8

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.

8.1. REBUILDING MINIX ON THE 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*:

center allbox; l l. **Directory Contents** include The headers used by the commands (has two subdirectories) kernelProcess, 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.

8.1.1. 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.

8.1.2. 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
```

8.1.3. 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.

8.1.4. 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*.

8.1.5. 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.

8.2. 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.

8.2.1. 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 =
```

8.2.2. 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!

8.2.3. 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 Sec. 3.12 if your new boot disk does not function properly.

8.2.4. 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.

8.2.5. 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*.

8.2.6. 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*.

8.3. 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.

8.4. REBUILDING MINIX ON THE 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*.

8.4.1. 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*.

8.4.2. 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.

8.4.3. 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.

8.4.4. 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.

8.4.5. 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 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.

8.4.6. Installing New Device Drivers

Follow the procedure outlined in the IBM PC section.

8.4.7. 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.

8.5. REBUILDING MINIX ON THE SUN SPARCSTATION 1

It is possible to rebuild MINIX-SPARC on any SparcStation with at least 4 MB of main memory and a hard disk. Some hints are given to rebuild MINIX-SPARC on a SparcStation with only a diskette drive and at least 8 MB of RAM.

8.5.1. Configuring the System

In order to rebuild MINIX-SPARC 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.

If you want to reconfigure the system you should first examine the files *include/minix/config.h* and *include/minix/boot.h*. These contain a number of tunable system parameters. For instance if you keep your root partition on */dev/sd15*, 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 + 15)
```

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.

The file *config.h* contains some user-settable parameters. Examine this file and make the changes you require. The macro *MACHINE* should be *SUN_4* on the SparcStation. You should change the value of *NR_BUFS* to a bigger value, like 1536, but only if your SparcStation has at least 8 MB of main memory. The file system process gets only a maximum of 1792 KB of memory, in which the FS code, data and stack should fit. So do not make the FS buffer cache too big or else the system will crash.

8.5.2. 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:

```
cd /usr/src/kernel
make
```

```
cd /usr/src/mm
make
cd /usr/src/fs
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 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/rfd0
```

If the diskette was not yet formatted, you have to *fdformat* it first. If you want to boot MINIX from hard disk, you should copy the MINIX-SPARC boot image to a previously prepared hard disk partition. To copy the image, construct a shell sript containing the following line:

```
dd if=/usr/src/tools/minix.img of=/dev/sdn skip=1 seek=1 conv=sync
```

where *n* is the ASCII representation of the minor partition number. It is wise to triple check this command when you have typed it, as it writes on the given hard disk partition without checking whether there was already a MINIX or SunOS filesystem on it. The point of putting it in a shell script is that you can examine it carefully after typing it. If you just type it in and make a mistake, you can wipe out your hard disk. After you are convinced that it is correct, execute the shell script.

Now you can logout and reboot the system to try your new boot disk. 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.

8.5.3. Rebuilding MINIX Using a Floppy Diskette Drive

It is possible to rebuild MINIX-SPARC on a SparcStation with at least 8 MB of main memory but without a hard disk. This section gives hints on how to create an environment to rebuild MINIX without using a hard disk, but is not quite exhaustive in explaining everything step-by-step. When you want to rebuild MINIX with only a diskette, you have to be creative.

To create the environment, you should format four 1.44M diskettes. The first diskette will be used for a big 4MB root filesystem, which should contain the normal root filesystem files plus the C compiler, editor, include files, etc. The second will be used for the kernel, the third for the memory manager, the file system and tools sources. The last diskette will be the new boot diskette. You should create a ROOT file system of 4 MB. A 3MB filesystem should do as well but that is the minimum. This is how you should create the ROOT file system. First reboot MINIX. Then log in as *root* and type:

```
for i in ar cpdir df rmdir mkfs fdformat chmod compress
do cp /usr/bin/$i /bin; done
/etc/umount /dev/fd0
```

Next, insert a new diskette, format it, create a MINIX file system on it and copy the ROOT file system to it:

```
fdformat
mkfs –t /dev/fd0 4096
/etc/mount /dev/fd0 /user
cpdir –msv / /user
/etc/umount /dev/fd0
```

Reboot MINIX again with the new ROOT diskette. To compile programs, you will need *cc, as, ld, /usr/include, cpp, cc1* and *libc.a*. You'll probably also want a program editor and *make*.

