port*, is used to access I/O ports in protected mode on the 80286 and 80386 CPUs.

The next group of files are for the diskette drives, with different names provided for different sizes (see Chap. 2).

Next come the special files for the hard disks. The first one refers to the entire device, with regard for the partition structure on it. It is occasionally used for

reading the boot block, or for copying one raw hard disk to another. The other entries refer to specific partitions. They are used in commands such as *df* to examine the amount of available space on a partition.

Groups 4 and 5 are for the terminals. The */dev/ttyX* entries are used to access a specific device, such as a modem or serial printer. In contrast, */dev/tty* refers to the current terminal, whatever its number may be.

The character special file */dev/lp* is for the line printer. It is write only. Bytes written to this file are sent to the line printer without modification (to make it possible to send escape sequences to graphics printers). Users normally print files by using the *lpr* program, rather than copying files directly to */dev/lp*. The latter method takes care of converting line feed to carriage return plus line feed, expanding tabs to spaces, etc., whereas the former method does not. Finally, */dev/net* is for networking.

To prevent problems, it is recommended that you remove entries in */dev* that correspond to nonexistent devices. For example, if you have only 1 diskette drive, you should remove */dev/fd1*, etc to eliminate the possibility that you inadvertently use one of them and thus hang the system (which will patiently wait until you insert a diskette in drive 1). If you have only 360K drives, you can remove */dev/atX*, but if you have 1.2M drives, you should *not* remove */dev/fdX* since they are needed when using 360K diskettes in your 1.2M drive.

Another important directory is */etc*. This directory contains files and programs used for mounting and unmounting file systems, the system profiles, the termcap data base, and general system files. For users who have */etc* on the RAM disk, */usr/etc* can be used to maintain a copy on the */usr* partition.

The directory */lib* holds libraries, such as *libc.a*, passes of the C compiler that are not normally directly called by users, and certain miscellaneous files related to compiling. As with *bin*, the full set of programs is kept in */usr/lib*, and the most important ones copied into */lib*. Please note that the *cc* program, which calls the compiler passes, has built-in path names using */usr/lib*. If you want to install parts of the compiler in */lib*, you will have to edit *cc* and recompile it.

The */tmp* and */usr/tmp* directories are used by many programs for temporary files. By putting */tmp* on the RAM disk, these programs are speeded up.

The directories */user* and */usr* are empty. They should be used for mounting file systems. Frequently, */usr* will be partition 1 or 2 of the hard disk, and will contain all the directories listed above, including all the sources.

6.3.1. Mounted File Systems

When MINIX is first started up, the only device present is the root device (default: RAM disk). After the files and directories that belong on the root device are copied there from the root file system diskette, MINIX prints a message asking the user to remove the diskette. It then executes the shell script */etc/rc* as the final step in bringing up the system.

The file */etc/rc* first prints a message asking the user to put the */usr* diskette in drive 0. Then it pauses to allow the diskette to be inserted and the date entered. The shell script now executes the command:

```
/etc/mount /dev/fd0 /usr
```

to mount the system disk on */usr*. From this point on, all the files in */usr*, including the binary programs in */usr/bin*, are available.

On a PC with two diskette drives and no hard disk, you should insert a file system diskette in drive I and type:

```
/etc/mount /dev/fd1 /user
```

If you want to mount the same diskette in drive 1 whenever the system is brought up can modify */etc/rc* to perform the mount on drive 1 analogously to the mount on drive 0. Alternatively, a hard disk partition can be mounted. Note, however, that changes made to */etc/rc* on the RAM disk will be lost when the system is next booted unless they are also made to the root file system diskette, which can be mounted and modified, just like any other diskette.

If it is desired to remove the diskette in drive I during operation, first type the command:

```
/etc/umount /dev/fd1
```

and wait for the prompt. (Note that the program is called *umount*, just as it is in UNIX, not *unmount.)* There is no *n* in *umount*. You cannot unmount a device holding the working or root directory of any process, or which is otherwise in use.

If you remove a diskette while it is still mounted, the system may hang, but it can be brought back to life by simply re-inserting the same diskette. If you remove a diskette while it is still mounted and insert another in its place, the contents of both file systems will be seriously damaged and information may be irretrievably lost (see below about repairing damaged file systems). Experienced MS-DOS users who are used to constantly switching diskettes without telling the operating system should post discrete KEEP OFF signs on their drives as a reminder.

Although it is permitted to *insert* a non-MINIX diskette in a drive (e.g., to read an MS-DOS diskette), only MINIX file system diskettes can be *mounted*. Attempts to mount a diskette not containing a MINIX file system will be detected and rejected.

6.4. HELPFUL HINTS

In this section we will point out several aspects of MINIX that will frequently be useful. Most of these relate to areas in which MINIX is different from UNIX, so even experienced UNIX users should read it carefully.

6.4.1. Making Backups

As a starter, it is wise to back up your files periodically. To make a backup, first format a diskette as you usually do. If you want to back up a diskette and you have two diskette drives, unmount the file systems in drives 0 and 1. It is possible to back up a mounted file system, but only if no background processes are running. To be doubly safe, give a *sync* command. Insert the newly formatted diskette in drive 1, and then type:

```
cp /dev/fd0 /dev/fd1
```

to copy information from drive 0 to drive 1, assuming you want to copy 360K diskettes. For 1.2M diskettes on the IBM PC, use */dev/at0* and */dev/at1*. When the drive lights go out, the diskettes can be removed.

If you have one diskette drive and a hard disk, to back up a diskette, insert the diskette to be backed up, and copy it to the hard disk. Then insert the new diskette and copy the image back. The following three commands will do the job:

```
cp /dev/fd0 /usr/tmp/image
cp /usr/tmp/image /dev/fd0
rm /usr/tmp/image
```

This command sequence presumes that enough free space exists in */usr/tmp*.

To back up a hard disk, it is best to do it directory by directory. Format enough blank diskettes, and put empty file systems on them using *mkfs*. Mount one of these diskettes. Then use the *backup* program. For example, one might use the sequence:

```
mkfs /dev/fd0 360
/etc/mount /dev/fd0 /user
backup -jmvz /usr/ast /user/ast
/etc/umount /dev/fd0
```

The *backup* program has a variety of useful flags. The **-j** flag suppresses the copying of useless junk, like old core images. The **-m** flag is used to backup large directories over multiple diskettes. The **-v** flag enables verbose mode. In this mode the names of the files are printed as they are backed up. Finally, the **-z** flag arranges for *compress* to be called to compress the files as they are backed up. While compression slows up *backup* considerably, it also doubles the effective capacity of each diskette. Note that *backup* also backs up all the subdirectories in the directory it is working on (i.e., it is recursive).

Suppose a directory is backed up onto a diskette Monday evening. On Tuesday, a number of files are changed in that directory. If the backup diskette from Monday is mounted (instead of a blank diskette) and *backup* called, only those files that have changed since the previous backup will be copied. Be sure to use the same flags (i.e., do not mixed compressed and uncompressed).

6.4.2. Printing

Files can be printed using the *lpr* program. It can be given an explicit list of files, as in

```
lpr file1 file2 file3 &
```

If no arguments are supplied, *lpr* prints its standard input, for example:

```
pr file1 file2 file3 | lpr &
```

Note that *lpr* is not a spooling daemon. It sits in a loop copying files to */dev/lp*. For this reason, it should be started off in the background with the ampersand, so the user can continue working while printing is going on. Only one *lpr* at a time may be running.

6.4.3. Checking on Disk Space

Disk space always seems to be in short supply, no matter how big the disks are. To find out how much space and how many i-nodes are left on diskette 0, type:

```
df /dev/fd0
```

Similar commands can be used for other devices, including */dev/ram* and the hard disk partitions. When *df* is called with no arguments, it checks */etc/mtab* and prints the statistics for the root device and all mounted file systems.

6.4.4. Profiles

When you log in, the shell checks to see if there is a file *.profile* in your home directory. If it finds one, it executes the file as a shell script. This file is commonly used to set shell variables, *stty* parameters, and so on. See */usr/ast/.profile* as a simple example. The system profile, */etc/rc* is executed when MINIX is booted.

6.4.5. Stack Size

The IBM PC does not have any protection hardware. Neither do the Atari, Amiga, or Macintosh. As a result, if a program's stack overruns the area available for it, it will overwrite the data segment. This usually results in a system crash. When a program crashes unexpectedly or acts strange, it is probably worthwhile to find out how much memory is allocated for it (see the "memory" column in the output of *size*). In many cases, increasing the stack space with *chmem* will make it work again. On the IBM PC, the largest executable program has 64K instruction space and 64K data space; the 68000 versions have no limit. To get separate instruction and data spaces, the **–i** flag should be used when compiling programs. When working with unreliable programs, doing *sync*s frequently is advisable.

The problems with memory allocation are due to a large chunk of memory being taken up by the operating system, its buffers, and the RAM disk, plus the fact that multiple programs can be running at once. This, plus the lack of hardware protection, requires that a more economical approach be taken to memory use than the standard MS-DOS method of just giving each program the whole machine to itself. In practice, once the sizes have been set right for a given configuration, they need not be fiddled with any more.

It sometimes happens that a program (or a compiler pass) cannot be executed due to lack of memory for it. When this happens, the shell may a message of the form *program: cannot execute*. The solution is to run fewer programs at once, or reduce the program's size with *chmem*. The amount of stack space assigned to the shell, *make*, etc. in the standard distribution may not be optimal for all applications. Change it if problems arise. To see how much is currently assigned, type

```
size /usr/bin/* | mined
```

In general, if a program goes berserk or the compiler gives nonsensical error messages, the first thing to suspect is stack overrun, which can be tackled with *chmem*.

6.4.6. Compilation Problems

Space is often tight, especially when the amount of program memory is only 512K. It can happen that the C compiler fails due to lack of space, in which case the –F flag should be used.

Although an individual compilation can get into space problems, far more likely is that *make* will be unable to run the compiler. The problem is that in addition to the login shell and make itself, several other programs may be running simultaneously, including other shells started by *make*. If problems arise, several approaches can be taken. One is to run:

```
make –n >s
sh s
```

to find out what *make* wants to do, put it on a shell script, and then execute it without *make*. Often this helps.

Another method is to fiddle with the stack sizes of *make*, *sh*, and the compiler passes, *cpp*, *cem*, *opt*, *cg*, and *asld* (some of which can be found in */usr/lib*). By reducing the stack allocated to some of these programs using *chmem* it is frequently possible to solve the problem. Of course if they are given too little stack, they may go berserk. Thus fine tuning the sizes requires some patience.

One last note in this regard, sometimes it is necessary to do something as *root*. There are two ways to become *root*: to log in as *root* and to use the *su* program. They are not quite identical. When using *su* an additional shell is created, taking up memory. If space problems occur after having become *root* using *su*, it is best to hit CTRL-D twice to log out, then log in as *root* directly.

6.4.7. Temporary Files

Several of the utility programs, including the C compiler, create their temporary files in */tmp*, on the RAM disk. If the RAM disk fills up, a message will be printed on the terminal. The first thing to do is check */tmp* to see if there is any debris left over from previous commands, and if so, remove it. If that does not solve the problem, temporarily removing some of the larger files from */bin* or */lib* will usually be enough. These files can be restored later by mounting the root file system on any drive and copying the needed files from it. In a pinch, you can mount a diskette on */tmp* to provide more space for a command that needs a lot of it. When *cc* fills up */tmp*, the **–T** flag can be used to put the temporary files in another directory.

6.4.8. Aborting Commands

MINIX, like UNIX, will not break off a system call part way through just because the DEL key has been struck. When the system call in question happens to be an EXEC, which is loading a long program from a slow diskette, it can take a few seconds before the shell prompt appears. Be patient. Hitting DEL again makes things worse, rather than better.

6.4.9. System Status Reporting

Although it is really intended as a debugging aid, rather than a permanent part of the system, on the IBM PC version the F1 and F2 function keys cause dumps of some of the internal tables to be printed on the screen. (For the 68000s, other keys are used, as described later.) F1 gives a dump like *ps*, but instantly. Frequently, the system appears to be stopped, but it is actually thinking its little head off and using the RAM disk, which, unlike the other disks, is not accompanied by whirring and clicking noises and flashing lights. The nervous user can press F1 to see the internal process table to verify that progress is still being made. The F1 and F2 keys are intercepted directly by the keyboard driver, so they always work, no matter what the computer is doing. The values in the columns *user* and *sys* are the number of clock ticks charged to each process. By hitting F1 twice, a few seconds apart, it is possible to see where the CPU time is going.

6.4.10. Escape Sequences

MINIX supports ANSI escape sequences as well as Berkeley termcap entries. The latter can be found in the file */etc/termcap*. The entries use the ANSI escape sequences. The *TERM* variable should be set to *minix* to use these entries. A library routine, *termcap.c* is provided to manipulate them.

6.4.11. Serial Lines

Communication with the outside world over a modem is possible. The number of RS232 ports supported in controlled by the constant *NR_RS_LINES* defined in *kernel/tty.h*. This constant should be set to the proper number of ports for your configuration, since each port requires about 1K of table space in the kernel. To log into other systems or transfer files, see the manual pages for *kermit*, *rz*, *sz*, and *term*. On the Atari, *stterm* is also available.

6.4.12. Transferring Files to and from Other Operating systems

It is possible to copy files from an MS-DOS disk to MINIX or vice versa. See the description of *dosread* and *doswrite* for details. Similarly, see *tosread* and *toswrite* for the Atari, *macread* and *macwrite* for the Macintosh, and *transfer* for the Amiga.

6.4.13. Keyboard Mapping

The ASCII codes produced by the IBM PC keyboard are determined by software, not hardware. A mapping has been chosen to try to produce a unique value for each key, so programs can see the difference between, for example, the + in the top row and the + in the numeric keypad. Since the keyboards of the various machines differ, the mappings are not identical. To see which code or codes a given key produces, use *od –b*, and then type the key or keys followed by a carriage return and a CTRL-D.

6.5. SYSTEM ADMINISTRATION

Since MINIX is in principle a multiuser timesharing system, not unlike what large computer centers run, you will have to learn how to administer your system. Fortunately, doing this is not difficult. System administration tasks have to be done by the superuser. Superusers have more power than ordinary users. They can violate nearly all of the system's protection rules. Although there is no Hippocratic Oath for superusers (yet), tradition requires them to exercise their great power with care and responsibility. Superusers get a special prompt (#), to remind them of their awesome power.

To become superuser, login as *root* using the password *Geheim*. (Notice the capital *G*). Alternatively, use the *su* program with *Geheim* as password. Please take note that these two methods of becoming superuser are not quite the same. Using *su* causes an extra shell to be created. If you are short on memory, and intend to do something complicated as superuser (such as running a large *make* job), you may have to log out and log in again as *root*.

6.5.1. Making New File Systems

One of the things that superusers do is make new file systems. This is possible using the program *mkfs* (make file system). To make an empty 360 block file system on diskette 0, type:

```
mkfs /dev/fd0 360
```

When the program finishes, the file system will be ready to mount. On a system with only one diskette drive and no hard disk, *mkfs* will first have to be copied to */bin*, (on the RAM disk), the */dev/fd0* file system unmounted, a blank diskette inserted into drive 0 and then the file system made.

It is also possible to make a file system that is initialized with files and directories. A command for doing this is:

```
mkfs /dev/fd0 proto
```

where *proto* is a prototype file. The manual entry for *mkfs* (in Chap. 8) gives an example of a prototype file.

6.5.2. File System Checking

File systems can be damaged by system crashes, by accidently removing a mounted file system, by forgetting to run *sync* before shutting the system down and in other ways. Repairing a file system by hand is a tricky business, so a program, called *fsck*, has been provided to automate the job. It is best to first copy *fsck*, to the root file system and then unmount the file system to be repaired, unless it is the root file system. If the root file system is on a hard disk partition, it is best to reboot MINIX and run *fsck* from a diskette so that the root file system is unmounted while *fsck* is modifying it.

The simplest way to repair a file system is to run *fsck* in automatic mode. To repair */dev/hd1*, for example, just type:

```
cd /
cp /usr/bin/fsck /fsck
/etc/umount /dev/hd1
fsck -a /dev/hd1
/etc/mount /dev/hd1 /usr
```

Fsck will run, ask some questions, answer its own questions, and fix everything. When it is done, you can remount the repaired file system and continue. Other options are described in the manual page for *fsck*.

6.5.3. The /etc Directory

The */etc* directory contains several files that superusers should know about. They are:

Name	**- Description**
gettydefs	- Used for configuring dial in lines using modems
group	- Contains names of the user groups
message	- Message of the day
passwd	- Password file
rc	- Shell script executed after the system is booted
setup_move	- Additional hard disk setup (remove after installation)
setup_root	- Used to set up the hard disk RAM image (remove after installation)
setup_usr	- Used to set up the hard disk partition for */usr* (remove after installation)
termcap	- Berkeley termcap entries for MINIX
ttys	- Enables/disables RS-232 ports for use as terminals
ttytype	- Terminal configuration

Probably the most important of these is the password file, */etc/passwd*. You can enter new users by editing this file and adding a line for each new user. The entry for a user named *fozzie* might be:

```
fozzie::15:1:Fozzie the Bear:/usr/fozzie:/bin/sh
```

The entry contains seven fields, separated by colons. These fields contain the login name, password (initially null), uid, gid, name, home directory, and shell for the new user. When a new user is entered, the corresponding home directory must also be created, using *mkdir*, and its owner set correctly, using *chown* and *chgrp*. Each user must have a unique uid, but the numerical values are unimportant. It is probably adequate to put all ordinary users in group 3, unless there really are distinct groups of users. When the new user logs in for the first time, he should choose a password and enter it using *passwd*.

Another important file is */etc/rc*. Each time the system is booted, this file is run as a shell script just before the

```
login:
```

message is printed. It can be used to mount file systems, request the date, erase temporary files, and anything else that needs to be done before starting the system. It also forks off *update*, which runs in the background and issues a SYNC system call every 30 seconds to flush the buffer cache.

If you do not have a hard disk and want to use two diskette drives during normal operations, it may be convenient to modify */etc/rc* to mount */dev/fd1* on */user* during system boot. If you do this, you can also change */etc/passwd* to put your home directory on */user* instead of */usr*. Of course you can also change */etc/rc* to mount a hard disk partition.

The file */etc/ttys* contains one line for each terminal in the system. During startup, *init* reads this file and forks off a login process for each terminal. When the console is the only terminal, *ttys* contains only 1 line.

Also contained in */etc* are the programs *mount*, and *umount* for mounting and unmounting file systems, respectively.

When any of the files on the RAM disk, such as */etc/passwd*, are modified, the changes will be lost when the system is shut down unless the modified files are explicitly copied back to the root file system. This can be done by mounting the root file system diskette and then copying the files with *cp*.

6.5.4. Miscellaneous Notes

A few MINIX programs can only be executed by the superuser. Some of these, such as *df*, are owned by the root and have the SETUID bit on, so that when they are executed, the effective uid is that of the superuser, even though the real uid is not.

In general, if a program, *prog*, needs to run as the superuser but is to be made generally available to all users, it can be made into a SETUID program owned by the root by the command lines:

```
chown root prog
chmod 4755 prog
```

Needless to say, only the superuser can execute these commands. The shell script *fixbin* can be run by the superuser to set all the permissions (and sizes) as follows:

```
fixbin /bin /bin
```

If the two arguments are different, the executable files will be first copied from the first directory to the second one.

7

RECOMPILING MINIX

This chapter is intended for those readers who wish to modify MINIX or its utilities. In the following pages we will tell what the various files do and how the pieces are put together to form the whole. It should be emphasized that if you simply intend to use MINIX as distributed, then you do not have to recompile the system and you do not have to read this chapter. However, if you want to make changes to the core of the operating system itself, for example, to add a device driver for a streamer tape, then you should read this chapter.

7.1. REBUILDING MINIX ON AN IBM PC

Although this section is specifically for IBM PC users, it should also be read carefully by everyone interested in recompiling MINIX. Most of what is said here applies to all versions of MINIX. The sections about other processors mostly discuss the differences between recompiling MINIX on an IBM PC and on another system.

The MINIX sources are contained in the following directories, normally all subdirectories of */usr/src* except for *include* which goes in */usr/include*:

Directory	Contents
include	The headers used by the commands (has two subdirectories)
kernel	Process, message, and I/O device handling
mm	The memory manager
fs	The file system
tools	Miscellaneous tools and utilities
test	Test programs
lib	Libraries (has several subdirectories)
commands	The utility programs (has many subdirectories)

Some of the directories contain subdirectories. If you are working on a hard disk, be sure that all these directories have been set up, and all files copied there from the distribution diskettes and decompressed and dearchived. If you do not have a hard disk, format and make empty file systems on five diskettes. On the first one, make a directory *kernel* and copy all the kernel files to it. In a similar way, prepare diskettes for *fs*, *mm*, *tools*, and *test* as well. If you do not have a hard disk, there are still three ways you can recompile the system. First, if you have two diskette drives, use drive 0 to hold the root file system, including the compiler, */usr/lib* and */usr/include*. Diskettes with programs to be compiled are mounted on drive 1.

Second, if you have a sufficiently large RAM disk (at least 512K), you can put the root file system there, along with the compiler, */usr/lib* and */usr/include*.

Third, if you have no hard disk, one diskette drive and insufficient memory for a 512K RAM disk, you should have at least a 1.2M diskette drive in which you can put the root file system, although in a pinch a 720K diskette might work with a lot of shoehorning. If you use this approach, each of the five diskettes made above must contain enough of */usr/bin*, */usr/lib*, and */usr/include* to allow compilation of the kernel, file system, or whatever other files are on that disk. With only 640K RAM and a single 360K diskette, it is not possible to recompile the system. Expanded memory (LIM standard) is not supported and cannot be used as a RAM disk.

As a test to see if everything is set up properly, type in, compile, and run the following program:

```
#include <limits.h>
main()
{
   printf("PATH_MAX = %d\n", PATH_MAX);
}
```

It should print out the value 255 for *PATH_MAX*. If it fails to compile, be sure that the file */usr/include/limits.h* is installed and readable.

7.1.1. Configuring the System

The file */usr/include/minix/config.h* contains some user-settable parameters. Examine this file and make any changes that you require. For example, if *LINEWRAP* is set to 0, then lines longer than 80 characters will be truncated; with nonzero values they will be wrapped. If you want more information than is provided in this file, examine the system sources themselves, for example, using *grep* to locate the relevant files. In any event, be sure *MACHINE* is set to *IBM_PC* (or one of the 68000 types if you have one). If you have an 80386-based processor, use *IBM_PC*, not *IBM_386*, as that is intended for a future 32-bit version of MINIX, and will not work at present. The current 16-bit version works fine on 80386s, but initializes all segment descriptors to 16-bit mode.

The kernel directory contains a shell script *config*. Before starting to compile the system, examine this file using your favorite editor. You will see that it begins with a **case** statement that switches on the first argument. Each of the clauses defines some variables that are used later. The idea here is that you need files called *mpx.x*, *klib.x*, and *wini.c*. For each of these there are several candidates. Which one you use depends on the your system configuration.

If you have a PC/AT with a PC/AT hard disk controller, type:

```
config at
```

to set up the files. On the other, if you have a PC/XT (8088), use *xt* instead of *at* as the argument. For a PS/2, use *ps*. If none of these produce working systems, run *config* again using *bios* as the argument this time. If you happen to have a PC with a PC/AT disk controller or a PC/AT with an XT disk controller, you will have to build the configuration by hand.

7.1.2. Compiling the Pieces

Once everything has been set up, actually compiling the pieces is easy. First go to the *kernel* directory on you hard disk (or mount the *kernel* diskette and go to it). Then type:

```
make –n
```

to see what *make* is planning to do. Normally it will list a sequence of compilations to be done. If it complains that it cannot find some file, please install the file.

Now do it for real by typing:

```
make
```

The kernel will be compiled. On a 33 MHz 80386 with a fast hard disk, it will take under 3 minutes. On a 4.77 MHz 8088 with two diskette drives it will take rather longer. When it is finished, you will be left with a collection of *.s* files, all of which can now be removed if space is tight, and a file *kernel*, which will be needed.

If you have a small system, it is possible that there will not be enough room for *make* and the C compiler simultaneously. In that case type:

```
make -n >script
sh script
```

If even that fails due to lack of memory, examine *script* and type in all the commands by hand, one at a time.

Now go to *fs*. If you are using diskettes, first unmount the one containing the kernel sources and mount the one containing the file system sources. Now type

```
make -n
```

to see if everything is all right, followed by:

```
make
```

to do the work. Again here, the *.s* can be removed, but the file *fs* must be kept. In a similar way, go to *mm* and use *make* to produce the *mm* file.

Finally, go to *tools* and type:

```
make
```

to produce *init*, *bootblok*, *build*, and *menu*. (Actually a binary version of *bootblok* is provided since it is so short, but making a new one does not take very long.) Check to see that all of them have been made. If one is missing, use *make* to produce it, for example:

```
make init
```

7.1.3. Building the Boot Diskette

In this section we will describe how the six independently compiled and linked programs, *kernel*, *fs*, *mm*, *init*, *bootblok*, and *menu* are forged together to make the boot diskette using *build*.

The boot diskette contains the six programs mentioned above, in the order given. The boot block occupies the first 512 bytes on the disk. When the computer is turned on, the ROM gets control and tries to read the boot block from drive 0 into memory (at address 0x7C00 on the IBM PC). If this read succeeds, the ROM jumps to the boot block to begin the load.

The MINIX boot program first copies itself to an address just below 256K, to get itself out of the way. Then it calls the BIOS repeatedly to load cylinders of data into low core. This data is the bootable image of the operating system, followed directly by *menu* (on the IBM PC). When the loading is finished, the boot program jumps to the start of *menu*, which then displays the initial menu. If the user types an equal sign, *menu* jumps to an address in low core to start MINIX.

The boot diskette is generated by *tools/build*. It takes the six programs listed

above and concatenates them in a special way. The first 512 bytes of the boot diskette come from *bootblok*. If need be, some zero bytes are added to pad *bootblok* out to 512. *Bootblok* does not have a header, and neither does the boot diskette because when the ROM loads the boot block to address 0x7C00, it expects the first byte to be the start of the first instruction.

At position 512, the boot diskette contains the kernel, again without a header. Byte 512 of the boot diskette will be placed at memory address 1536 by the boot program, and will be executed as the first MINIX instruction when *menu* terminates. After the kernel comes *mm*, *fs*, *init*, and *menu*, each padded out to a multiple of 256 bytes so that the next one begins at a click boundary.

Each of the programs may be compiled either with or without separate I and D space (on the IBM PC; the 68000 versions do not have this feature). The two models are different, but *build* explicitly checks to see which model each program uses and handles it. In short, what *build* does is read six files, stripping the headers off the last five of them, and concatenate them onto the output, rounding the first one up to 512 bytes and the rest up to a multiple of 16 bytes.

After concatenating the six files, *build* makes three patches to the output.

1. The last 4 words of the boot block are set to the number of cylinders to load, and the DS, PC, and CS values to use for running *menu*. The boot program needs this information so that it can jump to *menu* after it has finished loading. Without this information, the boot program would not know where to jump.

2. *Build* loads the first 8 words of the kernel's data segment with the CS and DS segment register values for *kernel*, *mm*, *fs*, and *init*. Without this information, the kernel could not run these programs when the time came: it would not know where they were. It also sets word 4 of the kernel's text segment to the DS value needed to run the kernel.

3. The origin and size of *init* are inserted at address 4 of the file system's data space. The file system needs this information to know where to put the RAM disk, which begins just after the end of *init*, exactly overwriting the start of *menu*.

To have *build* actually construct the new boot diskette, insert a blank, formatted diskette in drive 0 and type:

```
make image
```

It will run, build the boot diskette, and display the sizes of the pieces on the screen. When it is finished, kill any background processes, do a *sync*, and reboot the system. After logging in, go the *test* directory and type:

```
run
```

to run all the test programs, assuming they have already been compiled. If they have not been, log in as root and type:

```
make all
```

If you do not have a hard disk, the above procedure has to be modified slightly. You will have to copy the *kernel*, *fs*, and *mm* files to the *tools* directory and change *Makefile* accordingly.

7.1.4. Installing New Device Drivers

Once you have successfully reached this point, you will now be able to modify MINIX. In general, if a modification only affects, say, the file system, you will not have to recompile the memory manager or kernel. If a modification affects any of the files in */usr/include* you should recompile the entire system, just to be safe.

It is conceivable that your modification has increased the size of some file so much that the compiler complains about it. If this occurs, try to determine which pass it is using the **–v** flag to *cc*, and then give that pass more memory using the *chmem* program.

One common modification is adding new I/O devices and drivers. To add a new I/O device to MINIX, it is necessary to write a driver for it. The new driver should use the same message interface as the existing ones. The driver should be put in the directory *kernel* and *Makefile* updated. In addition, the entry point of the new task must be added to the list contained in the array *task* in *kernel/table.c*.

Two changes are also required in */usr/include/minix*. In *const.h*, the constant *NR_TASKS* has to be increased by 1, and the new task has to be given a name in *com.h*.

A new special file will have to be created for the driver using *mknod*.

To tell the file system which task is handling the new special file, a line has to be added to the array *dmap* in *fs/table.c*.

7.1.5. Troubleshooting

If you modify the system, there is always the possibility that you will introduce an error. In this section, we will discuss some of the more common problems and how to track them down.

To start with, if something is acting strange, turn the computer off, wait about one minute, and reboot from scratch. This gets everything into a known state. Rebooting with CTRL-ALT-DEL may leave the system in a peculiar state, which may be the cause of the trouble.

If a message like

```
Booting MINIX 1.5
```

does not appear on the screen after the power-on self-tests have completed (on the

IBM PC), something is wrong with the boot block. The boot block prints this message by calling the BIOS. Make a dump of the first block of the boot diskette and examine it by hand to see if it contains the proper program.

If the above message appears, but the initial menu does not, it is likely that *menu* is not being started, since the first thing *menu* does is print the menu. Check the last 6 bytes of the boot block to see if the segment and offset put there by *build* correspond to the address at which *menu* is located (right after *init*).

If the menu appears, but the system does not respond to the equal sign, MINIX is probably being started, but crashing during initialization. One possible cause is the introduction of print statements into the kernel. However, it is not permitted to display anything until after the terminal task has run to initialize itself. Be careful about where you put the print statements.

If the screen has been cleared and the message giving the sizes has appeared, the kernel has initialized itself, the memory manager has run and blocked waiting for a message, and the file system has started running. This message is printed as soon as the file system has read the super-block of the root file system.

If the system appears to hang before or after reading the root file system, some help can be obtained by hitting the F1 or F2 function keys (unless the dump routines have been removed). By hitting F1 twice a few seconds apart and noting the times in the display, it may be possible to see which processes are running. If, for example, *init* is unable to fork, for whatever reason, or cannot open */etc/ttys*, or cannot execute */bin/sh* or */bin/login*, the system will hang, but process 2 (*init*) may continue to use CPU cycles. If the F1 display shows that process 2 is constantly running, it is a good bet that *init* is unable to make a system call or open a file that is essential. The problem can usually be localized by putting statements in the main loops of the file system and memory manager to print a line describing each incoming message and each outgoing reply. Recompile and test the system with the new output.

7.2. REBUILDING MINIX ON THE ATARI ST

It is possible to rebuild MINIX-ST on any system with at least 1 MB of memory and a 720K disk drive. However such a configuration is the bare minimum. Additional hardware greatly speeds up the process.

7.2.1. Configuring the System

In order to rebuild MINIX-ST you must first prepare your system. What you must do depends on your system. If you have a hard disk, you should install all the sources and binaries on your disk. Chapter 3 describes how to achieve this.

If you do not have a hard disk, you should create 4 720K disks. These disks should contain the unpacked mm, fs, kernel and tools sources respectively. Chapter 3 describes how to unpack the sources.

If you want to reconfigure the system you should examine the files *include/minix/config.h* and *include/minix/boot.h*. These files are found on 06.ACK, and contain a number of tunable system parameters. For instance if you keep your root partition on */dev/hd3*, but you do not want to load this partition into the RAM disk upon startup, you could change the line

```
#define DROOTDEV  (DEV_RAM + 0)
```

in *include/minix/boot.h* into

```
#define DROOTDEV  (DEV_HD0 + 3)
```

If you do not want to copy the root partition, but you want to keep a RAM disk, you should modify the value of the constant *DRAMSIZE* in *include/minix/boot.h* as well.

If you have a system with a United Kingdom or German keyboard, it is recommended to go to the directory with the kernel sources, and substitute in the file *Makefile*, the string *us* in the line:

```
KEYMAP = keymap.us.h
```

by *uk* or *ge* respectively. If you do this you will generate MINIX for use with your native keyboard instead of a US one. By doing so, you do not need to run *fixkeys* on your boot disk any more.

If you have a system with a real time clock on the disk controller it is recommended to go to the directory with the kernel sources, and modify the first few lines of the file *Makefile* so that they read:

```
CLOCKS = -DCLOCKS
#CLOCKS =
```

7.2.2. Rebuilding MINIX Using a Hard Disk

Rebuilding MINIX is fairly simple when you have a hard disk. Assuming that you have installed the sources in */usr/src*, and that there is enough free space on your hard disk to accommodate all object files and results, type:

```
chmem =110000 /usr/lib/cem
cd /usr/src/mm
make
cd /usr/src/fs
make
cd /usr/src/kernel
make
cd /usr/src/tools
make
```

If disk space is tight you can remove all *.o* files after each make. If everything

succeeds, you will have a file called *minix.img* in */usr/src/tools*. You can either write this file to TOS using the *toswrite* command, or create a new boot diskette by inserting an empty, formatted disk into the disk drive and issuing the command:

```
cp /usr/src/tools/minix.img /dev/fd0
```

Now you can logout and reboot the system to try your new boot disk. If required run the TOS program *fixkeys* to modify the keyboard tables to reflect your hardware. It is advised to generate a new file */etc/psdatabase*, which is used by the *ps* program. The command:

```
ps –U
```

will make this file for you. Do not forget to copy */etc/psdatabase* to your root disk!

Refer to Sec. 3.12 if your new boot disk does not function properly.

7.2.3. Rebuilding MINIX Using 1 MB or Two 720K Disk Drives

If your have more than 1 MB of memory, your should create a huge RAM disk. The size of the RAM disk is not critical. A RAM disk of 1 MB will do, but more does not harm you. In addition to the usual contents of the RAM disk, you should also copy disk 06.ACK onto the RAM disk. Take care that the various compiler passes are found in */usr/lib* or */lib*.

If you have two disk drives you should use one drive to hold the 06.ACK disk. This disk should be mounted on */usr*. The other drive will be used to hold the disks with the sources. You will also need a RAM disk which has at least 150 KB free.

In both cases after setting up, execute the following steps:

```
cd /
chmem =110000 /usr/lib/cem
```

Insert 03.USR1 into the disk drive and type:

```
mount /dev/dd0 /user
cp /user/bin/dd /bin/dd
cp /user/bin/make /bin/make
umount /dev/dd0
```

Insert the disk with the mm sources into the disk drive and type:

```
mount /dev/dd0 /user
cd /user/src/mm
make
cp mm.mix /tmp/mm.mix
cd /
umount /dev/dd0
```

Insert the disk with the tools sources into the disk drive and type:

```
mount /dev/dd0 /user
mkdir /user/src/mm
cp /tmp/mm.mix /user/src/mm/mm.mix
rm /tmp/mm.mix
umount /dev/dd0
```

Insert the disk with the fs sources into the disk drive and type:

```
mount /dev/dd0 /user
cd /user/src/fs
make
cp fs.mix /tmp/fs.mix
cd /
umount /dev/dd0
```

Insert the disk with the tools sources into the disk drive and type:

```
mount /dev/dd0 /user
mkdir /user/src/fs
cp /tmp/fs.mix /user/src/fs/fs.mix
rm /tmp/fs.mix
umount /dev/dd0
```

Insert the disk with the kernel sources into the disk drive and type:

```
mount /dev/dd0 /user
cd /user/src/kernel
make
cp kernel.mix /tmp/kernel.mix
cd /
umount /dev/dd0
```

Insert the disk with the tools sources into the disk drive and type:

```
mount /dev/dd0 /user
mkdir /user/src/kernel
cp /tmp/kernel.mix /user/src/kernel/kernel.mix
rm /tmp/kernel.mix
cd /user/src/tools
make
cp minix.img /tmp/minix.img
cd /
umount /dev/dd0
```

If everything succeeds, you will have a file called *minix.img* in */tmp*. You can either write this file to TOS using the *toswrite* command, or create a new boot

diskette by inserting a blank, formatted diskette into the disk drive and then typing:

```
cp /tmp/minix.img /dev/fd0
```

Now you can log out and reboot the system to try your new boot disk. If required run the TOS program *fixkeys* to modify the keyboard tables to reflect your hardware. It is advised to generate a new file */etc/psdatabase*, which is used by the *ps* program. The command:

```
ps –U
```

will make this file for you. Do not forget to copy */etc/psdatabase* to your root disk!

Refer to section 3.12 if your new boot disk does not function properly.

7.2.4. Rebuilding MINIX Using 1 MB and a 720K Disk Drive

Rebuilding MINIX with only one 720K disk drive and 1 MB of memory is somewhat more complicated. Therefore it is highly recommended to study this subsection completely before even attempting to rebuild MINIX. First you have to prepare a compiler disk. This is done by making a copy of 06.ACK. Remove all but the following files from your newly created compiler disk: *bin/as*, *bin/cc*, *lib/cem*, *lib/cg*, *lib/crtso.o*, *lib/cv*, *lib/end.o*, *lib/head.o*, *lib/ld*, *lib/libc.a*, *lib/opt*, *include/**, (all files in *include* and its subdirectories) Now mount the USR1 disk and copy the following programs to */tmp*: *make*, *mined*, *dd*, *cpdir*. Then mount your compiler disk, and copy these programs onto the bin directory of the compiler disk. After doing so you should remove them from */tmp*.

Make a set of source disks as specified in the previous subsection. Reboot the system with a root disk which contains a 400 KB RAM disk. Log in as root. Unmount the usr filesystem, and mount your compiler disk on /usr.

Now we are ready to start the compilation process. By and large, the next steps are similar to the one from the previous subsection. However, since you have only one drive, which holds the compiler disk, the sources are going to be kept in the RAM disk. During the remainder of this subsection we will assume that your sources are kept in */tmp/src*.

Whenever it is stated that you should insert a disk with sources you should unmount your compiler disk. Mount the disk which contained the sources on which you were working. Then copy the contents of */usr/src* back to the disk where the sources came from. This is most easily done through the command:

```
cpdir –msv /tmp/src /usr/src
```

Now erase your source directory by issuing the command:

```
cd /tmp/src; rm –rf *
```

Unmount your old source disk and mount the new one. Copy the sources to the RAM disk by typing:

```
cpdir -msv /usr/src /tmp/src
```

Whenever the steps tell you to issue the command *make*, you should type:

```
make -n >script
```

followed by the command:

```
sh -v script
```

Now the sources are being compiled. This can take a substantial amount of time. It is possible that during the compilation process your RAM disk runs out of space. This is reported by the message:

```
No space left on device 1/0
```

If that happens, you should delete all source files with extension *.c* that are already compiled. Do NOT remove files with a *.h* or *.o* extension or files that are not yet compiled. Modify the file script using *mined*. Remove all lines preceding the line on which your RAM disk ran out of space. Do not remove the line on which the error occurred, since that file is not yet completely processed. After modifying the file script, restart the compilation process by re-issuing the command:

```
sh -v script
```

Notice again that all sources which are compiled reside on the RAM disk in the directory */tmp/src*. Whenever issuing commands like *make* and *rm*, be sure that you are indeed on the RAM disk, and that you are not accidently cluttering up your compiler disk or one of your source disks.

7.2.5. Installing New Device Drivers

Once you have successfully reached this point, you will now be able to modify MINIX. In general, if a modification only affects, say, the file system, you will not have to recompile the memory manager or kernel. If a modification affects any of the files in */usr/include* you should recompile the entire system, just to be safe.

It is conceivable that your modification has increased the size of some file so much that the compiler complains about it. If this occurs, try to determine which pass it is using the **-v** flag to *cc*, and then give that pass more memory using *chmem*.

One common modification is adding new I/O devices and drivers. To add a new I/O device to MINIX, it is necessary to write a driver for it. The new driver should use the same message interface as the existing ones. The driver should be put in the directory *kernel* and *Makefile* should be updated. In addition, the entry point of the new task must be added to the list contained in the array *task* in *kernel/table.c*.

Two changes are also required in the */usr/include/minix* directory. In *const.h*, the constant *NR_TASKS* has to be increased by 1, and the new task has to be given a name in *com.h*.

A new special file will have to be created for the driver. This can be done with *mknod*.

To tell the file system which task is handling the new special file, a line has to be added to the array *dmap* in *fs/table.c*.

7.2.6. Recompiling Commands and Libraries

The procedure for recompiling the commands and the library is similar to the one for recompiling the kernel.

A major difference between recompiling commands and recompiling the kernel is that each command (and each library module) can be recompiled independently of all the others, so that less RAM disk is needed.

In order to run *make* in the commands directory you should give *make* 35000 bytes of memory by issuing the command:

```
chmem =35000 /usr/bin/make
```

A few command source files are so big that the compiler complains about it. If this occurs, try to determine which pass it is using the *–v* flag to *cc*, and then give that pass more memory using *chmem*.

Should the compiler run out of temporary space during a compilation you can either use a larger RAM disk, or you can tell the compiler to put its temporary files in another directory (on disk). Add –T*dir* to the compile command if you want to create the temporary files in directory *dir*.

7.3. REBUILDING MINIX ON THE COMMODORE AMIGA

To rebuild MINIX on the Amiga, you need at least 1M of memory. The procedure is the same as for a 1M Atari, as described earlier in this chapter. The only difference is that instead of copying the *minix.img* file to */dev/fd0* you have to transfer *minix.img* to an AmigaDOS floppy, using *transfer*. The exact details are given in the manual page of *transfer* in chapter 8.

7.4. REBUILDING MINIX ON A MACINTOSH

This section describes the procedure for building the boot application and the kernel programs for the Macintosh version of MINIX. Before continuing, see section 7.1 for a description of the source directories.

If you are working on a hard disk, be sure that all these directories have been set up, and all files copied there from the distribution diskettes and decompressed and dearchived.

If you do not have a hard disk, there are a couple of ways you can recompile the

system. First, if you have two diskette drives, use one drive to hold the root file system, including the compiler, */usr/lib* and */usr/include*. Diskettes with programs to be compiled are mounted on the other drive.

Second, if you have enough memory for a sufficiently large RAM disk, you can put the root file system there, along with the compiler, */usr/lib* and */usr/include*.

If you a system with only one diskette drive, no hard disk, and insufficient memory for a large RAM disk, it is probably not possible to recompile the system.

As a test to see if everything is set up properly, type in, compile, and run the following program:

```
#include <limits.h>
main()
{
   printf("PATH_MAX = %d\n", PATH_MAX);
}
```

It should print out the value 255 for *PATH_MAX*.

7.4.1. Configuring the System

The file */usr/include/minix/config.h* contains some user-settable parameters. Examine this file and make any changes that you require. For example, if *LINEWRAP* is set to 0, then lines longer than 80 characters will be truncated; with nonzero values they will be wrapped. If you want more information than is provided in this file, examine the system sources themselves, for example, using *grep* to locate the relevant files. In any event, be sure *MACHINE* is set to *MACINTOSH*.

7.4.2. Compiling the Pieces

Once everything has been set up, actually compiling the pieces is easy. First go to the *kernel* directory on you hard disk (or mount the *kernel* diskette and go to it). Then type:

```
make –n
```

to see what *make* is planning to do. Normally it will list a sequence of compilations to be done. If it complains that it cannot find some file, please install the file.

Now do it for real by typing:

```
make
```

The kernel will be compiled.

Now go to *fs*. If you are using diskettes, first unmount the one containing the kernel sources and mount the one containing the file system sources. Now type

```
make –n
```

to see if everything is all right, followed by

```
make
```

to do the work. In a similar way, go to *mm* and use *make* to produce the *mm* file.

Finally, go to *tools* and type

```
make
```

to produce *init*. Check to see that all of them have been made. If one is missing, use *make* to produce it.

7.4.3. The Boot Sequence

In this section we will describe how the four independently compiled and linked programs, *kernel*, *fs*, *mm*, and *init*, are used in conjunction with the boot application to boot MINIX on the Macintosh.

Basically, the boot application does the following:

1. It requests some memory from the the Macintosh operating system. This memory will be used to load the MINIX kernel programs. Anything remaining after these are loaded will be used by the MINIX kernel to run MINIX programs.
2. The *kernel* program is loaded first. The boot application reads this program from the *resource* fork (Macintosh resources are explained below) of the boot application, loads it into memory and relocates it so that the addresses that the kernel use correspond correctly to the place where it has been loaded in memory.
3. Similarly, *mm* is loaded, followed by *fs* and *init*. As each program is loaded, the boot application reports where in memory it has been loaded and how much memory has been consumed (text and data are shown separately, and each is padded to a multiple of 256 bytes).

After having loaded the four files, the boot application jumps to the first instruction of the kernel, where execution proceeds normally. Since the kernel needs to know where each program (mm, fs, and init) has been loaded, the boot application passes this information on the stack.

7.4.4. The Boot Application

The boot application is a relatively small program that is executed by the Macintosh operating system. Every application that is executable by the Macintosh operating system is composed of a number of *resources*. Each of these resources describes some aspect of the application. For instance, CODE resources are

compiled source code, MENU resources describe menu bars, ICON resources describe the icon of the program when it is displayed on the desktop, and so on. The Macintosh operating system contains many system calls to support the use and manipulation of resources. There are many, many different types of resources. The idea behind all of this was that the executable code of the application could be divorced from the user interface aspects, and the application could be easily customized for different countries and languages.

The boot application, then, consists of three categories of resources: the code for the boot application itself (CODE resources), a resource for each of the kernel programs (BOOT resources), and other peripheral resources. Included in this latter category are things like the picture that is displayed when you select the "About MINIX" menu item (the PICT resource). Note that the structure of resource files is not even slightly related to the structure of a normal MINIX executable, and they cannot be executed by the MINIX operating system.

7.4.5. Building and Testing a New Boot Application

Once you understand resources, the process of building the boot application becomes rather straight forward. First the boot code itself is compiled, then each of the kernel programs are compiled, and then a utility program called *rmaker* composes the actual boot application from a textual description of the resources. *Rmaker* is called a resource compiler; it is a very simple minded one and only knows how to build a resource file from a limited number of resource types, but it should be sufficient for most needs.

To build a new boot application, make a copy of the BOOT.00 diskette and set it aside. Now boot MINIX, make the new kernel programs if you have not already done so, go to the tools directory and type:

```
make boot
```

This will compile the code of boot program (if necessary), and then it will run the *rmaker* utility. The rmaker utility reads the resource descriptions from *boot.r* and builds the new boot application on the diskette (replacing the old one if necessary, so only use a COPY of BOOT.00). When the make is finished, kill any background processes, do a *sync*, and reboot the system with the new diskette. After logging in, go to the *test* directory and type:

```
run
```

to run all the test programs, assuming they have already been compiled. If they have not been, log in as root and type:

```
make all
```

If you do not have a hard disk, the above procedure has to be modified slightly.

You will have to copy the *kernel*, *fs*, and *mm* files to the *tools* directory and change *boot.r* to reflect the change.

7.4.6. Installing New Device Drivers

Follow the procedure outlined in the IBM PC section.

7.4.7. Troubleshooting

If you modify the system, there is always the possibility that you will introduce an error. In this section, we will discuss some of the more common problems and how to track them down.

To start with, if something is acting strange, turn the computer off, wait about one minute, and reboot from scratch. This gets everything into a known state. Rebooting with CTRL-ALT-DEL may leave the system in a peculiar state, which may be the cause of the trouble.

If a message like

```
Booting MINIX 1.5
```

does not appear on the screen after the power-on self-tests have completed (on the IBM PC), something is wrong with the boot block. The boot block prints this message by calling the BIOS. Make a dump of the first block of the boot diskette and examine it by hand to see if it contains the proper program.

If the above message appears, but the initial menu does not, it is likely that *menu* is not being started, since the first thing *menu* does is print the menu. Check the last 6 bytes of the boot block to see if the segment and offset put there by *build* correspond to the address at which *menu* is located (right after *init*).

If the menu appears, but the system does not respond to the equal sign, MINIX is probably being started, but crashing during initialization. One possible cause is the introduction of print statements into the kernel. However, it is not permitted to display anything until after the terminal task has run to initialize itself. Be careful about where you put the print statements.

If the screen has been cleared and the message giving the sizes has appeared, the kernel has initialized itself, the memory manager has run and blocked waiting for a message, and the file system has started running. This message is printed as soon as the file system has read the super-block of the root file system.

If the system appears to hang before or after reading the root file system, some help can be obtained by hitting the F1 or F2 function keys (unless the dump routines have been removed). By hitting F1 twice a few seconds apart and noting the times in the display, it may be possible to see which processes are running. If, for example, *init* is unable to fork, for whatever reason, or cannot open */etc/ttys*, or cannot execute */bin/sh* or */bin/login*, the system will hang, but process 2 (*init*) may continue to use CPU cycles. If the F1 display shows that process 2 is constantly running, it is

a good bet that *init* is unable to make a system call or open a file that is essential. The problem can usually be localized by putting statements in the main loops of the file system and memory manager to print a line describing each incoming message and each outgoing reply. Recompile and test the system using the new output as a guide.

9

EXTENDED MANUAL PAGES

This chapter contains extended manual pages for selected programs. These programs are sufficiently complex that it was thought wise to produce these documents. Not every program is present on all versions of MINIX. When a program is only available on some versions, the names of these versions are given in the section heading in square brackets.

9.1. ASLD—ASSEMBLER-LOADER [IBM]

The assembly language expected by the MINIX assembler for the 8088, *asld*, is identical to that of PC-IX, the version of UNIX IBM originally supported on the PC.

9.1.1. Tokens, Numbers, Character Constants, and Strings

The syntax of numbers is the same as in C. The constants 32, 040, and 0x20 all represent the same number, but are written in decimal, octal, and hex, respectively. The rules for character constants and strings are also the same as in C. For example, 'a' is a character constant. A typical string is "string".

9.1.2. Symbols

Symbols contain letters and digits, as well as three special characters: dot, tilde, and underscore. The first character may not be a digit or tilde. Only the first 8 characters are significant. Thus "hippopotamus" and "hippopotapig" cannot both be defined as external symbols.

The names of the 8088 registers are reserved. These are:

al, bl, cl, dl
ah, bh, ch, dh
ax, bx, cx, dx
si, di, bp, sp
cs, ds, ss, es
bx_si, bx_di, bp_si, bp_di

The last group of 4 are used for base + index mode addressing, in which two registers are added to form the effective address.

Names of instructions and pseudo-ops are not reserved. Alphabetic characters in opcodes and pseudo-ops must be in lower case.

9.1.3. Separators

Commas, blanks, and tabs are separators and can be interspersed freely between tokens, but not within tokens. Commas are only legal between operands.

9.1.4. Comments

The comment character is "|". The rest of the line is ignored.

9.1.5. Opcodes

The opcodes are listed below. Notes: (1) Different names for the same instruction are separated by "/". (2) Square brackets ([]) indicate that 0 or 1 of the enclosed characters can be included. (3) Curly brackets ({}) work similarly, except that one of the enclosed characters *must* be included. Thus square brackets indicate an option, whereas curly brackets indicate that a choice must be made.

Data Transfer

```
mov[b]     dest, source    | Move word/byte
mov{bw}    dest, source    | Move word/byte from source to dest
pop        dest            | Pop stack
push       source          | Push stack
xchg       op1, op2        | Exchange word/byte
xlat                       | Translate
```

Input/Output

in[w]	source	| Input from source I/O port
in[w]		| Input from DX I/O port
out[w]	dest	| Output to dest I/O port
out[w]		| Output to DX I/O port

Address Object

lds	reg,source	| Load reg and DS from source
les	reg,source	| Load reg and ES from source
lea	reg,source	| Load effect address of source to reg and DS
seg	reg	| Specify seg register for next instruction

Flag Transfer

lahf		| Load AH from flag register
popf		| Pop flags
pushf		| Push flags
sahf		| Store AH in flag register

Addition

aaa		| Adjust result of BCD addition
add[b]	dest,source	| Add
adc[b]	dest,source	| Add with carry
daa		| Decimal Adjust acc after addition
inc[b]	dest	| Increment by 1

Subtraction

aas		| Adjust result of BCD subtraction
sub[b]	dest,source	| Subtract
sbb[b]	dest,source	| Subtract with borrow from dest
das		| Decimal adjust after subtraction
dec[b]	dest	| Decrement by one
neg[b]	dest	| Negate
cmp[b]	dest,source	| Compare
cmp{bw}	dest,source	| Compare

Multiplication

aam		| Adjust result of BCD multiply
imul[b]	source	| Signed multiply
mul[b]	source	| Unsigned multiply

Division

```
    aad                                   | Adjust AX for BCD division
    cbw                                   | Sign extend AL into AH
    cwb                                   | Sign extend AX into DX
    idiv[b]        source                 | Signed divide
    div[b]         source                 | Unsigned divide
```

Logical

```
    and[b]         dest,source            | Logical and
    not[b]         dest                   | Logical not
    or[b]          dest,source            | Logical inclusive or
    test[b]        dest,source            | Logical test
    xor[b]         dest,source            | Logical exclusive or
```

Shift

```
    sal[b]/shl[b]  dest,CL                | Shift logical left
    sar{b}         dest,CL                | Shift arithmetic right
    shr[b]         dest,CL                | Shift logical right
```

Rotate

```
    rcl[b]         dest,CL                | Rotate left, with carry
    rcr[b]         dest,CL                | Rotate right, with carry
    rol[b]         dest,CL                | Rotate left
    ror[b]         dest,CL                | Rotate right
```

String Manipulation

```
    cmp[b]                                | Compare
    cmp{bw}                               | Compare
    lod{bw}                               | Load into AL or AX
    mov[b]                                | Move
    mov{bw}                               | Move
    rep                                   | Repeat next instruction until CX=0
    repe/repz                             | Repeat next instruction until CX=0 and ZF
    repne/repnz                           | Repeat next instruction until CX!=0 and ZI
    sca{bw}                               | Compare string element ds:di with AL/AX
    sto{bw}                               | Store AL/AX in ds:di
```

Control Transfer

Displacement is indicated by opcode; "jmp" generates a 16-bit displacement, and "j" generates 8 bits only. The provision for "far" labels is described below.

Asld accepts a number of special branch opcodes, all of which begin with "b". These are meant to overcome the range limitations of the conditional branches, which can only reach to targets within −126 to +129 bytes of the branch ("near"

labels). The special "b" instructions allow the target to be anywhere in the 64K-byte address space. If the target is close enough, a simple conditional branch is used. Otherwise, the assembler automatically changes the instruction into a conditional branch around a "jmp".

The English translation of the opcodes should be obvious, with the possible exception of the unsigned operations, where "lo" means "lower," "hi" means "higher," and "s" means "or same".

The "call", "jmp", and "ret" instructions can be either intrasegment or intersegment. The intersegment versions are indicated with the suffix "i".

Unconditional

```
br        dest      | jump, 16-bit displacement, to dest
j         dest      | jump, 8-bit displacement, to dest
call[i]   dest      | call procedure
jmp[i]    dest      | jump, 16-bit displacement, to dest
ret[i]              | return from procedure
```

Conditional with 16-bit Displacement

```
beq       | branch if equal
bge       | branch if greater or equal (signed)
bgt       | branch if greater (signed)
bho       | branch if higher (unsigned)
bhis      | branch if higher or same (unsigned)
ble       | branch if less or equal (signed)
blt       | branch if less (signed)
blo       | branch if lower (unsigned)
blos      | branch if lower or same (unsigned)
bne       | branch if not equal
```

Conditional with 8-bit Displacement

```
ja/jnbe        | if above/not below or equal (unsigned)
jae/jnb/jnc    | if above or equal/not below/not carry (unsigned
jb/jnae/jc     | if not above nor equal/below/carry (unsigned)
jbe/jna        | if below or equal/not above (unsigned)
jg/jnle        | if greater/not less nor equal (signed)
jge/jnl        | if greater or equal/not less (signed)
jl/jnqe        | if less/not greater nor equal (signed)
jle/jgl        | if less or equal/not greater (signed)
je/jz          | if equal/zero
jne/jnz        | if not equal/not zero
jno            | if overflow not set
jo             | if overflow set
jnp/jpo        | if parity not set/parity odd
```

```
    jp/jpe                      | if parity set/parity even
    jns                         | if sign not set
    js                          | if sign set
```

Iteration Control

```
    jcxz          dest          | jump if CX = 0
    loop          dest          | Decrement CX and jump if CX != 0
    loope/loopz   dest          | Decrement CX and jump if CX = 0 and ZF
    loopne/loopnz               dest| Decrement CX and jump if CX != 0 and
```

Interrupt

```
    int                         | Software interrupt
    into                        | Interrupt if overflow set
    iret                        | Return from interrupt
```

Flag Operations

```
    clc                         | Clear carry flag
    cld                         | Clear direction flag
    cli                         | Clear interrupt enable flag
    cmc                         | Complement carry flag
    stc                         | Set carry flag
    std                         | Set direction flag
    sti                         | Set interrupt enable flag
```

External Synchronization

```
    esc source                  | Put contents of source on data bus
    hlt                         | Halt until interrupt or reset
    lock                        | Lock bus during next instruction
    wait                        | Wait while TEST line not active
```

9.1.6. Location Counter

The special symbol "." is the location counter and its value is the address of the first byte of the instruction in which the symbol appears and can be used in expressions.

9.1.7. Segments

There are three different segments: text, data and bss. The current segment is selected using the *.text*, *.data* or *.bss* pseudo-ops. Note that the "." symbol refers to the location in the current segment.

9.1.8. Labels

There are two types: name and numeric. Name labels consist of a name followed by a colon (:).

Numeric labels consist of one or more digits followed by a dollar ($). Numeric labels are useful because their definition disappears as soon as a name label is encountered; thus numeric labels can be reused as temporary local labels.

9.1.9. Statement Syntax

Each line consists of a single statement. Blank or comment lines are allowed.

9.1.10. Instruction Statements

The most general form of an instruction is

```
label: opcode operand1, operand2    | comment
```

9.1.11. Expression Semantics

The following operator can be used: + – * / & ! < (shift left) > (shift right) – (unary minus). Sixteen-bit integer arithmetic is used. Division produces a truncated quotient.

9.1.12. Addressing Modes

Below is a list of the addressing modes supported. Each one is followed by an example.

8-bit constant	mov ax, *2
16-bit constant	mov ax, #12345
direct access (16 bits)	mov ax, counter
register	mov ax, si
index	mov ax, (si)
index + 8-bit disp.	mov ax, *-6(bp)
index + 16-bit disp.	mov ax, #400(bp)
base + index	movax, (bp_si)
base + index + 8-bit disp.	mov ax, *14(bp_si)
base + index + 16-bit disp.	mov ax, #-1000(bp_si)

Any of the constants or symbols may be replacement by expressions. Direct access, 16-bit constants and displacements may be any type of expression. However, 8-bit constants and displacements must be absolute expressions.

9.1.13. Call and Jmp

With the "call" and "jmp" instructions, the operand syntax shows whether the call or jump is direct or indirect; indirection is indicated with an "@" before the operand.

```
call _routine          | Direct, intrasegment
call @subloc           | Indirect, intrasegment
call @6(bp)            | Indirect, intrasegment
call (bx)              | Direct, intrasegment
call @(bx)             | Indirect, intrasegment
calli @subloc          | Indirect, intersegment
calli cseg, offs       | Direct, intersegment
```

Note that call (bx) is considered direct, though the register is not called, but rather the location whose address is in the register. With the indirect version, the register points to a location which contains the location of the routine being called.

9.1.14. Symbol Assigment

Symbols can acquire values in one of two ways. Using a symbol as a label sets it to "." for the current segment with type relocatable. Alternative, a symbol may be given a name via an assignment of the form

```
symbol = expression
```

in which the symbol is assigned the value and type of its arguments.

9.1.15. Storage Allocation

Space can be reserved for bytes, words, and longs using pseudo-ops. They take one or more operands, and for each generate a value whose size is a byte, word (2 bytes) or long (4 bytes). For example:

```
.byte 2, 6             | allocate 2 bytes initialized to 2 and 6
.word 3, 0x10          | allocate 2 words initialized to 3 and
16
.long 010              | allocate a long initialized to 8
.zerow 20              | allocates 20 words of zeroes
```

allocates 10 (decimal) bytes of storage, initializing the first two bytes to 2 and 6, the next two words to 3 and 16, and the last 4 bytes to a long with value 8 (010 octal).

9.1.16. String Allocation

The pseduo-ops *.ascii* and *.asciz* take one string argument and generate the ASCII character codes for the letters in the string. The latter automatically terminates the string with a null (0) byte. For example,

```
.ascii "hello"
.asciz "world\n"
```

9.1.17. Alignment

Sometimes it is necessary to force the next item to begin at an even address. The *.even* pseudo-op generates a null byte if the current location is odd, and nothing if it is even.

9.1.18. Segment Control

Every item assembled goes in one of the three segments: text, data, or bss. By using the *.text*, *.data* and *.bss* pseudo-ops, the programmer can force the next items to go in a particular segment.

9.1.19. External Names

A symbol can be given global scope by including it in a *.globl* pseudo-op. Multiple names may be listed, separate by commas. It can be used for both exporting symbols defined in the current program, or importing names defined outside.

9.1.20. Common

The *.comm* pseudo-op declares storage that can be common to more than one module. There are two arguments: a name and an absolute expression giving the size in bytes of the area named by the symbol. The type of the symbol becomes external. The statement can appear in amy segment. If you think this has something to do with FORTRAN, you are right.

9.1.21. Examples

In the kernel directory, there are several assembly code files that are worth inspecting as examples. However, note that these files, ending with .x, are designed to first be run through the C preprocessor. Thus they contain numerous constructs that are not pure assembler. For true assembler examples, compile any C program provided with MINIX using the –S flag. This will result in packed assembly language. The file can be unpacked using the *libupack* filter.

9.2. BAWK—BASIC AWK

AWK is a programming language devised by Aho, Weinberger, and Kernighan at Bell Labs (hence the name). *Bawk* is a basic subset of it. *Bawk* programs search files for specific patterns and performs "actions" for every occurrence of these patterns. The patterns can be "regular expressions" as used in the *ed* editor. The actions are expressed using a subset of the C language.

The patterns and actions are usually placed in a "rules" file whose name must be the first argument in the command line, preceded by the flag **–f**. Otherwise, the first argument on the command line is taken to be a string containing the rules themselves. All other arguments are taken to be the names of text files on which the rules are to be applied, with – being the standard input. To take rules from the standard input, use **–f –**.

The command:

```
bawk rules prog.*
```

would read the patterns and actions rules from the file *rules* and apply them to all the arguments.

The general format of a rules file is:

<pattern> { <action> } <pattern> { <action> } ...

There may be any number of these <pattern> { <action> } sequences in the rules file. *Bawk* reads a line of input from the current input file and applies every <pattern> { <action> } in sequence to the line.

If the <pattern> corresponding to any { <action> } is missing, the action is applied to every line of input. The default { <action> } is to print the matched input line.

9.2.1. Patterns

The <pattern>s may consist of any valid C expression. If the <pattern> consists of two expressions separated by a comma, it is taken to be a range and the <action> is performed on all lines of input that match the range. <pattern>s may contain "regular expressions" delimited by an @ symbol. Regular expressions can be thought of as a generalized "wildcard" string matching mechanism, similar to that used by many operating systems to specify file names. Regular expressions may contain any of the following characters:

x	An ordinary character
\	The backslash quotes any character
^	A circumflex at the beginning of an expr matches the beginning of a line.
$	A dollar-sign at the end of an expression matches the end of a line.
.	A period matches any single character except newline.

:x A colon matches a class of characters described by the next character:

:a ":a" matches any alphabetic;

:d ":d" matches digits;

:n ":n" matches alphanumerics;

: ": " matches spaces, tabs, and other control characters, such as newline.

* An expression followed by an asterisk matches zero or more occurrences of that expression: "fo*" matches "f", "fo", "foo", "fooo", etc.

+ An expression followed by a plus sign matches one or more occurrences of that expression: "fo+" matches "fo", "foo", "fooo", etc.

– An expression followed by a minus sign optionally matches the expression.

[] A string enclosed in square brackets matches any single character in that string, but no others. If the first character in the string is a circumflex, the expression matches any character except newline and the characters in the string. For example, "[xyz]" matches "xx" and "zyx", while "[^xyz]" matches "abc" but not "axb". A range of characters may be specified by two characters separated by "-".

9.2.2. Actions

Actions are expressed as a subset of the C language. All variables are global and default to int's if not formally declared. Only char's and int's and pointers and arrays of char and int are allowed. *Bawk* allows only decimal integer constants to be used—no hex (0xnn) or octal (0nn). String and character constants may contain all of the special C escapes (\n, \r, etc.).

Bawk supports the "if", "else", "while" and "break" flow of control constructs, which behave exactly as in C.

Also supported are the following unary and binary operators, listed in order from highest to lowest precedence:

<table>
<tr><th>Operator</th><th>Type</th><th>Associativity</th></tr>
<tr><td>() []</td><td>unary</td><td>left to right</td></tr>
<tr><td>! ++ -- - * &</td><td>unary</td><td>right to left</td></tr>
<tr><td>* / %</td><td>binary</td><td>left to right</td></tr>
<tr><td>+ -</td><td>binary</td><td>left to right</td></tr>
<tr><td><< >></td><td>binary</td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>binary</td><td>left to right</td></tr>
<tr><td>== !=</td><td>binary</td><td>left to right</td></tr>
<tr><td>&</td><td>binary</td><td>left to right</td></tr>
<tr><td>^</td><td>binary</td><td>left to right</td></tr>
<tr><td>|</td><td>binary</td><td>left to right</td></tr>
<tr><td>&&</td><td>binary</td><td>left to right</td></tr>
<tr><td>||</td><td>binary</td><td>left to right</td></tr>
<tr><td>=</td><td>binary</td><td>right to left</td></tr>
</table>

Comments are introduced by a '#' symbol and are terminated by the first newline character. The standard "/*" and "*/" comment delimiters are not supported and will result in a syntax error.

9.2.3. Fields

When *bawk* reads a line from the current input file, the record is automatically separated into "fields." A field is simply a string of consecutive characters delimited by either the beginning or end of line, or a "field separator" character. Initially, the field separators are the space and tab character. The special unary operator '$' is used to reference one of the fields in the current input record (line). The fields are numbered sequentially starting at 1. The expression "$0" references the entire input line.

Similarly, the "record separator" is used to determine the end of an input "line," initially the newline character. The field and record separators may be changed programatically by one of the actions and will remain in effect until changed again.

Multiple (up to 10) field separators are allowed at a time, but only one record separator. In either case, they must be changed by strcpy(), not by a simple equate as in the real AWK.

Fields behave exactly like strings; and can be used in the same context as a character array. These "arrays" can be considered to have been declared as:

```
char ($n)[ 128 ];
```

In other words, they are 128 bytes long. Notice that the parentheses are necessary because the operators [] and $ associate from right to left; without them, the statement would have parsed as:

```
char $(I[ 128 ]);
```

which is obviously ridiculous.

If the contents of one of these field arrays is altered, the "$0" field will reflect this change. For example, this expression:

```
*$4 = 'A';
```

will change the first character of the fourth field to an upper- case letter 'A'. Then, when the following input line:

```
120 PRINT "Name      address      Zip"
```

is processed, it would be printed as:

```
120 PRINT "Name      Address      Zip"
```

Fields may also be modified with the strcpy() function (see below). For example, the expression:

```
strcpy( $4, "Addr." );
```

applied to the same line above would yield:

```
120 PRINT "Name      Addr.     Zip"
```

9.2.4. Predefined Variables

The following variables are pre-defined:

FS	Field separator (see below).
RS	Record separator (see below also).
NF	Number of fields in current input record (line).
NR	Number of records processed thus far.
FILENAME	Name of current input file.
BEGIN	A special <pattern> that matches the beginning of input text.
END	A special <pattern> that matches the end of input text.

Bawk also provides some useful built-in functions for string manipulation and printing:

print(arg)	Simple printing of strings only, terminated by '\n'.
printf(arg...)	Exactly the printf() function from C.
getline()	Reads the next record and returns 0 on end of file.
nextfile()	Closes the current input file and begins processing the next file
strlen(s)	Returns the length of its string argument.
strcpy(s,t)	Copies the string "t" to the string "s".
strcmp(s,t)	Compares the "s" to "t" and returns 0 if they match.
toupper(c)	Returns its character argument converted to upper-case.
tolower(c)	Returns its character argument converted to lower-case.
match(s,@re@)	Compares the string "s" to the regular expression "re" and returns the number of matches found (zero if none).

9.2.5. Limitations

The maximum input line is 128 characters. The maximum action is 4K.

9.2.6. Author

Bawk was written by Bob Brodt.

9.3. DE—DISK EDITOR

The *de* program allows a system administrator to examine and modify a MINIX file system device. Commands are available to move to any address on the disk and display the disk block contents. This information may be presented in one of three visual modes: as two-byte words, as ASCII characters or as a bit map. The disk may be searched for a string of characters. If the **–w** option is given, *de* will open the device for writing and words may be modified. Without this flag, writing is prohibited. Lost blocks and files can be recovered using a variety of commands. The **–r** option supports automated recovery of files removed by *unlink*.

9.3.1. Positioning

Disks are divided into blocks (also called "zones") of 1024 bytes. *De* keeps a current address on the disk as a block number and a byte offset within the block. In some visual modes the offset is rounded off, for example, in "word" mode the offset must be even.

There are different types of blocks on a file system device, including a super block, bit maps, i-nodes and data blocks. *De* knows the type of the current block, but will allow most positioning commands and visual modes to function anywhere on the disk.

The *f* command (or PGDN on the keypad) moves forward to the next block, similarly *b* (PGUP) moves backwards one block. *F* (END) moves to the last block and *B* (HOME) moves to the first block.

The arrow keys (or *u*, *d*, *l*, and *r*) change the current address by small increments. The size of the increment depends on the current display mode, as shown below. The various sizes suit each display and pointers move on the screen to follow each press of an arrow key.

Mode	Up	Down	Left	Right
Word	–2	+2	–32	+32
Block	–64	+64	–1	+1
Map	–256	+256	–4	+4

The *g* command allows movement to any specified block. Like all commands that take arguments, a prompt and subsequent input are written to the bottom line of the screen. Numerical entry may be decimal, octal or hexadecimal, for example 234, –1, 070, 0xf3, –X3C.

While checking an i-node one may want to move to a block listed as a zone of the file. The *G* command takes the contents at the current address in the device as a block number and indirectly jumps to that block.

The address may be set to the start of any i-node using the command and supplying an i-node number. The *I* command maps a given file name into an i-node address. The file must exist on the current device and this device must be mounted.

9.3.2. The Display

The first line of the display contains the device name, the name of the current output file (if one is open) and the current search string. If *de* is being run with the **-w** option then the device name is flagged with "(w)." If a string is too long to fit on the line it is marked with "...".

The second line contains the current block number, the total number of blocks, and the type of the current block. The types are: boot, super, i-node bit map, zone bit map, i-nodes and data block. If the current address is within a data block then the string "in use" is displayed if the block corresponds to a set in the zone bit map.

The third line shows the offset in the current block. If the current address is within either the i-node or zone bit maps then the i-node or block number corresponding to the current bit is shown. If the current address is within an i-node then the i-node number and "in use" status is displayed. If the address is within a bit map or i-node block, but past the last usable entry, then the string "padding" is shown.

The rest of the screen is used to display data from the current block. There are three visual display modes: "word," "block," and "map." The *v* command followed by *w*, *b*, or *m* sets the current display mode.

In "word" mode 16 words, of two bytes each, are shown in either base 2, 8, 10 or 16. The current base is displayed to the far right of the screen. It can be changed using the *o* command followed by either an *h* (hexadecimal), *d* (decimal), *o* (octal) or *b* (binary).

De knows where i-nodes are, and will display the contents in a readable format, including the *rwx* bits, the user name and the time field. If the current page is at the beginning of the super block, or an executable file or an *ar* archive, then *de* will also inform the user. In all other cases the contents of the 16 words are shown to the right as equivalent ASCII characters.

In "block" mode a whole block of 1024 bytes is displayed as ASCII characters, 64 columns by 16 lines. Control codes are shown as highlighted characters. If the high order bit is set in any of the 1024 bytes then an "MSB" flag is shown on the far right of the screen, but these bytes are not individually marked.

In "map" mode 2048 bits (256 bytes) are displayed from the top to the bottom (32 bits) and from the left to the right of the screen. Bit zero of a byte is towards the top of the screen. This visual mode is generally used to observe the bit map blocks. The number of set bits displayed is written on the far right of the screen.

9.3.3. Searching

A search for an ASCII string is initiated by the / command. Control characters not used for other purposes may be entered in the search string, for example CTRL-J is an end-of-line character. The search is from the current position to the end of the current device.

Once a search string has been defined by a use of /, the next search may be initiated with the *n* command, (a / followed immediately by an ENTER is equivalent to an *n*).

Whenever a search is in progress *de* will append one . to the prompt line for every 500 blocks searched. If the string is found between the end of the file system and the actual end of the device, then the current address is set to the end of the file system.

Some of the positioning commands push the current address and visual mode in a stack before going to a new address. These commands are *B*, *F*, *g*, *G*, *i*, *I*, *n*, *x* and /. The *p* (previous) command pops the last address and visual mode from the stack. This stack is eight entries deep.

9.3.4. Modifying the File System

The *s* command will prompt for a data word and store it at the current address on the disk. This is used to change information that can not be easily changed by any other means.

The data word is 16 bits wide, it may be entered in decimal, octal or hexadecimal. Remember that the **−w** option must be specified for the *s* command to operate. Be careful when modifying a mounted file system.

9.3.5. Recovering Files

Any block on the disk may be written to an output file. This is used to recover blocks marked as free on the disk. A write command will request a file name the first time it is used, on subsequent writes the data is appended to the current output file.

The name of the current output file is changed using the *c* command. This file should be on a different file system, to avoid overwriting an i-node or block before it is recovered.

An ASCII block is usually recovered using the *w* command. All bytes will have their most significant bit cleared before being written to the output file. Bytes containing '\0' or '\177' are not copied. The *W* command writes the current block (1024 bytes exactly) to the output file.

When a file is deleted using *unlink* the i-node number in the directory is zeroed, but before its removal, it is copied into the end of the file name field. This allows the i-node of a deleted file to be found by searching through a directory. The *x* command asks for the path name of a lost file, extracts the old i-node number and changes the current disk address to the start of the i-node.

Once an i-node is found, all of the freed blocks may be recovered by checking the i-node zone fields, using 'G' to go to a block, writing it back out using 'w', going back to the i-node with *p* and advancing to the next block. This file extraction process is automated by using the *X* command, which goes through the i-node,

indirect and double indirect blocks finding all the block pointers and recovering all the blocks of the file.

The *X* command closes the current output file and asks for the name of a new output file. All of the disk blocks must be marked as free, if they are not the command stops and the file must be recovered manually.

When extracting lost blocks *de* will maintain "holes" in the file. Thus, a recovered sparse file does not allocate unused blocks and will keep its efficient storage scheme. This property of the *X* command may be used to move a sparse file from one device to another.

Automatic recovery may be initiated by the **–r** option on the command line. Also specified is the path name of a file just removed by *unlink*. *De* determines which mounted file system device held the file and opens it for reading. The lost i-node is found and the file extracted by automatically performing an *x* and an *X* command.

The recovered file will be written to */tmp*. *De* will refuse to automatically recover a file on the same file system as */tmp*. The lost file must have belonged to the user. If automatic recovery will not complete, then manual recovery may be performed.

9.3.6. Miscellaneous

The user can terminate a session with *de* by typing *q*, CTRL-D, or the key associated with SIGQUIT.

The *m* command invokes the MINIX *sh* shell as a subprocess.

For help while using *de* use *h*.

9.3.7. Command Summary

PGUP	b	Back one block
PGDN	f	Forward one block
HOME	B	Goto first block
END	F	Goto last block
UP	u	Move back 2/64/256 bytes
DOWN	d	Move forward 2/64/256 bytes
LEFT	l	Move back 32/1/4 bytes
RIGHT	r	Move forward 32/1/4 bytes
	g	Goto specified block
	G	Goto block indirectly
	i	Goto specified i-node
	I	Filename to i-node
	/	Search
	n	Next occurrence

p	Previous address	
	h	Help
EOF	q	Quit
	m	MINIX shell
	v	Visual mode (w b m)
	o	Output base (h d o b)
	c	Change file name
	w	Write ASCII block
	W	Write block exactly
	x	Extract lost directory entry
	X	Extract lost file blocks
	s	Store word

NOTES: When entering a line in response to a prompt from *de* there are a couple of editing characters available. The previous character may be erased by typing CTRL-H and the whole line may be erased by typing CTRL-U. ENTER terminates the input. If DELETE or a non-ASCII character is typed then the command requesting the input is aborted.

The commands *G*, *s* and *X* will only function if the current visual display mode is "word." The commands *i*, *I* and *x* change the mode to "word" on completion. The commands *G* and *I* change the mode to "block". These restrictions and automatic mode conversions are intended to aid the user.

The "map" mode uses special graphic characters, and only functions if the user is at the console.

De generates warnings for illegal user input or if erroneous data is found on the disk, for example a corrupted magic number. Warnings appear in the middle of the screen for two seconds, then the current page is redrawn. Some minor errors, for example, setting an unknown visual mode, simply ring the bell. Major errors, for example I/O problems on the file system device cause an immediate exit from *de*.

The i-node and zone bit maps are read from the device when *de* starts up. These determine whether "in use" or "not in use" is displayed in the status field at the top of the screen. The bit maps are not re-read while using *de* and will become out-of-date if observing a mounted file system.

De requires termcap definitions for "cm" and "cl". Furthermore, "so" and "se" will also be used if available. The ANSI strings generated by the keypad arrows are recognized, as well as any single character codes defined by "ku", "kd", "kl" and "kr".

9.3.8. Author

The *de* program was written by Terrence Holm.

9.4. DIS88—DISASSEMBLER FOR THE 8088 [IBM]

Dis88 disassembles 8088 object code to the assembly language format used by MINIX. It makes full use of symbol table information, supports separate instruction and data space, and generates synthetic labels when needed. It does not support 8087 mnemonics, symbolic data segment references, or the ESC mnemonic.

The program is invoked by:

```
dis88 [–o] infile [outfile]
```

The –o flag causes object code to be listed. If no outfile is given, *stdout* is used.

The text segment of an object file is always padded to an even address. In addition, if the file has split I/D space, the text segment will be padded to a paragraph boundary (i.e., an address divisible by 16). Due to padding, the disassembler may produce a few spurious, but harmless, instructions at the end of the text segment.

Because the information to which initialized data refers cannot generally be inferred from context, the data segment is treated literally. Byte values (in hexadecimal) are output, and long stretches of null data are represented by appropriate *.zerow* pseudo-ops. Disassembly of the bss segment, on the other hand, is quite straightforward, because uninitialized data is all zero by definition. No data is output in the bss segment, but symbolic labels are output as appropriate.

The output of operands in symbolic form is complicated somewhat by the existence of assembler symbolic constants and segment override opcodes. Thus, the program's symbol lookup routine attempts to apply a certain amount of intelligence when it is asked to find a symbol. If it cannot match on a symbol of the preferred type, it may output a symbol of some other type, depending on preassigned (and somewhat arbitrary) rankings within each type. Finally, if all else fails, it will output a string containing the address sought as a hex constant. For user convenience, the targets of branches are also output, in comments, as hexadecimal constants.

9.4.1. Error Messages

Various error messages may be generated as a result of problems encountered during the disassembly. They are listed below

Cannot access input file	– Input file cannot be opened or read
Cannot open output file	– Output file cannot be created
Input file not in object format	– Bad magic number
Not an 8086/8088 object file	– CPU ID of the file header is incorrect
Reloc table overflow	– Relocation table exceeds 1500 entries
Symbol table overflow	– Symbol table exceeds 1500 entries
Lseek error	– Input file corrupted (should never happen)
Warning: no symbols	– Symbol table is missing (use ast)
Cannot reopen input file	– Input file was removed during execution

9.4.2. Author

Dis88 was written and copyrighted by G. M. Harding and is included here by permission. It may be freely redistributed provided that complete source code, with all copyright notices, accompanies any redistribution. This provision also applies to any modifications you may make. You are urged to comment such changes, giving, as a minimum, your name and complete address.

9.5. ELLE—FULL-SCREEN EDITOR

ELLE (ELLE Looks Like Emacs) is an Emacs clone for MINIX. ELLE is not full Emacs but it has about 80 commands and is quite fast.

9.5.1. Key bindings

Mined only has a small number of commands. All of them are either of the form CTRL-x or are on the numeric keypad. Emacs, in contrast, has so many commands, that not only are all the CTRL-x commands used up, but so are all the ESC x (escape followed by x; escape is not a shift character, like CTRL). Even this is not enough, so CTRL-X is used as a prefix for additional commands. Thus CTRL-X CTRL-L is a command, and so is CTRL-X K. Note that what is conventionally written as CTRL-X K really means CTRL-X k. In some contexts it is traditional to write CTRL-X as ^X. Please note that they mean the same thing.

As a result, many Emacs commands need three or four key strokes to execute. Some people think 3-4 key strokes is too many. For this reason, Emacs and ELLE allow users to assign their own key bindings. In ELLE this is done with "user profiles." A user profile is a file listing which function is invoked by which key stroke. The user profile is then compiled by a program called ellec into binary form. When ELLE starts up it checks to see if a file .ellepro.b1 exists in $HOME. If it does, this file is read in and overrides the default bindings.

A user profile that simulates the *mined* commands fairly well is provided. Its installation is described later. If you have never used Emacs, it is suggested that you use the *mined* profile. If you normally use Emacs, then do not install the *mined* profile. You can also make your own using *ellec*. There is no Mock Lisp.

ELLE has a character-oriented view of the world, not a line oriented view, like *ed*. It does not have magic characters for searching. However, you can use line feed in search patterns. For example, to find a line consisting of the three characters "foo" all by themselves on a line, using the mined bindings (see below), use the pattern: CTRL-\ CTRL-J f o o CTRL-\ CTRL-J. The CTRL- means to interpret the next character literally, in this case it is CTRL-J, which is line feed. You can also

search for patterns involving multiple lines. For example, to find a line ending in an "x" followed by a line beginning with a "y", use as pattern: x CTRL- CTRL-J y.

9.5.2. Mined Key Bindings

These are the key bindings if the binary user profile, *.ellepro.bl*, is installed in $HOME. The ESCAPE key followed by a number followed by a command causes that command to be executed "number" times. This applies both to control characters and insertable characters. CTRL-X refers to a "control character." ESC x refers to an escape character followed by x. In other words, ^X is a synonym for CTRL-X. ^X Y refers to CTRL-X followed by y. To abort the current command and go back to the main loop of the editor, type CTRL-G, rather than CTRL-\

Only a few commands are of the form CTRL-X Y. All of these are also bound to CTRL-X CTRL-Y, so you can hold down CTRL and then hit X Y, or release control after the X, as you prefer.

The key bindings that are not listed should not be used. Some of them actually do things. For example, the ANSI escape codes ESC [x are bound to ^X Y for a variety of y.

Some commands work on regions. A region is defined as the text between the most recently set mark and the cursor.

9.5.3. Mined Commands

If the *mined* profile, *.ellepro.bl* is installed in your home directory, the following commands will work.

CURSOR MOTION

arrows	Move the cursor in the indicated direction
CTRL-A	Move cursor to start of current line
CTRL-Z	Move cursor to end of current line
CTRL-F	Move cursor forward word
CTRL-B	Move cursor backward to start of previous word

SCREEN MOTION

Home key	Move to first character of the file
End key	Move to last character of the file
PgUp key	Scroll window up 22 lines (closer to start of the file)
PgDn key	Scroll window down 22 lines (closer to end of the file)
CTRL-U	Scroll window up 1 line
CTRL-D	Scroll window down 1 line
ESC ,	Move to top of screen
CTRL-_	Move to bottom of screen

MODIFYING TEXT

DEL key	Delete the character under the cursor
Backsp	Delete the character to left of the cursor
CTRL-N	Delete the next word
CTRL-P	Delete the previous word
CTRL-T	Delete tail of line (all characters from cursor to end of line)
CTRL-O	Open up the line (insert line feed and back up)
ESC G	Get and insert a file at the cursor position (CTRL-G in mined)

REGIONS

CTRL-^	Set mark at current position for use with CTRL-C and CTRL-K
CTRL-C	Copy the text between the mark and the cursor into the buffer
CTRL-K	Delete text between mark and cursor; also copy it to the buffer
CTRL-Y	Yank contents of the buffer out and insert it at the cursor

MISCELLANEOUS

numeric +	Search forward (prompts for expression)
numeric –	Search backward (prompts for expression)
CTRL-]	ESC n CTRL-[goes to line n (slightly different syntax than mine
CTRL-R	Global replace pattern with string (from cursor to end)
CTRL-L	Replace pattern with string within the current line only
CTRL-W	Write the edited file back to the disk
CTRL-S	Fork off a shell (use CTRL-D to get back to the editor)
CTRL-G	Abort whatever the editor was doing and wait for command (CT
CTRL-E	Redraw screen with cursor line positioned in the middle
CTRL-V	Visit (edit) a new file
CTRL-Q	Write buffer to a file
ESC X	Exit the editor

9.5.4. Non-Mined Commands

CURSOR MOTION

ESC P	Forward paragraph (a paragraph is a line beginning with a dot)
ESC]	Backward paragraph
ESC .	Indent this line as much as the previous one

MODIFYING TEXT

CTRL-\	Insert the next character (used for inserting control characters)
ESC T	Transpose characters
ESC W	Transpose words
ESC =	Delete white space (horizontal space)
ESC |	Delete blank lines (vertical space)

REGIONS

ESC M	Mark current paragraph
ESC ^	Exchange cursor and mark
ESC Y	Yank back the next-to-the-last kill (CTRL-Y yanks the last one)
ESC A	Append next kill to kill buffer

KEYBOARD MACROS

ESC /	Start Keyboard Macro
ESC \	End Keyboard Macro
ESC *	View Keyboard Macro (the PrtSc key on the numeric pad is also a
ESC E	Execute Keyboard Macro

WINDOW MANAGEMENT

^X 1	Enter one window mode
^X 2	Enter two window mode
^X L	Make the current window larger
^X P	Make the window more petit/petite (Yes, Virginia, they are Englis
^X N	Next window
^X W	New window

BUFFER MANAGEMENT

numeric 5	Display the list of current files and buffers
ESC B	Select a buffer
ESC S	Select an existing buffer
ESC N	Mark a buffer as NOT modified (even if it really is)

UPPER AND LOW CASE MANIPULATION

ESC I	Set first character of word to upper case
ESC C	Capitalize current word
ESC O	Make current word ordinary (i.e., lower case)
ESC U	Set entire region between mark and cursor to upper case
ESC L	Set entire region between mark and cursor to lower case

MISCELLANEOUS

ESC F	Find file and read it into its own buffer
ESC Z	Incremental search
ESC Q	Like CTRL-R, but queries at each occurrence (type ? for options)
ESC R	Reset the user profile from a file
ESC H	Help (ELLE prompts for the 1 or 2 character command to describe
ESC ;	Insert a comment in a C program (generates /* */ for you)
^X X	Exit the editor (same as ESC X and CTRL-X CTRL-X)

The major differences between ELLE with the *mined* profile and *mined* itself are:

1. The definition of a "word" is different for forward and backward word
2. The mark is set with CTRL-^ instead of CTRL-@
3. Use CTRL-G to abort a command instead of CTRL-\
4. Use CTRL- to literally insert the next character, instead of ALT
5. CTRL-E adjusts the window to put the cursor in the middle of it
6. To get and insert a file, use ESC G instead of CTRL-G
7. To go to line n, type ESC n CTRL-[instead of CTRL-[n
8. You exit with CTRL-X CTRL-X and then answer the question with "y".
9. There are many new commands, windows, larger files, etc.

9.5.5. Emacs Key Bindings

If you do not have the *mined* profile installed, you get the standard Emacs key bindings. These are listed below. Commands not listed are not implemented.

CURSOR MOVEMENT

CTRL-F	Forward one character.
CTRL-B	Backward one character.
CTRL-H	Same as CTRL-B: move backward one character.
ESC F	Forward one word.
ESC B	Backward one word.
CTRL-A	Beginning of current line.
CTRL-E	End of current line.
CTRL-N	Next line (goes to the next line).
CTRL-P	Previous line (goes to the previous line).
CTRL-V	Beginning of next screenful.
ESC V	Beginning of previous screenful.
ESC]	Forward Paragraph.
ESC [	Backward Paragraph.
ESC <	Beginning of whole buffer.
ESC >	End of whole buffer.

DELETING

CTRL-D	Deletes forward one character (the one the cursor is under).
DELETE	Deletes backward one character (the one to left of cursor).
ESC D	Kills forward one word.
ESC DEL	Kills backward one word.
CTRL-K	Kills the rest of the line (to the right of the cursor).
ESC \	Deletes spaces around the cursor.
^X CTRL-O	Deletes blank lines around the cursor.

CASE CHANGE

ESC C	Capitalizes word : first letter becomes uppercase; rest lower
ESC L	Makes the whole next word lowercase.
ESC U	Makes the whole next word uppercase.
^X CTRL-L	Makes whole region lowercase.
^X CTRL-U	Makes whole region uppercase.

SEARCHING (If no string is given, previous string is used)

CTRL-S	Incremental Search forward; prompts "I-search:"
CTRL-R	Reverse Incremental Search; prompts "R-search:" During an incre search, the following characters have special effects:

"normal"	- Begin searching immediately.
^G	- Cancel I-search, return to start.
DEL	- Erase last char, return to last match.
^S, ^R	- Repeat search (or change direction).
ESC or CR	- Exit I-search at current point.

ESC %	Query Replace. Interactive replace. Type "?" to see options.
^X %	Replace String. Like Query Replace, but not interactive

MARKING AREAS

CTRL-^	Set mark
^X CTRL-X	Exchange cursor and mark.
ESC H	Mark Paragraph. Sets mark and cursor to surround a para.
CTRL-W	Wipe-out -- kills a "region":
ESC W	Copy region. Like CTRL-W then CTRL-Y but modifies buffer
CTRL-Y	Yanks-back (un-kills) whatever you have most recently killed.
ESC Y	Yanks-back (un-kills) the next most recently killed text.
ESC CTRL-W	Append Next Kill. Accumulates stuff from several kills

FILLING TEXT

ESC Q	Fill the paragraph to the size of the Fill Column.
ESC G	Fill the region.
^X F	Set Fill Column. ESC Q will use this line size.
^X .	Set Fill Prefix. Asks for prefix string
^X T	Toggles Auto Fill Mode.

WINDOWS

^X 2	Make two windows (split screen).
^X 1	Make one window (delete window) (make one screen).
^X O	Go to Other window.
^X ^	Grow window: makes current window bigger.

BUFFERS

^X CTRL-F	Find a file and make a buffer for it.
^X B	Select Buffer: goes to specified buffer or makes new one
^X CTRL-B	Show the names of the buffers used in this editing session.
^X K	Kill Buffer.
ESC tilde	Say buffer is not modified.
^X CTRL-M	Toggle EOL mode (per-buffer flag).

KEYBOARD MACRO

^X (	Start collecting a keyboard macro.
^X)	Stop collecting.
^X E	Execute the collected macro.
^X *	Display the collected macro.

FILES

^X CTRL-I	Insert a file where cursor is.
^X CTRL-R	Read a new file into current buffer.
^X CTRL-V	Same as ^X ^R above (reads a file).
^X CTRL-W	Write buffer out to new file name.
^X CTRL-S	Save file: write out buffer to its file name.
^X CTRL-E	Write region out to new file name.

MISCELLANEOUS

^X CTRL-Z	Exit from ELLE.
^X !	Escape to shell (CTRL-D to return)
CTRL-O	Open up line
LINEFEED	Same as typing RETURN and TAB.
CTRL-T	Transposes characters.
ESC T	Transposes words.
CTRL-U	Makes the next command happen four times.
CTRL-U number	Makes the next command happen "number" times.
ESC number	Same as CTRL-U number.
CTRL-L	Refreshes screen.
CTRL-U CTRL-L	Refresh only the line cursor is on.
CTRL-U n CTRL-L	Change window so the cursor is on line n
CTRL-Q	Quote: insert the next character no matter what it is.
CTRL-G	Quit: use to avoid answering a question.
ESC ;	Inserts comment (for writing C programs).
ESC I	Inserts indentation equal to previous line.
ESC M	Move to end of this line's indentation.
CTRL-_	Describe a command (if the command database is online)

UNUSED CONTROLS

CTRL-C	Not used.
CTRL-Z	Not used.
CTRL-]	Not used.

9.5.6. Installing ELLE on MINIX

To install ELLE, you will need the following files:

elle	– executable binary of the editor
ellec	– executable binary of the profile compiler
.ellepro.e	– *mined* profile in source form
.ellepro.b1	– *mined* profile in binary form
help.dat	– help file

To install ELLE, please proceed as follows:

1. Check to see if */etc/termcap* is present and has an entry for "minix".
2. Set the environment properly by typing:

   ```
   TERM=minix
   ```

   You can also put it in the appropriate *.profile*, but be sure to also include a line:

   ```
   export TERM
   ```

   You can check the current environment with *printenv*. If the entry:

   ```
   TERM=minix
   ```

   does not appear, ELLE will not work.
3. Install the files *elle* and *ellec* in your */bin* or */usr/bin* directory.
4. Install *help.dat* in */usr/src/elle/help.dat*
5. If you want to use the mined-like commands, install *.ellepro.b1* in your home directory.
6. Type:

   ```
   elle filename
   ```

   and you are up and running.

It is possible to create your own user profile. The mechanism is different from Emacs, since ELLE does not have Mock Lisp. Proceed as follows.

1. Modify *.ellepro.e* to suit your taste.
2. Install *.ellepro.e* in your home directory.
3. Type:

   ```
   ellec –Profile
   ```

4. Check to see if *.ellepro.bl* has been created. If it has, you are ready to go.

9.5.7. Author

ELLE was written by Ken Harrenstien of SRI (klh@sri.com).

9.6. ELVIS—A CLONE OF THE BERKELEY VI EDITOR

Elvis is a full-screen editor closely modeled on the famous Berkeley *vi* editor. It provides essentially the same interface to the user as *vi*, but the code is completely new, written from scratch. This document provides a brief introduction to *vi*. It is not intended as a tutorial for beginners. Most books on UNIX cover *vi*.

Like *vi*, *elvis* can operate as a screen editor (*vi* mode) or as a line editor (*ex*) mode. It can be called either as *elvis vi*,or as *ex*, depending on which is desired. They are all links to the same file.

9.6.1. Vi Commands

Below is a list of the *vi* commands supported. The following symbols are used in the table:

count	Integer parameter telling how many or how much
key	One character parameter to the command
inp	Interactive input expected
mv	Indicates how much for commands like *delete* and *change*:
	(Previous sentence
	) Next sentence
	{ Previous paragraph
	] Next paragraph (delimited by blank line, *.PP*, *.LP*, *.IP* etc.)
	[Previous section (delimited by *.SH* or *.NH*)
	A repeated command character means the scope is this line

MOVE	Indicates commands that may also be used where *mv* is specified
EDIT	These commands affect text and may be repeated by the . command

In addition to the above notation, the caret (^) is used as an abbreviation for CTRL. For example, ^A means CTRL-A.

Count	Command	Description	Type
	^A	(Not defined)	
	^B	Move toward the top of the file by 1 screenful	
	^C	(Not defined)	
count	^D	Scroll down *count* lines (default 1/2 screen)	
count	^E	Scroll up *count* lines	
	^F	Move toward the bottom of the file by 1 screenful	
	^G	Show file status, and the current line	
count	^H	Move left, like *h*	MOVE
	^I	(Not defined)	
count	^J	Move down	MOVE
	^K	(Not defined)	
	^L	Redraw the screen	
count	^M	Move to the front of the next line	MOVE
count	^N	Move down	MOVE
	^O	(Not defined)	
count	^P	Move up	MOVE
	^Q	(Not defined)	
	^R	Redraw the screen	
	^S	(Not defined)	
	^T	(Not defined)	
count	^U	Scroll up *count* lines (default 1/2 screen)	
	^V	(Not defined)	
	^W	(Not defined)	
	^X	(Not defined)	
count	^Y	Scroll down *count* lines	
	^Z	(Not defined)	
	ESC	(Not defined)	
	^\	(Not defined)	
	^]	If the cursor is on a tag name, go to that tag	
	^^	(Not defined)	
	^_	(Not defined)	
count	SPACE	Move right,like *l*	MOVE
	! mv	Run the selected lines thru an external filter program	
	“ key	Select which cut buffer to use next	
	#	(Not defined)	
	$	Move to the rear of the current line	MOVE
	%	move to the matching (){}[] character	MOVE

	&	(Not defined)	
	' key	Move to a marked line	MOVE
count	(	Move backward *count* sentences	MOVE
count	)	Move forward *count* sentences	MOVE
	*	(Not defined)	
count	+	Move to the front of the next line	MOVE
count	,	Repeat the previous [*fFtT*] but the other way	MOVE
count	-	Move to the front of the preceding line	MOVE
	.	Repeat the previous "edit" command	
	/	Text search forward for a given regular expr	MOVE
	0	If not part of count, move to 1st char of this line	MOVE
	1	Part of count	
	2	Part of count	
	3	Part of count	
	4	Part of count	
	5	Part of count	
	6	Part of count	
	7	Part of count	
	8	Part of count	
	9	Part of count	
	:	Text. Run single *ex* cmd	
count	;	Repeat the previous [fFtT] cmd	MOVE
	< mv	Shift text left	EDIT
	=	(Not defined)	
	> mv	Shift text right	EDIT
	? text	Search backward for a given regular expression	MOVE
	@	(Not defined)	
count	A inp	Append at end of the line	EDIT
count	B	Move back Word	MOVE
	C inp	Change text from cursor through end of line	EDIT
	D	Delete text from cursor through end of line	EDIT
count	E	Move end of Word	MOVE
count	F key	Move leftward to a given character	MOVE
count	G	Move to line #*count* (default is the bottom line)	MOVE
count	H	Move to home row (the line at the top of the screen)	
count	I inp	Insert at the front of the line (after indents)	EDIT
count	J	Join lines, to form one big line	EDIT
	K	Look up keyword	
count	L	Move to last row (the line at the bottom of the screen)	
	M	Move to middle row (the line in the middle)	
	N	Repeat previous search, but the opposite way	MOVE
count	O inp	Open up a new line above the current line	EDIT
	P	Paste text before the cursor	

	Q	Quit to EX mode	
	R inp	Overtype	EDIT
count	S inp	Change lines, like *count*cc	
count	T key	Move leftward *almost* to a given character	MOVE
	U	Undo all recent changes to the current line	
	V	(Not defined)	
count	W	Move forward *count* Words	MOVE
count	X	Delete the character(s) to the left of the cursor	EDIT
count	Y	Yank text line(s) (copy them into a cut buffer)	
	Z Z	Save the file & exit	
	[[	Move back 1 section	MOVE
	(Not defined)		
	]]	Move forward 1 section	MOVE
	^	Move to the front of the current line (after indent)	MOVE
	~	(Not defined)	
	‘ key	Move to a marked character	MOVE
count	a inp	Insert text after the cursor	EDIT
count	b	Move back *count* words	MOVE
	c mv	Change text	EDIT
	d mv	Delete text	EDIT
count	e	Move forward to the end of the current word	MOVE
count	f key	Move rightward to a given character	MOVE
	g	(Not defined)	
count	h	Move left	MOVE
count	i inp	Insert text at the cursor	EDIT
count	j	Move down	MOVE
count	k	Move up	MOVE
count	l	Move right	MOVE
	m key	Mark a line or character	
	n	Repeat the previous search	MOVE
count	o inp	Open a new line below the current line	EDIT
	p	Paste text after the cursor	
	q	(Not defined)	
count	r key	Replace *count* chars by a given character	EDIT
count	s inp	Replace *count* chars with text from the user	EDIT
count	t key	Move rightward *almost* to a given character	MOVE
	u	Undo the previous edit command	
	v	(Not defined)	
count	w	Move forward *count* words	MOVE
count	x	Delete the character that the cursor's on	EDIT
	y mv	Yank text (copy it into a cut buffer)	
	z key	Scroll current line to the screen's +=top -=bottom .=middle	
count	{	Move back *count* paragraphs	MOVE

count	|	Move to column *count* (the leftmost column is 1)	
count	}	Move forward *count* paragraphs	MOVE
	~	Switch a character between upper & lower case	EDIT
	DEL	(Not defined)	

9.6.2. Ex Commands

Below is a list of the *ex* commands supported. All can be abbreviated.

General

[line]	append	
	args	[files]
	cd	[directory]
	chdir	[directory]
[line][,line]	change	
[line][,line]	copy	line
[line][,line]	debug[!]	
[line][,line]	Delete	[''x]
	edit[!]	[file]
	ex[!]	[file]
	file	
[line][,line]	global	/regexp/ command
[line]	Insert	
[line][,line]	join	
[line][,line]	list	
	map[!]	key mapped_to
[line]	mark	x
	mkexrc	
[line][,line]	Move	line
	next[!]	[files]
	Next[!]	
	previous[!]	
[line][,line]	print	
[line]	put	[''x]
	quit[!]	
[line]	read	file
	rewind[!]	
	set	[options]
[line][,line]	substitute	/regexp/replacement/[p][g]
	tag[!]	tagname
[line][,line]	to	line
	Undo	

	unmap[!]	key
	validate[!]	
	version	
[line][,line]	vglobal	/regexp/ command
	visual	
	wq	
[line][,line]	write[!]	[[>>]file]
	xit[!]	
[line][,line]	yank	["x]
[line][,line]	!	command
[line][,line]	<	
[line][,line]	=	
[line][,line]	>	

Text Entry

[line]	append
[line][,line]	change ["x]
[line]	Insert

The (a)ppend command inserts text after the specified line.

The (i)nsert command inserts text before the specified line.

The (c)hange command copies the range of lines into a cut buffer, deletes them, and inserts new text where the old text used to be.

For all of these commands, you indicate the end of the text you're inserting by hitting ˆD or by entering a line which contains only a period.

Cut & Paste

[line][,line]	Delete ["x]
[line][,line]	yank ["x]
[line]	put[!] ["x]
[line][,line]	copy line
[line][,line]	to line
[line][,line]	Move line

The (d)elete command copies the specified range of lines into a cut buffer, and then deletes them.

The (y)ank command copies the specified range of lines into a cut buffer, but does *not* delete them.

The (pu)t command inserts text from a cut buffer after the specified line—or before it if the ! is present.

The (co)py and (t)o commands yank the specified range of lines and then immediately paste them after some other line.

The (m)ove command deletes the specified range of lines and then immediately pastes them after some other line. If the destination line comes after the deleted text, then it will be adjusted automatically to account for the deleted lines.

Displaying Text

```
[line][,line]   print
[line][,line]   list
```

The (p)rint command displays the specified range of lines.

The (l)ist command also displays them, but it is careful to make control characters visible.

Global Operations

```
[line][,line]   global /regexp/ command
[line][,line]   vglobal /regexp/ command
```

The (g)lobal command searches through the lines of the specified range (or through the whole file if no range is specified) for lines that contain a given regular expression. It then moves the cursor to each of these lines and runs some other command on them.

The (v)global command is similar, but it searches for lines that *do not* contain the regular expression.

Line Editing

```
[line][,line]   join
[line][,line]   ! program
[line][,line]   <
[line][,line]   >
[line][,line]   substitute /regexp/replacement/[p][g]
```

The (j)oin command concatenates all lines in the specified range together to form one big line. If only a single line is specified, then the following line is catenated onto it.

The ! command runs an external filter program, and feeds the specified range of lines to it's stdin. The lines are then replaced by the output of the filter. A typical example would be ":'a,'z!sort -n" to sort the lines 'a,'z according to their numeric values.

The < and > commands shift the specified range of lines left or right, normally by the width of 1 tab character. The "shiftwidth" option determines the shifting amount.

The (s)ubstitute command finds the regular expression in each line, and replaces it with the replacement text. The "p" option causes the altered lines to be printed, and the "g" option permits all instances of the regular expression to be found & replaced. (Without "g", only the first occurrence is replaced.)

Undo

```
              undo
```

The (u)ndo command restores the file to the state it was in before your most recent command which changed text.

Configuration & Status

```
              map[!] [key mapped_to]
              unmap[!] key
              set [options]
              mkexrc
[line]        mark x
              visual
              version
[line][,line] =
              file
```

The (ma)p command allows you to configure *elvis* to recognize your function keys, and treat them as though they transmitted some other sequence of characters. Normally this mapping is done only when in the visual command mode, but with the [!]present it will map keys under all contexts. When this command is given with no arguments, it prints a table showing all mappings currently in effect. When called

with two arguments, the first is the sequence that your function key really sends, and the second is the sequence that you want *elvis* to treat it as having sent.

The (unm)ap command removes key definitions that were made via the map command.

The (se)t command allows you examine or set various options. With no arguments, it displays the values of options that have been changed. With the single argument "all" it displays the values of all options, regardless of whether they've been explicitly set or not. Otherwise, the arguments are treated as options to be set.

The (mk)exrc command saves the current configuration to a file called *.exrc* in the current directory.

The mar(k) command defines a named mark to refer to a specific place in the file. This mark may be used later to specify lines for other commands.

The (vi)sual command puts the editor into visual mode. Instead of emulating ex, *elvis* will start emulating vi.

The (ve)rsion command tells you that what version of *elvis* this is.

The = command tells you what line you specified, or, if you specified a range of lines, it will tell you both endpoints and the number of lines included in the range.

The file command tells you the name of the file, whether it has been modified, the number of lines in the file, and the current line number.

Multiple Files

args [files]
next[!] [files]
Next[!]
previous[!]
rewind[!]

When you invoke *elvis* from your shell's command line, any filenames that you give to *elvis* as arguments are stored in the args list. The (ar)gs command will display this list, or define a new one.

The (n)ext command switches from the current file to the next one in the args list. You may specify a new args list here, too.

The (N)ext and (pre)vious commands (they're really aliases for the same command) switch from the current file to the preceding file in the args list.

The (rew)ind command switches from the current file to the first file in the args list.

Switching Files

```
edit[!] [file]
tag[!] tagname
```

The (e)dit command allows to switch from the current file to some other file. This has nothing to do with the args list, by the way.

The (ta)g command looks up a given tagname in a file called "tags". This tells it which file the tag is in, and how to find it in that file. *Elvis* then switches to the tag's file and finds the tag.

Exiting

```
quit[!]
wq
xit
```

The (q)uit command exits from the editor without saving your file.

The (wq) and (x)it commands (really two names for the same command) both write the file before exiting.

File I/O

```
[line]          read file
[line][,line]   write[!][[>>]file]
```

The (r)ead command gets text from another file and inserts it after the specified line.

The (w)rite command writes the whole file, or just part of it, to some other file. The !, if present, will permit the lines to be written even if you've set the readonly option. If you precede the filename by >> then the lies will be appended to the file.

Directory

cd [directory]
chdir [directory]
shell

The (cd) and (chd)ir commands (really two names for one command) switch the current working directory.

The (sh)ell command starts an interactive shell.

Debugging

[line][,line] debug[!]
validate[!]

These commands are only available if you compile *elvis* with the **-DDEBUG** flag.

The de(b)ug command lists stats for the blocks which contain the specified range of lines. If the ! is present, then the contents of those blocks is displayed, too.

The (va)lidate command checks certain variables for internal consistency. Normally it does not output anything unless it detects a problem. With the !, though, it will always produce *some* output.

9.6.3. Extensions

In addition to the standard commands, a variety of extra features are present in *elvis* that are not present in *vi*. They are described below.

.exrc

Elvis first runs a *.exrc* file (if there is one) from your $HOME directory. After that, it runs a *.exrc* (if there is one) from the current directory. The one in the current directory may override settings made by the one in the $HOME directory.

:mkexrc
:mk

This EX command saves the current :set and :map configurations in the ".exrc" file in your current directory.

:args
:ar

You can use the :args command to define a new args list, as in:

```
:args *.h
```

After you have defined a new args list, the next time you issue a :next command *elvis* will switch to the first file of the new list.

:Next
:previous
:N
:pre

These commands move backwards through the args list.

zz

In VI, the (lowercase) "zz" command will center the current line on the screen, like "z="

The default count value for . is the same as the previous command which . is meant to repeat. However, you can supply a new count if you wish. For example, after "3dw", "." will delete 3 words, but "5." will delete 5 words.

The text which was most recently input (via a "cw" command, or something similar) is saved in a cut buffer called ". (which is a pretty hard name to write in an English sentence). You can use this with the "p" or "P" commands thusly:

```
".p
```

K

You can move the cursor onto a word and press shift-K to have *elvis* run a reference program to look that word up. This command alone is worth the price of admission! See the ctags and ref programs.

input

You can backspace back past the beginning of the line. If you type CTRL-A, then the text that you input last time is inserted. You will remain in input mode, so you can backspace over part of it, or add more to it. (This is sort of like CTRL-@ on the real vi, except that CTRL-A really works.)

Real *vi* can only remember up to 128 characters of input, but *elvis* can remember any amount.

:set charattr
:se ca

Elvis can display "backslash-f" style character attributes on the screen as you edit. The following example shows the recognized attributes:

normal **boldface** *italics*

NOTE: you must compile *elvis* without the –DSET_NOCHARATTR flag for this to work.

9.6.4. Omissions

A few *vi* features are missing. The replace mode is a hack. It does not save the text that it overwrites.

Long lines are displayed differently—where the real vi would wrap a long line onto several rows of the screen, *elvis* simply displays part of the line, and allows you to scroll the screen sideways to see the rest of it.

The ":preserve" and ":recover" commands are missing, as is the **–r** flag. ":Preserve" is practically never used and since use of ":recover*(CQ is so rare, it was decided to implement it as a separate program. There's no need to load the recovery code into memory every time you edit a file.

LISP support is missing. The "@" and ":@" commands are missing. You cannot APPEND to a cut buffer.

9.6.5. Options

A variety of options can be set as described below:

Name	Abbr	Type	Default	Description
autoindent	as	Bool	FALSE	autoindent during input?
autowrite	aw	Bool	FALSE	write file for :n command?
charattr	ca	Bool	FALSE	display bold & underline chars?
columns	co	Number	80	width of screen, in characters
directory	dir	String	/usr/tmp	where tmp files are kept
errorbells	eb	Bool	TRUE	ring bell on error?
exrefresh	er	Bool	TRUE	EX mode calls write() often?
ignorecase	ic	Bool	FALSE	searches: upper/lowercase OK?
keytime	kt	Number	1	allow slow receipt of ESC seq?
keywordprg	kp	String	/usr/bin/ref	program to run for shift-K
lines	ln	Number	25	height of screen, in lines
list	li	Bool	FALSE	show tabs as "^I"?
magic	ma	Bool	TRUE	searches: allow metacharacters?
paragraphs	pa	String	PPppPApa	paragraphs start with .PP, etc.

readonly	ro	Bool	FALSE	no file should be written back?
report	re	Number	5	report changes to X lines?
scroll	sc	Number	12	default #lines for ^U and ^D
sections	se	String	SEseSHsh	sections start with .SE, etc.
shell	sh	String	*/bin/sh*	shell program, from environment
shiftwidth	sw	Number	8	width of < or > commands
sidescroll	ss	Number	8	#chars to scroll sideways by
sync	sy	Bool	FALSE	call sync() after each change?
tabstop	ts	Number	8	width of a tab character
term	te	String	"?"	terminal type, from environment
vbell	vb	Bool	TRUE	use visible bell if possible?
warn	wa	Bool	TRUE	warn if file not saved for :!cmd
wrapmargin	wm	Number	0	Insert newline after which col?
wrapscan	ws	Bool	TRUE	searches: wrap at EOF?

autoindent

During input mode, the autoindent option will cause each added line to begin with the same amount of leading whitespace as the line above it. Without autoindent, added lines are initially empty.

autowrite

When you're editing one file and decide to switch to another—via the :tag command, or :next command, perhaps—if your current file has been modified, then *elvis* will normally print an error message and refuse to switch.

However, if the autowrite option is on, then *elvis* will write the modified version of the current file and successfully switch to the new file.

charattr

Many text formatting programs allow you to designate portions of your text to be underlined, italicized, or boldface by embedding the special strings \fU, \fI, and \fB in your text. The special string \fR marks the end of underlined or boldface text.

Elvis normally treats those special strings just like any other text. However, if the *charattr* option is on, then *elvis* will interpret those special strings correctly, to display underlined or boldface text on the screen. (This only works, of course, if your terminal can display underlined and boldface, and if the TERMCAP entry says how to do it.)

columns

This is a "read only" option. You cannot change its value, but you can have *elvis* print it. It shows how wide your screen is.

directory

Elvis uses temporary files to store changed text. This option allows you to control where those temporary files will be. Ideally, you should store them on in fast non-volatile memory, such as a hard disk.

This option can only be set in the ".exrc" file.

errorbells

Normally, *elvis* will ring your terminal's bell if you make an error. However, in noerrorbells mode, your terminal will remain silent.

exrefresh

The EX mode of *elvis* writes many lines to the screen. You can make *elvis* either write each line to the screen separately, or save up many lines and write them all at once.

The exrefresh option is normally on, so each line is written to the screen separately.

You may wish to turn the exrefresh option off (:se noer) if the "write" system call is costly on your machine, or if you're using a windowing environment. (Windowing environments scroll text a lot faster when you write many lines at once.)

This option has no effect in *vi* mode.

ignorecase

Normally, when *elvis* searches for text, it treats uppercase letters as being different for lowercase letters.

When the ignorecase option is on, uppercase and lowercase are treated as equal.

keytime

The arrow keys of most terminals send a multi-character sequence. It takes a measurable amount of time for these sequences to be transmitted. The keytime option allows you to control the maximum amount of time to allow for an arrow key (or other mapped key) to be received in full.

The default keytime value is 2. Because of the way UNIX timekeeping works, the actual amount of time allowed will vary slightly, but it will always be between 1 and 2 seconds.

If you set keytime to 1, then the actual amount of time allowed will be between

0 and 1 second. This will generally make the keyboard's response be a little faster (mostly for the ESC key), but on those occasions where the time allowed happens to be closer to 0 than 1 second, *elvis* may fail to allow enough time for an arrow key's sequence to be received fully. Ugh.

As a special case, you can set keytime to 0 to disable this time limit stuff altogether. The big problem here is: If your arrow keys' sequences start with an ESC, then every time you hit your ESC key *elvis* will wait... and wait... to see if maybe that ESC was part of an arrow key's sequence.

NOTE: this option is a generalization of the timeout option of the real vi.

keywordprg

Elvis has a special keyword lookup feature. You move the cursor onto a word, and hit shift-K, and *elvis* uses another program to look up the word and display information about it.

This option says which program gets run. It should contain the full pathname of the program; your whole execution path is *not* checked.

The default value of this option is */usr/bin/ref*, which is a program that looks up the definition of a function in C. It looks up the function name in a file called "refs" which is created by ctags.

You can substitute other programs, such as an English dictionary program or the online manual. *elvis* runs the program, using the keyword as its only argument. The program should write information to stdout. The program's exit status should be 0, unless you want *elvis* to print "<<< failed >>>".

lines

This "read only" option shows how many lines you screen has.

list

Normally (in "nolist" mode) *elvis* will expand tabs to the proper number of spaces on the screen, so that the file appears the same would it would be if you printed it or looked at it with *more*.

Sometimes, though, it can be handy to have the tabs displayed as "^I". In "list" mode, *elvis* does this, and also displays a "$" after the end of the line.

magic

The search mechanism in *elvis* can accept "regular expressions"—strings in which certain characters have special meaning. The magic option is normally

on, which causes these characters to be treated specially. If you turn the magic option off (:se noma), then all characters except ^ and $ are treated literally. ^ and $ retain their special meanings regardless of the setting of magic.

paragraphs

The { and } commands move the cursor forward or backward in increments of one paragraph. Paragraphs may be separated by blank lines, or by a "dot" command of a text formatter. Different text formatters use different "dot" commands. This option allows you to configure *elvis* to work with your text formatter.

It is assumed that your formatter uses commands that start with a "." character at the front of a line, and then have a one- or two-character command name.

The value of the paragraphs option is a string in which each pair of characters is one possible form of your text formatter's paragraph command.

readonly

Normally, *elvis* will let you write back any file to which you have write permission. If you do not have write permission, then you can only write the changed version of the file to a *different* file.

If you set the readonly option, then *elvis* will pretend you do not have write permission to *any* file you edit. It is useful when you really only mean to use *elvis* to look at a file, not to change it. This way you cannot change it accidentally.

This option is normally off, unless you use the "view" alias of *elvis*. "View" is like *vi* except that the readonly option is on.

report

Commands in *elvis* may affect many lines. For commands that affect a lot of lines, *elvis* will output a message saying what was done and how many lines were affected. This option allows you to define what "a lot of lines" means. The default is 5, so any command which affects 5 or more lines will cause a message to be shown.

scroll

The CTRL-U and CTRL-D keys normally scroll backward or forward by half a screenful, but this is adjustable. The value of this option says how many lines those keys should scroll by.

sections

The [[and]] commands move the cursor backward or forward in increment of 1

section. Sections may be delimited by a { character in column 1 (which is useful for C source code) or by means of a text formatter's "dot" commands.

This option allows you to configure *elvis* to work with your text formatter's "section" command, in exactly the same way that the paragraphs option makes it work with the formatter's "paragraphs" command.

shell

When *elvis* forks a shell (perhaps for the :! or :shell commands) this is the program that is uses as a shell. This is */bin/sh* by default, unless you have set the SHELL environment variable, it which case the default value is copied from the environment.

shiftwidth

The < and > commands shift text left or right by some uniform number of columns. The shiftwidth option defines that uniform number. The default is 8.

sidescroll

For long lines, *elvis* scrolls sideways. (This is different from the real *vi*, which wraps a single long line onto several rows of the screen.) To minimize the number of scrolls needed, *elvis* moves the screen sideways by several characters at a time. The value of this option says how many characters' widths to scroll at a time. Generally, the faster your screen can be redrawn, the lower the value you will want in this option.

sync

If the system crashes during an edit session, then most of your work can be recovered from the temporary file that *elvis* uses to store changes. However, sometimes MINIX will not copy changes to the hard disk immediately, so recovery might not be possible. The [no]sync option lets you control this. In nosync mode (which is the default), *elvis* lets the operating system control when data is written to the disk. This is generally faster. In sync mode, *elvis* forces all changes out to disk every time you make a change. This is generally safer, but slower.

tabstop

Tab characters are normally 8 characters wide, but you can change their widths by means of this option.

term

This "read only" option shows the name of the termcap entry that *elvis* is using for your terminal.

vbell

If your termcap entry describes a visible alternative to ringing your terminal's bell, then this option will say whether the visible version gets used or not. Normally it will be.

If your termcap does NOT include a visible bell capability, then the vbell option will be off, and you cannot turn it on.

warn

Elvis will normally warn you if you run a shell command without saving your changed version of a file. The "nowarn" option prevents this warning.

wrapmargin

Normally (with wrapmargin=0) *elvis* will let you type in extremely long lines, if you wish. However, with wrapmargin set to something other that 0 (wrapmargin=65 is nice), *elvis* will automatically cause long lines to be "wrapped" on a word break for lines longer than wrapmargin's setting.

wrapscan

Normally, when you search for something, *elvis* will find it no matter where it is in the file. *elvis* starts at the cursor position, and searches forward. If *elvis* hits EOF without finding what you're looking for, then it wraps around to continue searching from line 1.

If you turn off the wrapscan option (:se nows), then when *elvis* hits EOF during a search, it will stop and say so.

9.6.6. Cflags

Elvis uses many preprocessor symbols to control compilation. Most of these flags allow you to disable small sets of features. MINIX-ST users will probably want all features enabled, but MINIX-PC users will have to disable one or two feature sets because otherwise *elvis* would be too large to compile and run.

These symbols can be defined via flags passed to the compiler. The best way to do this is to edit the Makefile, and append the flag to the "CFLAGS=" line. After you do that, you must recompile elvis completely by saying

```
make clean
make
```

-DM_SYSV

This flag causes *elvis* to use System-V ioctl() calls for controlling your terminal; normally it uses v7/BSD/MINIX ioctl() calls.

–DDATE

The symbol DATE should be defined to look like a string constant, giving the date when *elvis* was compiled. This date is reported by the ":version" command.

You can also leave DATE undefined, in which case ":version" will not report the compilation date.

–DCRUNCH

This flag causes several large often-used macros to be replaced by equivalent functions. This saves about 4K of space in the ".text" segment, and it does not cost you any features.

–DDEBUG

This adds many internal consistency checks and the ":debug" and ":validate" commands. It increases the size of "text" by about 5K bytes.

–DNO_CHARATTR

This permanenently disables the "charattr" option. It reduces the size of ".text" by about 850 bytes.

–DNO_RECYCLE

Normally, *elvis* will recycle space in the temporary file which contains totally obsolete text. The **–DNO_RECYCLE** option disables this, making your ".text" segment smaller by about 1K but also permitting the temporary file to grow very quickly. If you have less than two megabytes of free space on your disk, then do not even consider using this flag.

–DNO_SENTENCE

This leaves out the "(" and ")" visual commands, and removes the code that allows the "[[", "]]", "{", and "}" commands to recognize nroff macros. The "[[" and "]]" commands will still move to the start of the previous/next C function source code, though, and "{" and "}" will move to the previous/next blank line. This saves about 650 bytes from the ".text" segment.

–DNO_CHARSEARCH

This leaves out the visual commands which locate a given character in the current line: "f", "t", "F", "T", ";", and ",". This saves about 900 bytes.

–DNO_EXTENSIONS

This leaves out the ":mkexrc" command, and the "K" and "#" visual commands. Other extensions are either inherent in the design of *elvis*, or are too tiny to be worth removing. This saves about 500 bytes.

–DNO_MAGIC

This permanently disables the "magic" option, so that most meta-characters in a regular expression are not recognized. This saves about 3K bytes from the ".text" segment.

9.6.7. Termcap

Elvis can use standard termcap entries, but it also recognizes and uses several extra capabilities, if you give them. All of these are optional.

Capability	Description
:PU=:	sequence received from the <PgUp> key
:PD=:	sequence received from the <PgDn> key
:HM=:	sequence received from the <Home> key
:EN=:	sequence received from the <End> key
:VB=:	sequence sent to start bold printing
:Vb=:	sequence sent to end bold printing

9.6.8. Author

Elvis was written by Steve Kirkendall. He can be reached by email at: kirkenda@cs.pdx.edu, or ...!uunet!tektronix!psueea!eecs!kirkenda for comments regarding *elvis*.

9.7. IC—INTEGER CALCULATOR

Ic is a simple RPN (Reverse Polish Notation) calculator, used for small calculations and base conversions. All calculations are done using 32 bit integers. The standard input is usually a keyboard and the standard output requires a device with a "termcap" entry. The program starts by interpreting any <args> as commands, where the separation between arguments is considered to be the same as the ENTER key. For example,

```
ic 692 784+
```

After reading the arguments input is from the keyboard.

9.7.1. Stack Operations

The operation of this program is similar to an RPN calculator. A six level stack is used. The ENTER key pushes the stack up one level. For example, "12+5" is entered as "12 ENTER 5 +".

The top two entries on the stack are exchanged by the *x* command, and the stack

is rolled down one (popped) by the *p* key. The top of the stack may be cleared by pressing the back-space key. The whole stack and the registers are initialized by a *z*.

9.7.2. Numeric Entry

The input and output bases are initially decimal, but they may be changed using the *i* and *o* commands. The *i* command changes both bases, but the *o* command changes just the output base. These commands take a one character argument of *h*, *d*, *o* or *b* to change to Hexadecimal, Decimal, Octal or Binary. While the input base is hexadecimal the letters *a* through *f* are used to represent the decimal values 10 through 15.

When the input base is decimal: multiply, divide and remainder are signed, otherwise they are performed unsigned.

The output base may also be changed to ASCII (*a*), this causes the least significant 7 bits of a value to be displayed as a character. To input an ASCII value the translate (*t*) command may be used, it accepts one character as its argument.

9.7.3. Calculations

The arithmetic operations supported are: Negate ("."), Add ("+"), Subtract ("–"), Multiply ("*"), Divide ("/"), and Remainder ("%"). The logical (Boolean) operations available are: NOT (" "), AND ("&"), OR ("|"), and EXCLUSIVE-OR ("^").

After one of these operations the last top of stack value is saved. It may be restored by pressing *l* (L).

9.7.4. Saving Results

Ten temporary registers are available. The Store (*s*) command followed by a digit ("0".."9") will copy the top of the stack to the specified register. The Recall (*r*) command pushes the contents of a register onto the top of the stack.

If the Store command is followed by a "+" preceding the digit, then the top of the stack will be added to the specified "accumulator" register.

Values may also be written to a file. The *w* command writes the top of the stack, using the current output base, to a file called "pad" in the current directory. If the user does not have write access to the current directory then the file */tmp/pad_$USER* is used as the scratch pad. The scratch pad file is erased on the first use of the *w* command within each new invocation of "ic".

9.7.5. Miscellaneous

The Quit (*q*) key causes an immediate exit. The *m* command temporarily leaves *ic* by invoking the shell as a sub-process. For help while using *ic*, hit the *h* key. If an erroneous key is pressed the bell will sound.

9.7.6. Command Summary

Note that many commands have an alternative key-code available on the extended AT keyboard. This aids entry by including most commands on the right side of the keyboard.

ENTER	Enter (push up)
BS (DEL)	Clear top of stack
h	Help
i	Input base (h, d, o, b)
l (PGDN)	Last top of stack
m	MINIX shell
o	Output base (h, d, o, b, a)
p (DOWN)	Pop stack (roll down)
q (END)	Quit
r (LEFT)	Recall (0-9)
s (RIGHT)	Store [+] (0-9)
t	Translate (char)
w (PGUP)	Write top of stack to scratch pad
x (UP)	Exchange top of stack
z (HOME)	Zero all state
.	Change sign
+ (+)	Add
- (-)	Subtract
*	Multiply
/	Divide
% (sh/5)	Remainder
(tilde)	Not
&	And
|	Or
^	Exclusive-or

9.7.7. Author

Ic was written by Terrence W. Holm.

9.8. INDENT—INDENT AND FORMAT C PROGRAMS

Indent reads a C program in, rearranges the layout, and outputs a new C program that will compile to the same executable binary as the original one. The difference between the input and output is that the output is in a standard layout determined by a large number of options. For most of the options there are two choices, one that enables it and one that disables it.

If *indent* is called with no file files, it operates as a filter. If called with one file name, that file is reformatted and the result replaces the original file. A backup is created, however, with the suffix *.BAK*. If it is called with two file names, the first one is the input file and the second one is the output file. Only one file can be reformatted at a time (e.g., one cannot call *indent* with *.c as argument; this is an error and will not work.).

9.8.1. Options

Many options are available. If you want to format a program to the "official" MINIX format, use *pretty*, which calls *indent* with the proper options and then postprocesses the output. The options listed below control the formatting style.

OPTION: **–bad, –nbad**
If **–bad** is specified, a blank line is forced after every block of declarations. Default: **–nbad**.

OPTION: **–bap, –nbap**
If **–bap** is specified, a blank line is forced after every procedure body. Default: **–nbap**.

OPTION: **–bbb, –nbbb**
If **–bbb** is specified, a blank line is forced before every block comment. Default: **–nbbb**.

OPTION: **–bc, –nbc**
If **–bc** is specified, then a newline is forced after each comma in a declaration. **–nbc** turns off this option. The default is **–nbc**.

OPTION: **–bl, –br**
Specifying **–bl** lines up compound statements like this:

```
if (...)
{
   code
}
```

Specifying **–br** (the default) makes them look like this:

```
if (...) {
    code
}
```

OPTION: **–c***n*
The column in which comments on code start. The default is 33.

OPTION: **–cd***n*
The column in which comments on declarations start. The default is for these comments to start in the same column as those on code.

OPTION: **–cdb, –ncdb**
Enables (disables) the placement of comment delimiters on blank lines. With this option enabled, comments look like this:

```
/*
 * this is a comment
 */
```

Rather than like this:

```
/* this is a comment */
```

This only affects block comments, not comments to the right of code. The default is **–cdb**.

OPTION: **–ce, –nce**
Enables (disables) forcing "else"s to cuddle up to the immediately preceding "}". The default is **–ce**.

OPTION: **–ci***n*
Sets the continuation indent to be *n*. Continuation lines will be indented that far from the beginning of the first line of the statement. Parenthesized expressions have extra indentation added to indicate the nesting, unless **–lp** is in effect. **–ci** defaults to the same value as **–i**.

OPTION: **–cli***n*
Causes case labels to be indented *n* tab stops to the right of the containing switch statement. **–cli**0.5 causes case labels to be indented half a tab stop. The default is **–cli**0. (This is the only option that takes a fractional argument.)

OPTION: **–d***n*
Controls the placement of comments which are not to the right of code.

Specifying **–d1** means that such comments are placed one indentation level to the left of code. The default **–d0** lines up these comments with the code. See the section on comment indentation below.

OPTION: **–di***n*

Specifies the indentation, in character positions, from a declaration keyword to the following identifier. The default is **–di**16.

OPTION: **–dj, –ndj**

–dj left justifies declarations. **–ndj** indents declarations the same as code. The default is **–ndj**.

OPTION: **–ei, –nei**

Enables (disables) special else–**if** processing. If enabled, ifs following elses will have the same indentation as the preceding if statement. The default is **–ei**.

OPTION: **–fc1, –nfc1**

Enables (disables) the formatting of comments that start in column 1. Often, comments whose leading "/" is in column 1 have been carefully hand formatted by the programmer. In such cases, **–nfc1** should be used. The default is **–fc1**.

OPTION: **–i***n*

The number of spaces for one indentation level. The default is 8.

OPTION: **–ip, –nip**

Enables (disables) the indentation of parameter declarations from the left margin. The default is **–ip**.

OPTION: **–ln**

Maximum length of an output line. The default is 78.

OPTION: **–lp, –nlp**

Lines up code surrounded by parenthesis in continuation lines. If a line has a left paren which is not closed on that line, then continuation lines will be lined up to start at the character position just after the left paren.

OPTION: **–npro**

Causes the profile files, *.indent.pro* in both the current directory and the user's home directory to be ignored.

OPTION: **–pcs, –npcs**

If true (**–pcs**) all procedure calls will have a space inserted between the name and the "(". The default is **–npcs**.

OPTION: **–ps, –nps**
If true (**–ps**) the pointer following operator "–>" will be surrounded by spaces on either side. The default is **–nps**.

OPTION: **–psl, –npsl**
If true (**–psl**) the names of procedures being defined are placed in column 1 – their types, if any, will be left on the previous lines. The default is **–psl**.

OPTION: **–sc, –nsc**
Enables (disables) the placement of asterisks (*) at the left edge of all comments. The default is **–sc**.

OPTION: **–sob, –nsob**
If **–sob** is specified, *indent* will swallow optional blank lines. You can use this to get rid of blank lines after declarations. The default is **–nsob**.

OPTION: **–st**
Causes *indent* to take its input from *stdin*, and put its output to *stdout*.

OPTION: **–T***typename*
Adds typename to the list of type keywords. Names accumulate: **–T** can be specified more than once. You need to specify all the typenames that appear in your program that are defined by #typedefs. Nothing will be harmed if you miss a few, but the program will not be formatted as nicely as it should. This sounds like a painful thing to have to do, but it is really a symptom of a problem in C: typedef causes a syntactic change in the language and *indent* cannot find all typedefs.

OPTION: **–troff**
Causes *indent* to format the program for processing by *troff*. It will produce a fancy listing in much the same spirit as *vgrind*. If the output file is not specified, the default is standard output, rather than formatting in place.

OPTION: **–v, –nv**
The **–v** flag turns on verbose mode; **–nv** turns it off. When in verbose mode, *indent* reports when it splits one line of input into two or more lines of output, and gives some size statistics at completion. The default is **–nv**.

9.8.2. User Profiles

You may set up your own profile of defaults to *indent* by creating a file called *.indent.pro* in either your login directory and/or the current directory and including whatever switches you like. Switches in *.indent.pro* in the current directory override those in your login directory (with the exception of **–T** type definitions, which

just accumulate). If *indent* is run and a profile file exists, then it is read to set up the program's defaults. The switches should be separated by spaces, tabs or newlines. Switches on the command line, however, override profile switches.

9.8.3. Comments

Indent assumes that any comment with a dash or star immediately after the start of comment (that is, "/*–" or "/**") is a comment surrounded by a box of stars. Each line of such a comment is left unchanged, except that its indentation may be adjusted to account for the change in indentation of the first line of the comment.

All other comments are treated as straight text. Indent fits as many words (separated by blanks, tabs, or newlines) on a line as possible. Blank lines break paragraphs.

If a comment is on a line with code it is started in the comment column, which is set by the –cn command line parameter. Otherwise, the comment is started at *n* indentation levels less than where code is currently being placed, where n is specified by the –dn command line parameter. If the code on a line extends past the comment column, the comment starts further to the right, and the right margin may be automatically extended in extreme cases.

9.8.4. Preprocessor Lines

In general, *indent* leaves preprocessor lines alone. The only reformatting that it will do is to straighten up trailing comments. It leaves embedded comments alone. Conditional compilation (#ifdef...#endif) is recognized and *indent* attempts to correctly compensate for the syntactic peculiarities introduced.

9.8.5. C Syntax

Indent understands a substantial amount about the syntax of C, but it has a forgiving parser. It attempts to cope with the usual sorts of incomplete and misformed syntax. In particular, the use of macros like:

```
#define forever for(;;)
```

is handled properly.

9.9. KERMIT—A FILE TRANSFER PROGRAM

This is a slightly lobotomized *kermit*. The help command, the script facility, and the automatic dial support have been removed. The ? and ESC commands still work, so there is still reasonable built-in help. The only V7 *kermit* feature that does not work is the ability to see whether there are input characters waiting. This

means that you will not be able to ask for status during a file transfer (though this is not critical, because *kermit* prints a dot every so often and other special characters whenever there is an error or timeout).

To use *kermit* on an IBM PC, you must first set the line speed (because *kermit* cannot do this) although it cannot hurt to set it on the 68000 as well. To set it to 2400 baud, for example, type:

```
stty 2400 </dev/tty1
```

Now start *kermit*, and then type

```
set line /dev/tty1
set speed 2400
connect
```

(It is more convenient if you put these commands in *.kermrc* in your home directory, so that they get done automatically whenever you run *kermit*.) This will connect you to the modem or whatever on the serial port. Now log into the other system.

When you want to transfer files, run *kermit* on the other system. To it, type

```
server
```

This puts its *kermit* into a sort of "slave mode" where it expects commands from the *kermit* running on your MINIX system. Now come back to the command level on MINIX *kermit*, by typing the escape character followed by *c*. (*Kermit* will tell you the current escape character when you do the connect command.) At this point you can issue various commands. Your *kermit* will coordinate things with *kermit* on the other machine so that you only have to type commands at one end. Common commands are

get *filename*
put *filename*
remote *dir*

Filenames can include wildcards. By default, *kermit* works in a system-independent, text mode. (In effect it assumes that the whole world is MS-DOS and converts end of line and file names accordingly.) To send binary files, you will want to type

```
set file type bin
```

on both ends before starting any transfers. This disables CR LF to newline conversion. If both of your systems are some flavor of UNIX, you might as well put this in *.kermrc* on both ends and run in binary mode all the time. Also, if both systems are UNIX it is recommended that you use

```
set file name lit
```

on both ends. This causes it to keep file names unchanged, rather than mapping to legal MS-DOS names.

Here is a typical *.kermrc* for use on MINIX:

```
set line /dev/tty1
set speed 1200
set esc 29
set file type bin
set file name lit
set retry 90
set prompt MINIX kermit>
connect
```

On the other end of the line, for example, the host at your local computer center to which you want to transfer files, a typical profile might be:

```
set rec packet 1000
set fil name lit
set fil type bin
server
```

On the IBM PC, It is not possible to recompile *kermit* on MINIX because it is so large that the assembler runs out of memory. However, you may be able to recompile it on MS-DOS using one of the C compilers there. You will have to convert the binary to MINIX format, however.

Kermit has many other options and features. For a pleasant and highly readable description of it, see the following book:

Title: Kermit: A File Transfer Protocol
Author: Frank da Cruz
Publisher: Digital Press
Date: 1987
ISBN: 0-932376-88

For information about recent *kermit* developments, versions for other systems, and so forth, please contact:

Christine M. Gianone
Manager, Kermit Development and Distribution
University Center for Computing Activities
Columbia University
612 West 115th Street
New York, N.Y. 10025

Over 400 versions of *kermit* are available, so it is likely there is one for any computer your MINIX system might want to talk to. Columbia University also publishes a newsletter about *kermit* that can be requested from the above address.

9.10. M4—MACRO PROCESSOR

M4 is a macro processor intended as a front end for Ratfor, Pascal, and other languages that do not have a built-in macro processing capability. M4 reads standard input, the processed text is written on the standard output.

The options and their effects are as follows:

–D name[=val]	Defines name to val, or to null in val's absence.
–U name	Undefines name.

Macro calls have the form: name(arg1,arg2, ..., argn)

The "(" must immediately follow the name of the macro. If the name of a defined macro is not followed by a (it is taken to be a call of that macro with no arguments, i.e. name(). Potential macro names consist of alphabetic letters and digits.

Leading unquoted blanks, tabs and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching). If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. (This is typically referred as inside-out macro expansion.) After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define "(name [, val])" the second argument is installed as the value of the macro whose name is the first argument. If there is no second argument, the value is null. Each occurrence of $ n in the replacement text, where n is a digit, is replaced by the n -th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string.

defn "(name [, name ...])" returns the quoted definition of its argument(s). Useful in renaming macros.

undefine "(name [, name ...])" removes the definition of the macro(s) named. If there is more than one definition for the named macro, (due to previous use of pushdef) all definitions are removed.

pushdef "(name [, val])" like define, but saves any previous definition by stacking the current definition.

popdef "(name [, name ...])" removes current definition of its argument(s), exposing the previous one if any.

ifdef "(name, if-def [, ifnot-def])" if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. A word indicating the current operating system is predefined. (e.g. unix or vms).

shift "(arg, arg, arg, ...)" returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote "(lqchar, rqchar)" change quote symbols to the first and second arguments. With no arguments, the quotes are reset back to the default characters. (i.e., ‘’).

changecom "(lcchar, rcchar)" change left and right comment markers from the default # and newline. With no arguments, the comment mechanism is reset back to the default characters. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected.

divert "(divnum)" maintains 10 output streams, numbered 0-9. Initially stream 0 is the current stream. The divert macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is lost.

undivert "([divnum [, divnum ...]])" causes immediate output of text from diversions named as argument(s), or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text. At the end of input processing, M4 forces an automatic undivert unless is defined.

divnum "()" returns the value of the current output stream.

dnl "()" reads and discards characters up to and including the next newline.

ifelse "(arg, arg, if-same [, ifnot-same | arg, arg ...])" has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr "(num)" returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr "(num)" returns the value of its argument decremented by 1.

eval "(expression)" evaluates its argument as a constant expression, using integer arithmetic. The evaluation mechanism is very similar to that of cpp (#if expression). The expression can involve only integer constants and character constants, possibly connected by the binary operators

```
*  /  %  +  -  >>  <<  <  >  <=  >=  ==  !=  &  ^  |  &&  ||
```

or the unary operators - ! or tilde or by the ternary operator ? : . Parentheses may be used for grouping. Octal numbers may be specified as in C.

len "(string)" returns the number of characters in its argument.

index "(search-string, string)" returns the position in its first argument where the second argument begins (zero origin), or 1 if the second argument does not occur.

substr "(string, index [, length])" returns a substring of its first argument. The second argument is a zero origin number selecting the first character (internally treated as an expression); the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit "(source, from [, to])" transliterates the characters in its first argument from the set given by the second argument to the set given by the third. If the third argument is shorter than the second, all extra characters in the second argument are deleted from the first argument. If the third argument is missing altogether, all characters in the second argument are deleted from the first argument.

include "(filename)" returns the contents of the file that is named in the argument.

sinclude "(filename)" is identical to include, except that it says nothing if the file is inaccessable.

paste "(filename)" returns the contents of the file named in the argument without any processing, unlike include.

spaste "(filename)" is identical to paste, except that it says nothing if the file is inaccessibl[De.

syscmd "(command)" executes the UNIX command given in the first argument. No value is returned.

sysval "()" is the return code from the last call to syscmd.

.PP **maketemp "(string)"** fills in a string of XXXXXX in its argument with the current process ID.

m4exit "([exitcode])" causes immediate exit from M4. Argument 1, if given, is the exit code; the default is 0.

m4wrap "(m4-macro-or-built-n)" argument 1 will be pushed back at final EOF; example: m4wrap('dumptable()').

errprint "(str [, str, str, ...])" prints its argument(s) on stderr. If there is more than one argument, each argument is separated by a space during the output. An arbitrary number of arguments may be supplied.

dumpdef "([name, name, ...])" prints current names and definitions, for the named items, or for all if no arguments are given.

9.10.1. Author

M4 was written by Ozan S. Yigif.

9.11. MDB—MINIX DEBUGGER [68000]

Mdb provides a means of debugging MINIX programs on the 68000. No IBM debugger is available at present. *Mdb* supports symbolic debugging. The argument to *mdb* is a MINIX executable file. This file defaults to *a.out*.

Once started, *mdb* will display an asterisk (*) as command prompt. A command may be entered in the following form:

[<expression>][<command> [<argument>]]"

<expression> is a symbolic expression representing a location in memory or a register, and should be in the form:

[<value> [+|– <value>]]

A <value> may be a symbol, a register, or a constant. The symbol may be any external symbol found in the executable file, or *start* if the executable file is stripped. *_start* represents the beginning address of the program. A register is specified by a "$" followed by the register's name, while a constant may be either an octal, decimal or hex unsigned long integer expressed using standard C notation (no "L" suffix should be used). A character constant is represented by a single quote followed by the character.

Not all commands use an address, but those that do have a default value of the current program counter with the exception of the continue command: "c" and "C". Not all commands use an argument list, but those that do default to an empty list. If the argument contains a semi-colon (";"), then except for the breakpoint command ("b") the string following the semi-colon is assumed to be a new command.

Following is the list of valid commands. When *mdb* is first started, no process is active. Most of the following commands require an active process which may be created using the run ("r") command.

!

If the exclamation point begins the command line, then the MINIX program that follows it is executed. The default command is */bin/sh*. Note: full path names are required.

Example: * !vi *mdb*.c

If the expression preceding the exclamation point is a valid data address, then the arguments specify how to modify its contents. The argument is to be in the form:

[<constant>] [<size>] [<expression>]

and are interpreted as: "fill the data space starting at the specified address with

<constant> values given by <expression> and are of size byte (b), half word (h, i.e. short), or long word (l)." The default value of constant is 1, of size is "h", and of expression is 0.

Example: * _buf ! 20b "a"

T

This command prints the current active function in the program with its arguments.

Example: * T

t

This command prints the current function invocation hierarchy with their arguments.

Example: * t

/

This command prints the values starting at the address specified.
The argument is to be in the form:

[<constant>][<size>][<format>]

format may have the values:

a - Displays chars until a zero is found
c - Displays <constant> values as characters.
d - Displays <constant> values as signed decimal numbers.
i - Disassembles <constant> instructions (<size> ignored).
I - Disassembles <constant> instructions (<size> ignored).
o - Displays <constant> vlaues as octal numbers.
s - Displays string at the location specified
u - Displays <constant> vlaues as unsigned decimal numbers.
x - Displays <constant> vlaues as hexadecimal numbers.

Example: * $a6-2/hx

x

This command prints the registers and the instructions starting at the specified address. An optional constant argument limits the number of instructions printed to the value given by the final argument.

Example: * x

X

This command prints the instructions starting at the specified address. An

optional constant argument limits the number of instructions printed to the value given by the final argument.

Example: * _start X 10

R

This command starts a process using the executable given as *mdb*'s first argument. The process will have no arguments and will be stopped prior to executing any instructions.

Example: * R

r

This command starts a process using the executable given as *mdb*'s first argument. The process will be given the arguments on the command line up to the first semi-colon. If no arguments are specified, then the arguments supplied with the last "r" command are used. The processes' standard input and output may be redirected. The process will be stopped prior to executing any instructions.

Example: * r 3 <input >output

C

This command results in the stopped process being restarted with a pending signal specified by the constant argument. A breakpoint is placed at the address specified if one is not already there, and if placed it is deleted when the process stops for any reason. No default address is assumed.

Example: * C 2

c

This command results in the stopped process being restarted with all pending signals canceled. A breakpoint is placed at the address specified if one is not already there, and if placed it is deleted when the process stops for any reason. No default address is assumed.

Example: * _func+4 c

I

This command results in the stopped process executing instructions in single step mode. The number of instructions executed is given in the constant argument. The signal that stopped the process will be sent when execution begins.

Example: * I 10

i

This command results in the stopped process executing instructions in single

step mode. The number of instructions executed is given in the constant argument. All pending signals are canceled before execution begins.

Example: * i 10

M

This command results in the stopped process resuming execution until the long word at the location specified by the address is modified or until the number of instructions specified in the optional constant argument are executed. The default number is 65,536. Each executed instruction is displayed prior to execution.

Example: * _var M 100

m

This command results in the stopped process resuming execution until the long word at the location specified by the address is modified or until the number of instructions specified in the obtional constant argument are executed. The default number is 65,536.

Example: * _var m

k

This command results in the current active process being terminated.

Example: * k

B

This command results in all the currently active breakpoints being listed.

Example: * B

b

This command results in a breakpoint being placed at the location specified by the address in program space. The string that follows the command contains the command(s) that are executed when the breakpoint is hit. It is recommended that breakpoints be placed at an offset of 4 bytes from the function entry point to permit a valid frame to be set up for back-tracing.

Example: * _func+4 b t;_var/lx

d

This command results in the breakpoint at the address specified being deleted.

Example: * _func+4 d

D

This command results in all breakpoints being deleted.

Example: * D

q

This command results in the currently active process being terminated and *mdb* exiting.

Example: * q

9.11.1. Author

Mdb was written by Bruce D. Szablak

9.12. MINED—A SIMPLE SCREEN EDITOR

Mined is a simple screen editor. At any instant, a window of 24 lines is visible on the screen. The current position in the file is shown by the cursor. Ordinary characters typed in are inserted at the cursor. Control characters and keys on the numeric keypad (at the right-hand side of the keyboard) are used to move the cursor and perform other functions.

Commands exist to move forward and backward a word, and delete words either in front of the cursor or behind it. A word in this context is a sequence of characters delimited on both ends by white space (space, tab, line feed, start of file, or end of file). The commands for deleting characters and words also work on line feeds, making it possible to join two consecutive lines by deleting the line feed between them.

The editor maintains one save buffer (not displayed). Commands are present to move text from the file to the buffer, from the buffer to the file, and to write the buffer onto a new file. If the edited text cannot be written out due to a full disk, it may still be possible to copy the whole text to the save buffer and then write it to a different file on a different disk with CTRL-Q. It may also be possible to escape from the editor with CTRL-S and remove some files.

Some of the commands prompt for arguments (file names, search patterns, etc.). All commands that might result in loss of the file being edited prompt to ask for confirmation.

A key (command or ordinary character) can be repeated *n* times by typing *ESC n key* where *ESC* is the "escape" key.

Forward and backward searching requires a regular expression as the search

pattern. Regular expressions follow the same rules as in the UNIX editor, *ed*. These rules can be stated as:

1. Any displayable character matches itself.
2. . (period) matches any character except line feed.
3. ^ (circumflex) matches the start of the line.
4. $ (dollar sign) matches the end of the line.
5. \c matches the character *c* (including period, circumflex, etc).
6. [*string*] matches any of the characters in the string.
7. [^string] matches any of the characters except those in the string.
8. [*x*–*y*] matches any characters between *x* and *y* (e.g., [*a*–*z*]).
9. Pattern* matches any number of occurrences of *pattern*.

Some examples of regular expressions are:

The boy	matches the string "The boy"
^$	matches any empty line.
^.$	matches any line containing exactly 1 character
^A.*\.$	matches any line starting with an *A*, ending with a period.
^[A–Z]*$	matches any line containing only capital letters (or empty).
[A–Z0–9]	matches any line containing either a capital letter or a digit.
.*X	matches any line ending in "X"
A.*B	matches any line containing an "A" and then a "B"

Control characters cannot be entered into a file simply by typing them because all of them are editor commands. To enter a control character, depress the ALT key, and then while holding it down, hit the ESC key. Release both ALT and ESC and type the control character. Control characters are displayed in reverse video.

The *mined* commands are as follows.

CURSOR MOTION

arrows	Move the cursor in the indicated direction
CTRL-A	Move cursor to start of current line
CTRL-Z	Move cursor to end of current line
CTRL-^	Move cursor to top of screen
CTRL-_	Move cursor to end of screen
CTRL-F	Move cursor forward to start of next word
CTRL-B	Move cursor backward to start of previous word

SCREEN MOTION

Home key	Move to first character of the file
End key	Move to last character of the file
PgUp key	Scroll window up 23 lines (closer to start of the file)
PgDn key	Scroll window down 23 lines (closer to end of the file)
CTRL-U	Scroll window up 1 line
CTRL-D	Scroll window down 1 line

MODIFYING TEXT

Del key	Delete the character under the cursor
Backspace	Delete the character to left of the cursor
CTRL-N	Delete the next word
CTRL-P	Delete the previous word
CTRL-T	Delete tail of line (all characters from cursor to end of line)
CTRL-O	Open up the line (insert line feed and back up)
CTRL-G	Get and insert a file at the cursor position

BUFFER OPERATIONS

CTRL-@	Set mark at current position for use with CTRL-C and CTRL-K
CTRL-C	Copy the text between the mark and the cursor into the buffer
CTRL-K	Delete text between mark and cursor; also copy it to the buffer
CTRL-Y	Yank contents of the buffer out and insert it at the cursor
CTRL-Q	Write the contents of the buffer onto a file

MISCELLANEOUS

numeric +	Search forward (prompts for regular expression)
numeric –	Search backward (prompts for regular expression)
numeric 5	Display the file status
CTRL-]	Go to specific line
CTRL-R	Global replace *pattern* with *string* (from cursor to end)
CTRL-L	Line replace *pattern* with *string*
CTRL-W	Write the edited file back to the disk
CTRL-X	Exit the editor
CTRL-S	Fork off a shell (use CTRL-D to get back to the editor)
CTRL-	Abort whatever the editor was doing and wait for command
CTRL-E	Erase screen and redraw it
CTRL-V	Visit (edit) a new file

The key bindings on the Atari ST are different. The table below summarizes the *mined* commands with the corresponding ST keys, and the PC keys if they differ.

CURSOR MOTION	**ST key**	**PC key**
up,down,left,right	arrows	
start of line	CTRL-A	
end of line	CTRL-Z	
top of screen	CTRL-^	
end of screen	CTRL-_	
next word	CTRL-F	
previous word	CTRL-B	

SCREEN MOTION	**ST key**	**PC key**
first char of file	Home	
last char of file	F6	End
scroll window up	F4	PgUp
scroll window down	F3	PgDn
scroll line up	CTRL-U	
scroll line down	CTRL-D	

MODIFYING TEXT	**ST key**	**PC key**
delete this char	Delete	
delete previous char	Backspace	
delete next word	CTRL-N	
delete previous word	CTRL-P	
delete tail of line	CTRL-T	
open up line	CTRL-O	
get file at cursor	CTRL-G	

MISCELLANEOUS	**ST key**	**PC key**
search forward	F1	numeric +
search backward	F2	numeric –
file status	F5	numeric 5
repeat	Esc	
goto line	CTRL-]	
global replace	CTRL-R	
line replace	CTRL-L	
write file	CTRL-W	
exit	CTRL-X	
fork shell	CTRL-S	
abort	CTRL-\	
redraw	CTRL-E	
new file	CTRL-V	
escape next char	F8	ALT-ESC

BUFFER OPERATIONS	ST key	PC key
set mark	F7	CTRL-@
copy to buffer	CTRL-C	
delete to buffer	CTRL-K	
insert buffer	CTRL-Y	
write buffer to file	CTRL-Q	

9.12.1. Author

Mined was designed by Andy Tanenbaum and written by Michiel Huisjes.

9.13. NROFF—A TEXT PROCESSOR

Nroff is a text processor and formatter based on the design provided in *Software Tools* by Kernighan and Plauger. It has been modified to resemble the UNIX *nroff* command. The text and commands found in the file(s) are processed to generate formatted text. Note that one (and only one) of the files can be – which reads input from stdin at that point. The output always goes to stdout which can be redirected by the shell. The **–o** option lets you redirect error output to the specified file rather than stderr.

The following command line options are available:

–m*name*	Process macro file *tmac.name*.
–o*file*	Set error log file (default is *stderr*).
–po*n*	Shift output right n spaces (like .po).
–pn*n*	Initial page number (like .pn).
–v	Prints the version information to *stdout*.
+*n*	Causes output to start with page *n*.
–*n*	Causes output to stop after page *n*.
–	Input from stdin.

Nroff recognizes the following environment variables from the shell. *TMACDIR* is alternate directory to find the files *tmac*.* (“.” for example). The default is */usr/lib/tmac*. *TMPDIR* is an alternate directory to place any temporary files. The default is the current directory.

9.13.1. Commands

Commands typically are distinguished by a period in column one of the input followed by a two character abbreviation for the command function. The abbreviation may then be followed by an optional numeric or character argument. The numeric argument may be an absolute value such as setting the right margin to a

particular column, or the argument may be preceded by a plus sign or a minus sign to indicate that the parameter should be modified relative to a previous setting. The following commands are recognized (those marked "extension" are requests that may be added some day in the distant future).

.ad

Begin line adjustment. If fill mode is not on, adjustment is deferred until it is back on. If a type indicator is present, the adjustment type is changed as follows:

Indicator	Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins (default)
absent	unchanged

.af

Assign format to number register. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,...
001	000,001,002,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,...,z,aa,ab,...zz,aaa,...
A	0,A,B,...,Z,AA,AB,...ZZ,AAA,...

The second format above indicates that the field width, i.e. number of digits, is specified by the number of digits in the format type.

.bd

Ignored by nroff.

.bo (extension)

Causes the following lines of text to appear in boldface. The optional argument specifies the number of lines to be typed in boldface. Boldface and underlining are mutually exclusive features. The appearance of a boldface command will cause any underlining to cease.

.bp (extension)

Causes succeeding text to appear at the top of a new page. The optional argument specifies the page number for the new page. The initial value is one and the default value is one more than the previous page number.

.br

Causes succeeding text to start on a new line at the current left margin. There is no numeric argument for this command.

.bs (extension)

Enables or disables the appearance of backspaces in the output text. Underlining and boldface options are implemented by inserting character-backspace-character combinations into the output buffer. This is fine for devices which properly recognize the backspace character. Some printers, however, do not recognize backspaces, so the option is provided to overprint one line buffer with another. The first line buffer is terminated with just a carriage return rather than the carriage return-linefeed combination. A zero argument or no argument to the backspace command removes backspaces from the output. A non-zero argument leaves them in the output. The default is to remove backspaces.

.cc

Changes the *nroff* command character to that specified by the character argument. If no argument is provided, the default is a period (.).

.ce

Causes the next line of text to appear centered on the output. The optional argument specifies if more than one line is to be centered.

.cs

Ignored by *nroff*.

.cu

Causes the next line(s) of text to be continuously underlined. Unlike the underline command (see .ul) which underlines only alphanumerics, continuous underlining underlines all printable characters. The optional argument specifies the number of lines of text to underlined. Any normal underlining or boldface commands currently in effect will be terminated.

.c2

Changes the *nroff* no break character to that specified by the character argument. If no argument is provided, the default is a single quote.

.de

Causes all text and commands following to be used to define a macro. The definition is terminated by a .en command or the default .. terminator. The first two characters of the argument following the .de command become the name of the new command. It should be noted that upper and lower case arguments are considered different. Thus, the commands .PP and .pp could define two

different macros. Care should be exercised since existing commands may be redefined.

A macro may contain up to ten arguments. In the macro definition, the placement of arguments is designated by the two character sequences, $1, $2, ... $9. When the macro is invoked, each argument of the macro command line is substituted for its corresponding designator in the expansion. The first argument of the macro command is substituted for the $1 in the expansion, the second argument for the $2, and so forth. Arguments are typically strings which do not contain blanks or tabs. If an argument is to contain blanks, then it should be surrounded by either single or double quotes.

.ds

Define a string. To initiate the string with a blank or include blanks in the string, start it with a single or double quite. The string can contain other defined strings or number registers as well as normal text. Strings are stored on the macro name space.

.ec

Changes the *nroff* escape character to that specified by the character argument. If no argument is provided, the default is a backslash.

.ef (extension)

Specifies the text for the footer on even numbered pages. The format is the same as for the footer command (see .fo).

.eh (extension)

Specifies the text for the header on even numbered pages. The format is the same as for the footer command (see .fo).

.en (extension)

Designates the end of a macro definition.

.eo

Turn the escape mechanism off.

.fi

Causes the input text to be rearranged or filled to obtain the maximum word count possible between the previously set left and right margins. No argument is expected.

.fl

Causes the output buffer to be flushed immediately.

.fo (extension)

Specifies text to be used for a footer. The footer text contains three strings separated by a delimiter character. The first non-blank character following the command is designated as the delimiter. The first text string is left justified to the current indentation value (specified by .in). The second string is centered between the current indentation value and the current right margin value (specified by .rm). The third string is right justified to the current right margin value. The absence of footer text will result in the footer being printed as one blank line. The presence of the page number character (set by .pc) in the footer text results in the current page number being inserted at that position. Multiple occurrences of the page number character are allowed.

.ft

Changes the current font. The choices are R (Times Roman), I (Times Italic), B (Times Bold), S (math special), and P used to request the previous font. P resets the next previous font to be the one just changed, amounting to a swap.

.he (extension)

Specifies text to be used for a header. The format is the same as for the footer (see .fo).

.in

Indents the left margin to the column value specified by the argument. The default left margin is set to zero.

.ju (extension)

Causes blanks to be inserted between words in a line of output in order to align or justify the right margin. The default is to justify.

.ll

Sets the current line length. The default is eighty.

.ls

Sets the line spacing to the value specified by the argument. The default is for single spacing.

.lt

Set length of three-part titles. Line length and title length are independent. Indents do not apply to titles but page offsets do.

.m1 (extension)

Specifies the number of lines in the header margin. This is the space from the physical top of page to and including the header text. A value of zero causes the

header to not be printed. A value of one causes the header to appear at the physical top of page. Larger argument values cause the appropriate number of blank lines to appear before the header is printed.

.m2 (extension)
Specifies the number of blank lines to be printed between the header line and the first line of the processed text.

.m3 (extension)
Specifies the number of blank lines to be printed between the last line of processed text and the footer line.

.m4 (extension)
Specifies the number of lines in the footer margin. This command affects the footer the same way the .m1 command affects the header.

.na
Noadjust. Adjustment is turned off; the right margin is ragged. The adjustment type for .ad is not changed. Output line filling still occurs if fill mode is on.

.ne
Specifies a number of lines which should not be broken across a page boundary. If the number of lines remaining on a page is less than the value needed, then a new output page is started.

.nf
Specifies that succeeding text should be printed without rearrangement, or with no fill. No argument is expected.

.nj (extension)
Specifies that no attempt should be made to align or justify the right margin. No argument is expected.

.nr
Causes the value of a number register to be set or modified. A total of twenty-six number registers are available designated \na through \nz (either upper or lower case is allowed). When the sequence \nc is imbedded in the text, the current value of number register c replaces the sequence, thus, such things as paragraph numbering can be accomplished with relative ease.

.of (extension)
Specifies the text for the footer on odd numbered pages. The format is the same as the footer command (see .fo).

.oh (extension)

Specifies the text for the header on odd numbered pages. The format is the same as the footer command (see .fo).

.pc

Specifies the page number character to be used in headers and footers. The occurrence of this character in the header or footer text results in the current page number being printed. The default for this character is the percent sign (%).

.pl

Specifies the page length or the number of lines per output page. The default is sixty-six.

.pm

Print macros. The names and sizes of the macros are printed to stdout. This is useful when building a macro package to see how much of the total namespace is consumed by the package.

.pn

Changes the page number of the current page and all subsequent pages to its argument. If no argument is given, the command is ignored.

.po

Specifies a page offset value. This allows the formatted text to be shifted to the right by the number of spaces specified. This feature may also be invoked by a switch on the command line.

.ps

Ignored by nroff.

.rr

Removes a number register.

.so

Causes input to be retrieved from the file specified by the command's character string argument. The contents of the new file are inserted into the output stream until an EOF is detected. Processing of the original file is then resumed. Command nesting is allowed.

.sp

Specifies a number of blank lines to be output before printing the next line of text.

.ti

Temporarily alters the indentation or left margin value for a single succeeding input line.

.tl

Specifies text to be used for a page title. The format is the same as for the header (see .he).

.ul

Causes the next line(s) of text to be underlined. Unlike the .cu command, this command causes only alphanumerics to be underlined, skipping punctuation and white space. Underline and boldface are mutually exclusive.

The following *nroff* commands, normally available, are currently not implemented in this version:

.fp, .mk, .rt, .vs, .sv, .os, .ns, .rs, .am, .as, .rm, .rn, .di, .da, .wh, .ch, .dt, .it, .em, .ta, .tc, .lc, .fc, .lg, .uf, .tr, .nh, .hy, .hc, .hw, .nm, .nn, .if, .ie, .el, .ev, .rd, .ex, .nx, .pi, .mc, .tm, and .ig.

9.13.2. Escape Sequences

Escape sequences are used to access special characters (such as Greek letters) which may be outside the normal printable ASCII character set. The are also used to toggle certain actions such as font selection. The escape sequences include:

\ backslash character
\e printable version of escape character
\' acute accent (equivalent to \(aa)
\` grave accent (equivalent to \(ga)
\- minus sign
\. period
\<sp> a single, unpaddable space
\0 digit-width space
\& non-printing zero-width character
\" beginning of comment
\% default hyphenation character
\(xx special character named xx
*x insert string named *x*
*(xx insert string named *xx*
\fc font change (c = R,I,B,S,P)
\nx interpolate number register x
\t horizontal tab

9.13.3. Predefined General Number Registers

The following number registers are available for both reading and writing. They are accessed with the \n(xx and \nx escape and can be set with .nr:

%	current page number
dw	current day of the week (1-7)
dy	current day of the month (1-31)
hh	current hours (0-23)
ln	current line number
mm	current minutes (0-59)
mo	current month (1-12)
ss	current seconds (0-59)
yr	last 2 digits of current year

The following number registers are available for reading only:

.$	number of args available in current macro
.A	always 1 in nroff
.H	available horizontal resolution
.T	always 0 in nroff
.V	available vertical resolution
.c	number of lines read from current file
.f	current font (1-4)
.i	current indent
.l	current line length
.o	current page offset
.p	current page length
.v	current vertical spacing

9.13.4. Notes

There are several missing features, notably diversions, traps, and conditionals. This means you cannot use some existing macro packages. There are no –ms and –me packages as a result. The goal is to (eventually) make *nroff* work with all the SunOS macro packages.

9.13.5. Authors

This version of *nroff* was originally written in BDS C by Stephen L. Browning. It was adapted for standard C by W. N. Paul. Bill Rosenkranz modified it heavily and ported it to MINIX.

9.14. PATCH—A PROGRAM FOR APPLYING DIFF LISTINGS

The MINIX user community on USENET frequently makes improvements to the MINIX software. The changes are distributed in the form of differences between the original file and the new one, made with *cdiff*. To update the original version (which you must have), use *patch*. If the original file is called *prog.c* and the patch is called *prog.cdif* then you should type:

```
patch prog.c prog.cdif
```

In some cases, a large number of files in a single directory will be updated at once. In this case, the difference file may be the concatenation of many individual difference files. The resulting file usually has a name like *dir.cdif*. To apply all the patches, type:

```
patch <dir.cdif
```

Patch will take a patch file containing any of the three forms of difference listing produced by the diff program and apply those differences to an original file, producing a patched version. By default, the patched version is put in place of the original, with the original file backed up to the same name with a tilde appended, or as specified by the **–b** flag. You may also specify where you want the output to go with a **–o** flag. If patchfile is omitted, or is a hyphen, the patch will be read from standard input.

Upon startup, patch will attempt to determine the type of the diff listing, unless over-ruled by a **–c**, **–e**, or **–n** flag. Context diffs and normal diffs are applied by the patch program itself, while *ed* diffs are simply fed to the *ed* editor via a pipe.

Patch will try to skip any leading garbage, apply the diff, and then skip any trailing garbage. Thus you could feed an article or message containing a diff listing to patch and it should work. If the entire diff is indented by a consistent amount, this will be taken into account.

With context diffs, and to a lesser extent with normal diffs, patch can detect when the line numbers mentioned in the patch are incorrect, and will attempt to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned for the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, patch will scan both forwards and backwards for a set of lines matching the context given in the hunk. First patch looks for a place where all lines of the context match. If no such place is found, and it is a context diff, and the maximum fuzz factor is set to 1 or more, then another scan takes place ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, the first two and last two lines of context are ignored, and another scan is made. (The default maximum fuzz factor is 2.) If patch cannot find a place to install that hunk of the patch, it will put the hunk out to a reject file, which normally is the name of the output file plus "#". (Note that the rejected hunk will come out in context diff form whether the input patch was a context diff

or a normal diff. If the input was a normal diff, many of the contexts will simply be null.) The line numbers on the hunks in the reject file may be different than in the patch file: they reflect the approximate location patch thinks the failed hunks belong in the new file rather than the old one.

As each hunk is completed, you will be told whether the hunk succeeded or failed, and which line (in the new file) patch thought the hunk should go on. If this is different from the line number specified in the diff you will be told the offset. A single large offset MAY be an indication that a hunk was installed in the wrong place. You will also be told if a fuzz factor was used to make the match, in which case you should also be slightly suspicious.

If no original file is specified on the command line, patch will try to figure out from the leading garbage what the name of the file to edit is. In the header of a context diff, the filename is found from lines beginning with "***" or "---", with the shortest name of an existing file winning. Only context diffs have lines like that, but if there is an "Index:" line in the leading garbage, patch will try to use the filename from that line. The context diff header takes precedence over an Index line. If no filename can be intuited from the leading garbage, you will be asked for the name of the file to patch.

(If the original file cannot be found, but a suitable SCCS or RCS file is handy, patch will attempt to get or check out the file.)

Additionally, if the leading garbage contains a "Prereq:" line, patch will take the first word from the prerequisites line (normally a version number) and check the input file to see if that word can be found. If not, patch will ask for confirmation before proceeding.

If the patch file contains more than one patch, patch will try to apply each of them as if they came from separate patch files. This means, among other things, that it is assumed that the name of the file to patch must be determined for each diff listing, and that the garbage before each diff listing will be examined for interesting things such as filenames and revision level, as mentioned previously. You can give flags (and another original file name) for the second and subsequent patches by separating the corresponding argument lists by a "+". (The argument list for a second or subsequent patch may not specify a new patch file, however.)

Patch recognizes the following flages:

The **–b** flag causes the next argument to be interpreted as the backup extension, to be used in place of the tilde.

The **–B** flag causes the next argument to be interpreted as a prefix to the backup file name. If this argument is specified any argument from **–b** will be ignored. This argument is an extension to Larry Wall's patch v2.0.1.4, patchlevel 8, made by M. Greim (greim@sbsvax.uucp).

The **–c** flag forces patch to interpret the patch file as a context diff.

The **–d** flag causes patch to interpret the next argument as a directory, and cd to it before doing anything else.

The **–D** flag causes patch to use the "#ifdef...#endif" construct to mark changes.

The argument following will be used as the differentiating symbol. Note that, unlike the C compiler, there must be a space between the -D and the argument.

The **–e** flag forces patch to interpret the patch file as an *ed* script.

The **–f** flag forces patch to assume that the user knows exactly what he or she is doing, and to not ask any questions. It does not suppress commentary, however. Use **–s** for that.

The **–F***n* flag sets the maximum fuzz factor. This flag only applies to context diffs, and causes patch to ignore up to that many lines in looking for places to install a hunk. Note that a larger fuzz factor increases the odds of a faulty patch. The default fuzz factor is 2, and it may not be set to more than the number of lines of context in the context diff, ordinarily 3.

The –l flag causes the pattern matching to be done loosely, in case the tabs and spaces have been munged in your input file. Any sequence of whitespace in the pattern line will match any sequence in the input file. Normal characters must still match exactly. Each line of the context must still match a line in the input file.

The **–n** flag forces patch to interpret the patch file as a normal diff.

The –N flag causes patch to ignore patches that it thinks are reversed or already applied. See also **–R**.

The **–o** flag causes the next argument to be interpreted as the output file name.

The **–p***n* flag sets the pathname strip count, which controls how pathnames found in the patch file are treated, in case the you keep your files in a different directory than the person who sent out the patch. The strip count specifies how many slashes are to be stripped from the front of the pathname. (Any intervening directory names also go away.) As a simple example, let us suppose that the filename in the patch file is */u/howard/src/blurfl/blurfl.c* setting **–p** or **–p0** gives the entire pathname unmodified, **–p1** gives *u/howard/src/blurfl/blurfl.c* without the leading slash, **–p**4 gives *blurfl/blurfl.c* and not specifying **–p** at all just gives you *blurfl.c*. Whatever you end up with is looked for either in the current directory, or the directory specified by the **–d** flag.

The **–r** flag causes the next argument to be interpreted as the reject file name.

The **–R** flag tells patch that this patch was created with the old and new files swapped. (That does happen occasionally, human nature being what it is.) Patch will attempt to swap each hunk around before applying it. Rejects will come out in the swapped format. The **–R** flag will not work with *ed* diff scripts because there is too little information to reconstruct the reverse operation.

If the first hunk of a patch fails, patch will reverse the hunk to see if it can be applied that way. If it can, you will be asked if you want to have the **–R** flag set. If it cannot, the patch will continue to be applied normally. (Note: this method cannot detect a reversed patch if it is a normal diff and if the first command is an append (i.e. it should have been a delete) since appends always succeed, due to the fact that a null context will match anywhere. Luckily, most patches add or change lines rather than delete them, so most reversed normal diffs will begin with a delete, which will fail, triggering the heuristic.)

The **–s** flag makes patch do its work silently, unless an error occurs.

The **–S** flag causes patch to ignore this patch from the patch file, but continue on looking for the next patch in the file. Thus

```
patch –S + –S + <patchfile
```

will ignore the first and second of three patches.

The **–v**flag causes patch to print out its revision header and patch level.

The **–x***number* flag sets internal debugging flags, and is of interest only to patch patchers.

9.15. ZMODEM—FILE TRANSFER PROGRAM

The XMODEM, YMODEM, and ZMODEM family of file transfer programs are widely used on personal computers. MINIX supports ZMODEM, the most advanced of the set. The programs *sz* and *rz* are used for sending and receiving, respectively.

The sz Command

Sz uses the ZMODEM error correcting protocol to send one or more files over a dial-in serial port to a variety of programs running under MINIX, UNIX, MS-DOS, CP/M, VMS, and other operating systems. It is the successor to XMODEM and YMODEM.

ZMODEM greatly simplifies file transfers compared to XMODEM. In addition to a friendly user interface, ZMODEM provides Personal Computer and other users an efficient, accurate, and robust file transfer method.

ZMODEM provides complete end-to-end data integrity between application programs. ZMODEM's 32 bit CRC catches errors that sneak into even the most advanced networks.

Output from another program may be piped to *sz* for transmission by denoting standard input with –:

```
ls –l | sz –
```

The program output is transmitted with the filename *sPID.sz* where PID is the process ID of the *sz* program. If the environment variable *ONAME* is set, that is used instead. In this case, the command:

```
ls –l | ONAME=con sz –ay –
```

will send a "file" to the PC-DOS console display. The **–y** option instructs the receiver to open the file for writing unconditionally. The **–a** option causes the receiver to convert UNIX newlines to PC-DOS carriage returns and linefeeds. On UNIX systems, additional information about the file is transmitted. If the receiving

program uses this information, the transmitted file length controls the exact number of bytes written to the output dataset, and the modify time and file mode are set accordingly.

If *sz* is invoked with $SHELL set and if that variable contains the string *rsh* or *rksh* (restricted shell), *sz* operates in restricted mode. Restricted mode restricts pathnames to the current directory and *PUBDIR* (usually */usr/spool/uucppublic*) and/or subdirectories thereof.

The options and flags available are:

+

Instruct the receiver to append transmitted data to an existing file.

a

Convert NL characters in the transmitted file to CR/LF. This is done by the sender for XMODEM and YMODEM, by the receiver for ZMODEM.

b

Binary override: transfer file without any translation.

c

Send COMMAND (follows *c*) to the receiver for execution, return with COMMAND's exit status.

d

Change all instances of "." to "/" in the transmitted pathname. Thus, C.omenB0000 (which is unacceptable to MS-DOS or CP/M) is transmitted as C/omenB0000. If the resultant filename has more than 8 characters in the stem, a "." is inserted to allow a total of eleven.

e

Escape all control characters; normally XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.

f

Send Full pathname. Normally directory prefixes are stripped from the transmitted filename.

i

Send COMMAND (follows *i*) to the receiver for execution, return Immediately upon the receiving program's successful reception of the command.

L

Use ZMODEM sub-packets of length *n* (follows *L*). A larger *n* (32 <= *n* <=

1024) gives slightly higher throughput, a smaller one speeds error recovery. The default is 128 below 300 baud, 256 above 300 baud, or 1024 above 2400 baud.

l

Wait for the receiver to acknowledge correct data every *n* (32 <= *n* <= 1024) characters. This may be used to avoid network overrun when XOFF flow control is lacking.

n

Send each file if destination file does not exist. Overwrite destination file if source file is newer than the destination file.

N

Send each file if destination file does not exist. Overwrite destination file if source file is newer or longer than the destination file.

o

Disable automatic selection of 32 bit CRC.

p

Protect existing destination files by skipping transfer if the destination file exists.

q

Quiet suppresses verbosity.

r

Resume interrupted file transfer. If the source file is longer than the destination file, the transfer commences at the offset in the source file that equals the length of the destination file.

t

Change timeout. The timeout, in tenths of seconds, follows, the –t flag.

u

Unlink the file after successful transmission.

w

Limit the transmit window size to *n* bytes (*n follows* (**enw**).

v

Verbose causes a list of file names to be appended to */tmp/szlog*.

y

Instruct a ZMODEM receiving program to overwrite any existing file with the same name.

Y

Instruct a ZMODEM receiving program to overwrite any existing file with the same name, and to skip any source files that do have a file with the same pathname on the destination system.

Examples

Below are some examples of the use of *sz*.

```
sz −a *.c
```

This single command transfers all .c files in the current directory with conversion (**−a**) to end-of-line conventions appropriate to the receiving environment.

```
sz −Yan *.c *.h
```

Send only the *.c* and *.h* files that exist on both systems, and are newer on the sending system than the corresponding version on the receiving system, converting MINIX to MS-DOS text format.

The rz Command

Rz and *sz* are programs that uses an error correcting protocol to transfer files over a dial-in serial port from a variety of programs running under various operating systems. *Rz* (Receive ZMODEM) receives files with the ZMODEM batch protocol. Pathnames are supplied by the sending program, and directories are made if necessary (and possible).

The meanings of the available options are:

−a

Convert files to UNIX conventions by stripping carriage returns and all characters beginning with the first Control Z (CP/M end of file).

−b

Binary (tell it like it is) file transfer override.

−c

Request 16 bit CRC. XMODEM file transfers default to 8 bit checksum. YMODEM and ZMODEM normally use 16 bit CRC.

–D

Output file data to /dev/null; for testing.

–e

Force sender to escape all control characters; normally XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.

–p

Protect: skip file if destination file exists.

–q

Quiet suppresses verbosity.

–t

Change timeout tenths of seconds (timeout follows flag).

–v

Verbose causes a list of file names to be appended to */tmp/rzlog*. More v's generate more output.

–y

Yes, clobber any existing files with the same name.

–D

Output file data to /dev/null; for testing.

–e

Force sender to escape all control characters; normally XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.

–p

Protect: skip file if destination file exists.

–q

Quiet suppresses verbosity.

–t

Change timeout tenths of seconds (timeout follows flag).

–v

Verbose causes a list of file names to be appended to */tmp/rzlog*. More v's generate more output.

–y

Yes, clobber any existing files with the same name.

10.2. LIST OF MINIX SYSTEM CALLS

MINIX has a total of 49 system calls, most of them identical to UNIX V7 calls in terms of name, function, and parameters. They are listed below alphabetically:

Returns	Prototype	- Description
int	access(char *path, int amode)	- Determine if access permitt
unsigned	alarm(unsigned int sec)	- Set alarm clock timer
char *	brk(char *addr)	- Change size of data segmen
int	chdir(char *path)	- Change working directory
int	chmod(char *path, mode_t mode)	- Change mode of file
int	chown(char *path, uid_t owner, gid_t group)	- Change file's group id
int	chroot(char *path)	- Change root directory
int	close(int fd)	- Close file
int	creat(char *path, mode_t mode)	- Create file
int	dup(int fd)	- Duplicate file descriptor
int	dup2(int fd, int fd2)	- Duplicate file descriptor 2
int	exec(...)	- Execle, execve, etc.
void	exit(int status)	- Terminate process
int	fcntl(int fd, int cmd, int arg)	- Misc. controls
pid_t	fork()	- Fork
int	fstat(int fd, struct stat *buf)	- Stat file
gid_t	getegid()	- Get effective group id
uid_t	geteuid()	- Get effective user id
gid_t	getgid()	- Get group id
pid_t	getpid()	- Get process id
uid_t	getuid()	- Get user id
int	ioctl, (int fd, int request, struct sgttyb *argp)	- Set tty parameters
int	kill(pid_t pid, int sig)	- Send a signal
int	link(char *path1, char *path2)	- Link file
off_t	lseek(int fd, off_t offset, int whence)	- Seek
int	mkdir(char *path, mode_t mode)	- Make directory
int	mknod(char *name, int mode, int addr, int size)	- Create special file
int	mount(char *name, char *special)	- Mount file system
int	open(char *path, int oflag, mode_t mode)	- Open a file
int	pause()	- Suspend caller
int	pipe(int fd[])	- Create pipe
long	ptrace(int req, pid_t pid, long addr, long data)	- Trace a process
int	read(int fd, char *buf, unsigned nbyte)	- Read data from file
int	rename(char *old, char *new)	- Rename file
int	rmdir(char *path)	- Remove directory
int	setgid(gid_t gid)	- Set gid
int	setuid(uid_t uid)	- Set uid

void	signal(int signr, void (*func()))	- Enable signal catching
int	stat(char *path, struct stat *buf)	- Get file statistics
int	stime(long *timep)	- Set the wall clock time
int	sync()	- flush cache to disk
time_t	time(time_t *loc)	- Get time since 1970
clock_t	times(struct tms *buffer)	- Get accounting times
mode_t	umask(mode_t cmask)	- Set file mask
int	umount(char *name)	- Unmount file system
int	unlink(char *path)	- Unlink file
int	utime(char *path, struct utimbuf *times)	- Set file times
pid_t	wait(int *stat_loc)	- Wait for child to exit
int	write(int fd, char *buf, unsigned nbyte)	- Write data to file

11

NETWORKING

MINIX supports networking. This chapter describes the kind of support provided, how to use it, and how it should be installed.

11.1. INTRODUCTION

Network software can be divided into two general categories differing in the way the software is integrated into the operating system and the user software. When networks first developed, they were used over slow wide-area links (56 kbps or less), so the designers' main concern was using the available bandwidth efficiently. Programmer convenience was not considered. Later, as higher bandwidth networks became widespread (especially local area networks, such as Ethernet), the focus changed from worrying about bandwidth utilization, to worrying about making the network interface convenient for the programmers. This evolution is very similar to the evolution from assembly language programming, where the machine came first, to programming in high level languages, where the programmer came first.

Networks of the first type of are said to be connection oriented, and use what are called *sliding window protocols*. All older networks, especially wide area networks, are of this type. Some of the better known protocols are X.25, TCP/IP, and OSI. Networks of the second type are connectionless, and use what is called remote

procedure call (RPC). Virtually all modern distributed operating systems are based on this concept. Some well-known examples are the work of Xerox PARC [1], the V kernel [2], and Amoeba [3-11]. While it is certainly possible to build RPC on top of a connection-oriented protocol, this approach is inefficient compared to building the RPC on top of the bare network. For an introduction to connection-oriented protocols, RPC, and networking in general, see [12].

Networking in MINIX is based on RPC. Briefly summarized, communication between two processes works as follows. One of the processes, called the *server*, has some service to offer, such as a file storage. The other process, the *client*, wants to use this service. The interface to the service consists of a collection of procedures that the client can call. In the case of a file server, the procedures might be CREATE_FILE, RENAME_FILE, READ_DATA, WRITE_DATA, and so on. These are library routines available on the client's machine.

When the client calls one of these procedures, the procedure sends a message to the server containing the procedure name and its parameters. The procedure then blocks waiting for the reply. When the message gets to the server, it is decoded there and executed. The reply is sent back to the calling procedure on the client's machine, which then returns the results to the caller. From the programmer's point of view, having remote services in the network essentially means that there is a new collection of procedures to call. The programmer is not burdened with concepts like opening connections, sending data, or thinking in terms of acknowledgements, all of which are needed in the connection-oriented model. Nor is the network software burdened with having to manage connections.

In effect, RPC is based on the abstraction of the procedure call, whereas connection-oriented networks are based on the much lower-level concept of making the network look like an input/output device. While at first glance it might seem that connection-oriented networking could be made to fit with the UNIX/MINIX concept of a pipe, pipes are set up in a very different way (by a common ancestor), and fit very poorly to the most common style of local area network programming, where the client has a request and the server gives a response. With wide area networks, this kind of interaction is painfully slow, due to the low bandwidth. so the only services generally available are mail and file transfer, which are batch-oriented. MINIX networking has been designed for interactive use on high performance local area networks, so for this reason, RPC has been chosen over the older connection-oriented style.

In particular, MINIX networking has been designed to be compatible with the form of RPC used in the Amoeba distributed operating system [3-11]. Not only have the concepts and the implementation been well tested, but the performance is exceedingly good. For example, for doing file transfers, something that connection-oriented protocols are supposed to be good at, Amoeba running on two Sun 3s achieves triple the throughput of TCP/IP running on the same hardware. Data transfers between two Zenith Z-248s running the Amoeba RPC on MINIX have been measured at 165 kbytes/sec, almost as fast as TCP/IP transfers between two

Sun 3/50s. Considering that the Suns are two times as fast as the Z-248s and the network software is 100% CPU limited (doubling the CPU speed doubles the throughput), this is a strong argument for the Amoeba RPC. As a final statistic, the RPC throughput between a client and server located on the same Z-248 is 1.5 Mbytes/sec, an extremely high figure for this class of machine, and much better than what Suns and VAXes normally achieve locally, despite their greater CPU power. In conclusion, although RPC was chosen for its elegance and ease of use, it turns out that it also has excellent performance, even doing things like bulk transfer, and certainly doing things like short request-reply interactions.

A few words about Amoeba are probably in order here. It is a distributed operating system that was developed at the Vrije Universiteit in Amsterdam and is now being further developed there and at the Centre for Mathematics and Computer Science in Amsterdam. It currently runs on the Sun 3 (and other 680x0 processors), VAXstations, and 80386s. Note that Amoeba is a complete operating system, just like UNIX, MINIX or VMS. The only relation between Amoeba and MINIX is that MINIX networking uses the Amoeba RPC protocols. Other than that they are quite different in structure, funtionality, and goals. Amoeba was designed to run on systems consisting of dozens of processors, and yet give the programmer the illusion that it is a traditional single-CPU time sharing system. For more information about Amoeba, see the references.

11.2. OBJECTS

Amoeba is an object-based system, and to a considerable extent this orientation is reflected in the protocol. As a consequence, MINIX also acquires a certain object-orientation. Very briefly, an object is a programmer defined abstract data type that has well-defined operations on it. As an example, a file server could define file and directory objects, and provide operations to read and write the file objects, and insert files in, and delete files from, directory objects. Clients can perform these operations by doing RPCs with the file server. Henceforth we will adopt the Amoeba terminology and call these RPCs *transactions*. A transaction consists of a request message from a client to a server, followed by a reply message from the server back to the client.

It is up to the writer of each server to decide what kinds of objects the server will support and what operations will be available on them. The structure of the system guarantees that clients can only perform the operations provided by the server. This style of networking is intended to force constraints on programmers, just as high-level languages force constraints on former assembly-language programmers.

Objects are normally protected by capabilities, which are currently (Amoeba 4.0) 128-bit numbers, although in the the next version of Amoeba (Amoeba 5.0) this will become 256 bits. When a client asks a server to create an object, the server

returns a capability for the object. This capability must be presented by the client to perform subsequent operations on the object. In Amoeba, capabilities are protected crytographically. Since the MINIX kernel, unlike the Amoeba kernel, was not designed from scratch as a distributed system, the protection aspects in MINIX are not fully implemented.

A capability has 4 fields, described below. These fields are important because they appear in the Amoeba and MINIX message headers.

Port:	48-bit number used to identify the server owning the object.
Object:	24-bit number used by the server to identify the object
Rights:	8 bits telling which operations are allowed
Cksum:	48-bit checksum to prevent tampering with the capability

The *port* field is a (random) 48-bit number used for addressing. Any 48-bit number can be used as a port. In some situations, an ASCII string can be used as a port, with the first 48 bits taken as the port number. All messages in Amoeba and MINIX are sent to ports, not to machine addresses. The mapping of ports to machine addresses is done deep down in the system, and is of little concern to the average programmer. Thus: a port uniquely identifies a server and provides a logical address to which all messages for the server are sent.

The remaining three fields are called the private part of the capability. In theory, each server can use them any way it wants to. In practice, to prevent total chaos, all existing servers adhere to the following conventions (just as most UNIX programs adhere to the convention that certain files contain ASCII characters with a line feed at the end of each line). The *object* field is used by the server to identify the specific object being accessed. For example, when a file server created a new file on behalf of a client, it could put the i-node number of the new file in this field, so that when the client later used the capability, the server could tell which file was being addressed. The field is 24-bits long, providing each server with 16 million object identifers.

The *rights* field contains a bit map for up to eight protected operations. Each bit controls permission to perform one operation. Thus a file server could allocate bit 0 for READ_DATA, bit 1 for WRITE_DATA, bit 2 for APPEND_DATA, bit 3 for DELETE_FILE, and so on. When a capability arrives from a client, the server checks to see if the bit corresponding to the relevant operation is on. If it is not, the operation is rejected. In this way, a user can create a file, ask the server to turn off the WRITE_DATA and DELETE_FILE bits, and then give the capability to another user. This new user cannot perform WRITE_DATA and DELETE_FILE operations, but can perform the operations whose bits are turned on.

A moment's thought will reveal that the above protection scheme is worthless if users can turn the rights bits on and off by themselves. To prevent this, the *cksum* field is used. When creating a new object, the server simultaneously creates a random number and stores it in its internal tables (e.g., in the i-node). It then combines the rights bits and the random number, and passes the result through a one-way

cryptographic function. The result of this function is put in the *cksum* field. When a capability comes in from a client, the server uses the object number to locate the original random number. It then combines it with the rights bits present in the capability, and runs the result through the one-way function. If the result disagrees with the *cksum* field, the capability is considered invalid, and an error return is sent back. In this way, users who change the rights bits will simply invalidate their capabilities. Attempts to break the scheme by finding an inverse to the one-way function can be handled by choosing a cryptographically strong one-way function. Brute force does not work either, as picking checksums at random will require, on the average, 2**47 attempts to guess the 48-bit checksum. Since a null transaction over a 10 Mbit/sec Ethernet using SUN 3/50s takes about 1.4 msec, about 3000 years are needed to perform the search. Furthermore, it is easy enough to program a server to artificially increase the transaction time to 1 sec after 10 unsuccessful attempts have been made, thus increasing the mean search time to 3,000,000 years.

11.3. OVERVIEW OF TRANSACTIONS

To summarize what we have covered so far, the normal style of networking in MINIX (and Amoeba) is to structure dialogues in terms of clients and servers. Each server manages one or more types of objects, and provides operations for clients to perform operations on these objects. When a client asks a server to create an object for it, the server then returns a capability for the object to the client. This capability identifies the server, identifies the object, and tells which subset of the operations the holder of the capability may perform. To have an operation performed, the client sends a request message to the server (with the capability embedded in the message header), and the server then sends back a reply. In most cases, the calls to the server are embedded in library procedures, called *stubs*, to encapsulate the message passing and hide it from the users.

Transactions provide a basis for a large number of user services. In MINIX, users can use them to build arbitrary services. Two key services are provided as standard for MINIX, remote execution and remote file copying. These services make use of a process called the shell server, or sherver for short. The sherver accepts messages from remote (or local) clients, executes the commands in them, and returns the output.

Communication is implemented as follows. Each server listens to a unique 48-bit port. A client that wants service from the server sends a request to that port and blocks until it receives a reply. (If the client cannot find anyone listening to the port after a given period, it times out and returns an error status.) When the server is ready, it returns a reply to the client, which then continues execution. Each transaction is independent of the previous transactions; there is no connection or virtual circuit.

Clients must have some way of discovering a server's port. Under Amoeba a

directory server is used. The directory server stores capabilities for objects and associates them with an ASCII string. The directory server has a well known port. Under MINIX you make initial contact with a sherver that has a well known port and then the sherver creates a secret port for all further transactions on that machine.

There are four stub routines in the user library which provide the basic interface between user processes and transactions. They are:

1. *getreq* - Get request (used by servers to get a request)
2. *putrep()* - Put reply (used by servers to send reply)
3. *trans()* - Transaction (used by clients to do a transaction)
4. *timeout* - Sets the time limit at which *trans* gives up

Getreq() and *putrep* are used by servers to get a request from a client and to send a reply. A server may not do a *getreq* until it has replied to the previous *getreq*. The call *trans* is used by clients to send a request to a server. It blocks until a reply or a signal arrives, or, if it cannot find a server listening to his port, it times out and returns an error code. The length of the timeout is set using the function *timeout*. This timeout has to do with locating servers, not how long they have to do the work.

Messages of up to 30000 bytes can be sent between client and server. This limit will increase to 1 Gbyte in the next version of Amoeba but will probably remain at 30000 bytes in MINIX due to the small address space of the IBM PC. It is possible to provide security so that servers only execute remote procedure calls for authorized users. The protection mechanism uses capabilities and is discussed in detail in the references. It will not be discussed much here. This protection mechanism is not implemented in the remote shell software available with MINIX. (It requires a directory server, among other things. The implementation is left as an exercise for the reader.)

11.4. SYNTAX AND SEMANTICS OF TRANSACTION PRIMITIVES

Now we will take a detailed look at the syntax and semantics of the library routines for using transactions, followed by some simple examples to indicate how the functions are typically used. Remember, that when programming with transactions, the primitives used in C programs are *getreq*, *putrep*, *trans*, and *timeout*. These can be thought of as network system calls, although they are not implemented quite like that in MINIX. If you are building a server, it will typically have a main loop with a *getreq* at the top, a switch in the middle based on some field of the incoming message, and a *putrep* at the bottom. Furthermore, the server writer will generally also provide a set of stub procedures that contain *trans* calls to access the server. The average user will call these library procedures, and will not make *trans* calls directly, although he is, of course, free to do so if he wishes.

Transaction messages always begin with a special header. The exact layout of these messages is defined by the Amoeba protocol. By using this protocol, MINIX

machines can communicate with one another, and with Suns and Vaxes running Amoeba. Device drivers have also been written for UNIX to allow UNIX processes to speak Amoeba, and have Amoeba clients and servers run on UNIX. At the Vrije Universiteit, all the Suns, Vaxes, and other machines that run UNIX have such drivers to communicate with each other and with machines running Amoeba and MINIX. It is the local lingua franca, just as TCP/IP is at some sites.

The Amoeba header is defined in the header file */usr/include/amoeba.h*, which must be included in all programs using transactions. The header definition is given below. The types used in the header struct are also defined in *amoeba.h*.

```
typedef struct {
    port h_port;            /* port (i.e., logical address) of the dest. */
    port h_signature;       /* used for authentication and protection */
    private h_priv;         /* 10 bytes: object, rights, and cksum */
    unshort h_command;      /* code for operation desired/status returned */
    long h_offset;          /* parameter field */
    unshort h_size;         /* parameter field */
    unshort h_extra;        /* parameter field */
} header;
```

The message header contains the port to which the message should be sent, a command/status field for use by the server and space for some parameters to go with the command or status. Let us now look at the four network primitives. The first one, getreq, has the following declaration:

```
unshort getreq(hdr, buffer, size)
header *hdr;
char *buffer;
unshort size;
```

The three parameters refer to the header, the buffer, and the buffer size, respectively. In a sense, they are analogous to the parameters of the MINIX READ and WRITE system calls. The *hdr* parameter points to a header struct, which is used to allow the server to specify which port it wants to listen to. The *h_port* field of the header must be initialized with the port number. The *buffer* parameter is a pointer to a buffer to hold the incoming message. It can hold a maximum of size bytes, specified by the third parameter. If successful *getreq* returns the number of the bytes of data in the buffer that were actually received. In addition, the other fields of the header are filled in by the system. If an error occurs then it returns a negative error code. Possible error codes (defined in *amoeba.h*) are:

FAILED:	- Null port or *getreq* done before previous *putrep*
BADADDRESS:	- The buffer pointer and/or size was not valid
ABORTED:	- A signal was received
TRYAGAIN:	- There were no free transaction slots in the kernel tables

Note that after a *getreq*, *trans* may be used to communicate with another server before doing the *putrep*. In other words, a server may call other servers to help it do its job, but it may not process multiple transactions simultaneously. (In Amoeba, server processes may contain multiple threads to allow parallelism, but MINIX does not allow multiple threads per process.)

The next call is *putrep*, used by servers to reply to requests and send back results and status information. The declaration is:

```
unshort putrep(hdr, buffer, size)
header *hdr;
char *buffer;
unshort  size;
```

The header returned contains status information, and possibly a new port (in the *h_signature* field). A buffer containing size bytes of data is also returned to the client. If successful, *putrep* returns the number of bytes sent. The reply message is not acknowledged, so that a successful return from this call does not guarantee that the client got the reply. In general, it is up to the client to try again if the reply is not forthcoming quickly enough. Possible error conditions for *putrep* are defined in *amoeba.h* as follows:

FAILED:	- No *getreq* was done first
BADADDRESS:	- The buffer pointer and/or size was not valid
ABORTED:	- A signal was received

Now we come to the call used by clients to request services and wait for replies. Servers can also use this call to request services from other servers. Thus at one instant a process may be acting as a server and at another the same process may be acting as a client. The client call is:

```
unshort trans(hdr1, buffer1, size1, hdr2, buffer2, size2)
header *hdr1, *hdr2;
char *buffer1, *buffer2;
unshort size1, size2;
```

The call has two independent sets of parameters. Those with suffix 1 are used for sending the request message to the server. Those with suffix 2 are used for getting the reply. Both sets have a header, a buffer, and a size. The two *hdr* pointers point to structs for message headers. The first one contains parameters copied to the outgoing message to the server and the second one contains space for the data to be

copied in from the server's *putrep*. The two buffer parameters are for the outgoing and incoming data, respectively, and the two sizes tell how large these buffers are.

After making a *trans* call, the client blocks until the message has been sent, received, processed by the server, and replied to. Only then can the client continue execution. At this point the fields of *hdr2* and *buffer2* will contain the reply data. Like MINIX itself, transactions support only this synchronous form of communication. Experience has painfully shown that asynchronous stream communication is difficult for programmers to deal with. After all, everything else in programming languages is synchronous. (Can you imagine what it would be like to have a procedure call return control to the caller before having finished its work?)

If successful, *trans*, returns the number of bytes in the reply. Possible error codes are:

FAILED:	- Null port or server crashed between *getreq* and *putrep*
NOTFOUND:	- The port locate failed to find a server before the timeout
BADADDRESS:	- A buffer pointer and/or size was not valid
ABORTED:	- A signal was received
TRYAGAIN:	- There were no free transaction slots in the kernel's tables

The final network primitive deals with setting timeouts. When a client first does a transaction on a previously unknown port, the kernel broadcasts a locate message to find the server. It then waits a certain amount of time for a server to reply. If no server replies before the timer goes off, the *trans* fails with NOTFOUND. The *timeout* call allows the client to determine how long to wait for a server to reply. After a reply has been received, the kernel keeps it in a cache, so that locates will not be needed subsequently. It is important to realize that the timeout relates to locating servers, not to how much time servers have to perform their work. The declaration is:

```
unshort timeout(time)
unshort time;
```

The function sets the length of the locate timeout in tenths of a second. The default is 300 (30 seconds). A timeout of 0 means do not time out. The *timeout* call returns the length of the previous timeout.

11.5. SERVER STRUCTURE

A typical server has the following form:

```
/* Declarations needed by the server. */
header hdr;                          /* header for receiving requests */
char buffer[BUFSIZE];                /* buffer for receiving requests */
char reply[BUF2SIZE];                /* buffer for sending replies */
unshort size, replysize;             /* sizes of the two buffers */
unshort getreq;                      /* function declaration */
char *strncpy();                     /* string function */

signal(SIGAMOEBA, SIG_IGN);          /* ignore signals */

while (1) {

  /* Have the server listen to a 48-bit port equal to ASCII "MyServ" */
  strncpy(&hdr.h_port, "MyServ", HEADERSIZE);

  /* Wait for a request to come in for that port. */
  size = getreq(&hdr, buffer, BUFSIZE);

  /* If the size returned is negative then an error occurred. */
  if ((short) size < 0) {
       handle_error();
  } else {
       perform_request();            /* carry out the work */
       hdr.h_status = OK;            /* or whatever */
       putrep(&hdr, reply, replysize); /* send reply back */
  }
}
```

If all the information necessary for the request is in the headers then the buffers in *getreq* and *putrep* can be replaced by the value NILBUF and the buffer sizes can be replaced by 0.

11.6. CLIENT STRUCTURE

The structure of a client program is much more variable. A program that deals with the above server might look like this:

```
/* Declarations needed by the client. */
header hdr;                            /* header used for request */
char buffer[BUFSIZE];                  /* buffer used for request */
short size;                            /* size of the buffer */
unshort trans;                         /* function declaration */
char *strncpy();                       /* string function */

/* Initialize server port to "MyServ". */
strncpy(&hdr.h_port, "MyServ", HEADERSIZE);

/* Send request to server listening to that port. */
size = (short) trans(&hdr, buffer, BUFSIZE, &hdr, NILBUF, 0);
if (size < 0) {
      printf("trans failed %d0, size);
} else {
      if (hdr.h_status != OK)          /* nonzero status is an error */
            work_not_done();
                                       else
            successful_trans;
}
```

11.7. SIGNAL HANDLING

It is important for programmers to understand how signals work. If a client receives a signal while doing a *trans*, the signal propagates to the server. If the server is also doing a *trans* then it propagates again to the next server, and so on. The aim of this is to request all servers to terminate their transaction as soon as possible.

If the server receiving the signal is not doing a transaction and not already doing a *putrep* then the server code must handle the signal. It may choose to catch the signal and send a reply immediately or simply ignore the signal. If it does not catch the signal then it will die since the signal propagated is SIGAMOEBA (which is defined as SIGEMT for MINIX). In this case the transaction will fail (with return status FAILED for the client).

Once the transaction is completed the client process will be signaled. It in turn must handle the original signal (not necessarily SIGAMOEBA). The exact transaction semantics of Amoeba are not supported under MINIX due to difficulty in

keeping user processes alive until a transaction terminates after a signal. Signal propagation does occur, but the client may die before a reply comes in. This should not matter too much for most applications. In the next rewrite of Amoeba the syntax and semantics of these functions will change in non-compatible ways, but this will probably not appear in MINIX.

11.8. IMPLEMENTATION OF TRANSACTIONS IN MINIX

Amoeba transactions are implemented in the MINIX kernel as a number of kernel tasks. Several alterations were made to the kernel to support these tasks, including the addition of an (optional) ethernet driver (for the Western Digital EtherCard Plus, also known as the WD1003E) and the possibility to specify the size of the stack for kernel tasks on a per task basis. (Amoeba tasks need larger stacks than the other MINIX kernel tasks.) There is also an extra system call that is handled by MM. This is the Amoeba system call and is the interface to the kernel. Special handling of signals is also provided for in the MM task.

There are five kernel tasks for Amoeba. The first acts as a manager which accepts asynchronous events. Possible events are:

1. An ethernet packet has arrived
2. A local signal has arrived
3. A user task involved in an active transaction has died
4. A sweep timeout has occurred

(Locate timeouts are implemented using a counter which is decremented every tenth of a second by a sweep routine.) Each of the other four tasks manage a single user process' transactions. Thus, a maximum of four processes can simultaneously do transactions under MINIX. The number of transaction tasks is, however, a constant in an include file and can be increased if needed.

In the MINIX kernel there is a table which keeps a record of the current state of a transaction. This table is called *am_task* and is declared in the file *amoeba.c*. This records many things, including, the process number of the task doing the transaction, the current state (locating, waiting for a reply, waiting for a request, etc.) and the relevant ports and machine addresses.

The Amoeba network protocol is a stop and wait protocol that guarantees at most once delivery of a message. A message consists of the concatenation of the transaction header with the data in the buffer (if any) given to *trans*, *getreq* or *putrep*. The transaction code divides messages up into packets which fit on the underlying network medium (which is ethernet in the case of MINIX). It then sends over the message fragments and they are reassembled on the remote machine before being given to the recipient.

Each packet begins with an ethernet header (which consists of the source and destination ethernet addresses) followed by a 10-byte Amoeba internet header

containing data about the source and destination processes to ensure that the message is delivered to the correct process. The rest of the packet is used for sending data.

11.9. COMPILING THE SYSTEM

There are several interesting things you need to know before you can build a MINIX kernel with Amoeba transactions in it. First of all, you do not need an Ethernet to use transactions. You can have your clients and servers running on a single machine. In this mode, it is possible to write and debug network software without having a network. Later, when you move to a real network, the code will already be fully debugged, as the system itself makes no distinction between local and remote transactions.

Second, the transaction code is quite substantial. So much so that it would tend to overshadow the rest of MINIX if it were fully integrated into it. This fact, combined with the knowledge that not all MINIX users are interested in networking has led to adding a new top-level directory in MINIX, *amoeba*. This directory and its subdirectories contain all the networking code. If you are not interested in networking, just ignore it.

Installation of networking is largely auto-configured using the makefiles provided. Two new **–D** entries are used in the *mm* and *amoeba/kernel* makefiles:

–DAM_KERNEL	(used in *mm* and *amoeba/kernel*) enables networking
–DNONET	(used in *amoeba/kernel*) single machine networking

in other words, local transactions only If you use **–DAM_KERNEL** but not **–DNONET**, you get full networking and MUST have a Western Digital Etherplus card.

If you add a new kernel task of your own then it MUST come between the Amoeba kernel tasks and the printer task in the file kernel/table.c and should be numbered relative to AMOEBA_CLASS in the file *h/com.h* (i.e. The task number should be AMOEBA_CLASS+1 for the first new task, AMOEBA_CLASS+2 for the second new task, etc.). Be sure to set NR_TASKS correctly.

To compile and install networking, you must follow the steps below carefully.

11.10. HOW TO INSTALL NETWORKING IN MINIX

You must do the following important steps carefully. However, before starting, make sure that */usr/lib/cpp* has at least 50000 bytes of stack space (*size* will tell you). If you, use *chmem* to give it more.

1. Make sure that you are in the Amoeba directory and that there is plenty of free disk space. Now edit *Makefile* to include or exclude *NONET* from *CFLAGS* as you prefer.

2. Type:

   ```
   make
   ```

3. When you are instructed to do so, insert a blank diskette and hit the return key.

4. Reboot your machine using the new boot floppy.

5. Test the system. The directory *amoeba/examples* contains several programs to test the reliability of transactions. The *READ_ME* file in the directory gives more details.

6. If you have an ethernet card then install the network tools. The directory amoeba/util contains utilities for remote shells, remote file copying and message sending. These only work with machines that have Amoeba transactions installed. The *READ_ME* file there gives more details.

11.11. NETWORKING UTILITIES

There are several utility programs which you may find useful if you have a network connection. They are listed below with a brief outline of their use. Other utilities are possible and reasonably simple to write as shell scripts that use *rsh* (remote shell, described below). The utilities are located in the amoeba/utilities directory.

11.12. REMOTE SHELL

One of the main features of MINIX networking is the use of the remote shell. This utility is a server that accepts commands over the network from clients and executes them. The syntax of this command is:

rsh [**–bei**] *port command*

This program executes the command specified by command on the machine with a sherver (described below) listening to the port *port*, which is an ASCII string of up to 6 characters. It is used to generate a unique port name for the underlying transaction mechanism.

Normally standard output and standard error from the command are written on standard output of the local process. If the **–e** flag is specified then they are kept

separate. The **–i** flag specifies that standard input for the command should come from the local process. The **–b** flag specifies that the *rsh* should be started in the background. Some examples:

```
rsh bozo
```

starts an interactive shell on the machine running a sherver with port *bozo*. Subsequent commands that you type will be fed to the remote shell. You can use cd to change to a directory on the remote machine, ls to list files in the remote directory, and any other commands you want. In effect, *rsh* gives you a simple form of remote login. Note that to make this work, the remote process listening on the port *bozo* must be a shell server (sherver).

As a second example of *rsh*, consider

```
rsh jumbo cat /etc/passwd
```

which displays on your screen the file */etc/passwd* from the machine running a sherver with port jumbo. The *rsh* command could also have redirected this output to a local file or pipe.

A slightly more complex example is

```
rsh –i freddo 'cat >/usr/ast/junk' </etc/termcap
```

which runs the command

```
cat >/usr/ast/junk
```

on machine the machine running a sherver with port *freddo* and takes as input the file */etc/termcap* from the local machine. Note that by quoting the second argument, it is passed as a string to the remote sherver. If the command contains magic characters (e.g., *.c) the resulting action depends on whether the command is quoted or not. If it is not quoted, the local shell will expand the magic characters before *rsh* is even called. If the command is quoted, the command string is passed unmodified to the remote sherver, which then expands it in the directory it is currently working in.

When you log into a remote machine with *rsh*, you get a shell having the uid and gid of the sherver (see below). To get your own uid and gid, type

```
exec su george
```

assuming that your login is *george*. If you have a password, *su* will ask for it. Needless to say, the *su* program will use */etc/passwd* on the remote machine. Do not forget to use exec, as this eliminates the need for an extra shell. If you do not need your own uid, do not bother, as it costs memory.

11.13. SHERVERS

To enable remote shell operations, it is necessary to have a sherver running on the destination machine. Shervers can be started up by:

```
sherver port
```

assuming that sherver is kept in */usr/bin*. This program listens to the port specified and accepts a single request from the program *rsh*. It then executes it with the uid and gid of the sherver. When it is finished, the sherver exits.

The sherver gets its input from a pipe. This means that it can only do those things possible with a pipe as input. In particular, signals (e.g., DEL), EOF (e.g., CTRL–D), and the ioctl system call do not work properly. Hitting DEL remotely will kill the sherver. There is no simple solution, except to use stty to change your DEL character so that you do not hit it out of habit.

11.14. MASTERS

Another useful program is *master*. It is started up as follows:

```
master count uid gid command
```

This program starts up count copies of the program specified by command with user id uid and group id gid. The command may be given parameters. If at any time the command exits or dies then *master* will start up a new invocation of it. This was designed to work with shervers but has other applications as well. For example,

```
/usr/bin/master 1 2 2 /etc/sherver mumbo
```

will start a single sherver listening to the port *mumbo* and ensure that there is always a sherver running. This sherver will have uid=2 and gid=2, so that *rsh* calls to *mumbo* will be executed with this uid/gid combination. It is suggested to start up *master* in the */etc/rc* file of any machine running shervers. When a sherver finishes executing a command, it exists. By having *master* running in the background all the time, every time a sherver exists, its parent, *master*, will create a new one. This mechanism is somewhat akin to init creating a new login process whenever a shell exits. Since $PATH is generally not set prior to executing */etc/rc*, *master* should be specified as */usr/bin/master*.

The amount of stack space to give to *master* (and *sherver*) is important. If it is too little, the programs will act weird. If it is too much, everything will work fine, but memory will be wasted and there may not be enough left to run all the programs. Some experimentation is required. In any event, if things act strange, use *chmem* to allocate more stack space to these programs to see if that helps.

11.15. FILE TRANSFER

The standard MINIX networking provides for file transfer using a shell script called rcp (remote cp). The syntax of the call is

```
rcp [port!]from_file [port!]to_file
```

It can also do local file copy but this is more easily accomplished with cp. Here are two examples of rcp usage:

```
rcp jumbo!/etc/passwd
rcp jumbo!/etc/passwd freddo!/usr/ast/pebble
```

The first one will copy the file */etc/passwd* from the machine running a sherver with the port jumbo to the file *passwd* in the current directory. The second one will copy the file */etc/passwd* from the machine running a sherver with the port jumbo to the file */usr/ast/pebble* on the machine running a sherver with the port *freddo*. Thus it is possible to issue commands on machine A to copy files from machine B to machine C.

11.16. REMOTE PIPES

It is possible to set up remote pipes using the programs *to* and *from*. The program *to* reads from standard input and writes its output to the named port. Similarly, *from* reads from the named port and writes to standard output. For example, consider the following commands, possibly given on two different machines:

```
cat F* | sort | to 'port66'
from 'port66' | uniq -c | sort -n
```

The first command concatenates files beginning with 'F', sorts them, and writes the output to 'port66'. The second commands reads from 'port66' and provides input to the rest of the pipeline.

11.17. THE ETHERNET INTERFACE

The ethernet driver in this version of Minix is for the Western Digital Ethercard Plus card, which is also known as the WD1003E. The ethernet controller chip on this board is the National Semiconductor DP8390. If you have a different type of ethernet controller then there are several things you need to know about the interface between the driver and the Amoeba transaction layer in order to write a suitable driver for your card.

There were several fundamental assumptions made while designing the high level protocol which affect the ethernet driver.

1. The ethernet controller has enough local memory to buffer at least one incoming packet and one outgoing packet and will not overwrite a buffer with a new incoming packet until the buffer has been released.

2. Read buffers are released in the same order as they were allocated. After a read interrupt has occurred and (**bufread*)() has been called, then *bufread* will not be called again until an eth_release has been done.

3. The ethernet driver generates no write interrupts. This is because we found that busy waiting was more efficient than doing a context switch and waiting for an interrupt. By the time the context switch was done, the interrupt had already happened, so we had to switch back. It's faster to just wait for it. On a very slow machine, a different strategy might be
appropriate.

There are several routines used by the high level code which should be provided by the ethernet driver. Unless otherwise stated, these routines are called in the file *amoeba.c*.

1. *etheraddr* - get ethernet address of this host from rom.

2. *eth_init* - initialises the ethernet card and sets pointers to routines to be called on packet arrival and departure.

3. *eth_getbuf* - returns pointer to next write buffer.

4. *eth_write* - writes the current "write buffer" to the net.

5. *eth_release* - release a read buffer for reuse.

6. *eth_stp* - shuts up the ethernet chip so that reboot can stop all interrupts from the chip. The normal reboot procedure does not stop the WD1003E from running, so the next time interrupts are enabled it makes a fuss (called from klib88.s).

The files *dp8390.c*, *dp8390.h*, *dp8390info.h* and *dp8390stat.h* contain routines specific to the NS DP8390 chip. These may need some slight changes before working correctly with another manufacturer's board which also uses this chip. The files *etherplus.c* and *etherplus.h* contain routines specific to the WD1003E board.

11.18. REFERENCES

1. Birrell, A.D., and Nelson, B.J.: "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39-59, Feb. 1984.

2. Cheriton, D.. "The V Kernel: A Software Base for Distributed Systems," *IEEE Software Magazine*, vol. 1, pp. 19-42, April 1984.

3. Bal, H.E., Renesse, R. van, and Tanenbaum, A.S.: "Implementing Distributed Algorithms using Remote Procedure Call," *Proc.* National Computer Conference AFIPS, pp. 499-505, 1987.

4. Renesse, R. van, Tanenbaum, A.S., Staveren, H., and Hall, J.: "Connecting RPC-Based Distributed Systems using Wide-Area Networks," *Proc. Seventh International Conf. on Distr. Computer Systems*, IEEE, pp. 28-34, 1987.

5. Tanenbaum, A.S., Mullender, S.J., and van Renesse, R.: "Using Sparse Capabilities in a Distributed Operating System," *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, 1986.

6. Mullender, S.J., and Tanenbaum, A.S.: "The Design of a Capability-Based Distributed Operating System," *Computer Journal*, vol. 29, pp. 289-299, Aug. 1986.

7. Tanenbaum, A.S., and Renesse, R. van: "Distributed Operating Systems," *Computing Surveys*, vol. 17, pp. 419-470, Dec. 1985.

8. Mullender, S.J., and Tanenbaum, A.S.: "A Distributed File Service Based on Optimistic Concurrency Control," *Proc. Tenth Symp. Oper. Syst. Prin.*, pp. 51-62, 1985.

9. Mullender, S.J., and Tanenbaum, A.S.: "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, pp. 421-432, Oct. 1984.

10. Mullender, S.J., Rossum, G. van, Tanenbaum, A.S., Renesse, R. van, Staveren, H. van: "Amoeba—A Distributed Operating System for the 1990s," *IEEE Computer Magazine*, May 1990.

11. Tanenbaum, A.S., Renesse, R. van, Staveren, H. van, Sharp, G.J., Mullender S.J., Jansen, A.J., and Rossum, G. van: "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM*.