```
/etc/umount /dev/fd0          # unmount the /usr/disk
/etc/mount /dev/fd0 /user     # insert 04.USR2
cp /user/bin/elvis /bin/vi    # copy elvis (or mined)
cp /user/bin/make /bin
/etc/umount /dev/fd0
/etc/mount /dev/fd0 /user     # insert 05.CSYS
cp /user/bin/* /bin           # get cc, as and ld
cd /bin; compress –d *.Z; rm *.Z
mkdir /usr/lib; cd /usr/lib   # create the lib directory
cp /user/lib/[cehl]* .
compress –d *.Z; rm *.Z
cpdir /user/include /usr/include; cd /usr/include
compress –d *.Z; rm *.Z; ar x *.a; rm *.a
(cd minix; compress –d *.Z; rm *.Z; ar x *.a; rm *.a)
(cd sys; compress –d *.Z; rm *.Z; ar x *.a; rm *.a)
mkdir /usr/tmp                # vi and the C compiler want /usr/tmp
mkdir /usr/src
```

You could copy your current ram device to the root diskette, but you should make sure, that it fits (i.e., no more than 1440 used blocks on */dev/ram*; check this with *df*).

Now, you need to copy and unpack the kernel sources:

```
/etc/umount /dev/fd0          # unmount 05.CSYS
```

```
/etc/mount  /dev/fd0  /user   # insert 07.SRC2
cp  /user/src/kernel/*  /tmp  # temporarily store kernel sources in tmp
/etc/umount  /dev/fd0         # unmount diskette and insert empty diskette
fdformat                      # format the diskette
mkfs  /dev/fd0  1440          # create an empty filesystem
/etc/mount  /dev/fd0  /usr/src
mkdir  /usr/src/kernel        # create kernel directory on the diskette
cp  /tmp/*  /usr/src/kernel   # copy the kernel files to it
cd  /usr/src/kernel;  compress  –d  *.Z;  rm  *.Z;  ar  x  *.a;  rm  *.a
cd  /;  /etc/umount  /dev/fd0 # remove the diskette
rm  /tmp/*                    # and the temporary files
```

The same needs to be done for the MM/FS/tools diskette:

```
/etc/mount  /dev/fd0  /user   # insert 07.SRC2
for  i  in  mm  fs  tools;  do  cpdir  –mv  /user/src/$i  /tmp/$i;  done
/etc/umount /dev/fd0          # unmount 07.SRC2 and insert empty diskette
fdformat                      # format the diskette
mkfs  /dev/fd0  1440          # create an empty filesystem
/etc/mount  /dev/fd0  /usr/src
for  i  in  mm  fs  tools
do  cpdir  –mv  /tmp/$i  /usr/src/$i;  cd  /usr/src/$i
compress  –d  *.Z;  rm  *.Z;  ar  x  *.a;  rm  *.a;  rm  –rf  /tmp/$i
done
mkdir  /usr/src/kernel
/etc/umount  /dev/fd0         # remove the diskette
```

Now, you can rebuild the MINIX-SPARC kernel in three phases. First recompile the kernel, then recompile MM and FS and merge them producing a new bootable MINIX image and finally copy this image to a diskette. To start, insert the *kernel* diskette, you just created:

```
/etc/mount  /dev/fd0  /usr/src       # mount the kernel diskette
cd  /usr/src/kernel;  make
cp  /usr/src/kernel/kernel.out  /tmp
```

The last command copies the linked kernel image to the RAM device. Now unmount the *kernel* diskette, replace it with the diskette, containing the *mm*, *fs* and *tools*, and mount it. Compile the sources:

```
/etc/umount  /dev/fd0         # unmount the kernel diskette
/etc/mount  /dev/fd0  /usr/src       # mount the mm/fs/tools diskette
cd  /usr/src/mm;  make               # make mm.out
cd  /usr/src/fs;  make        # make fs.out
cp  /tmp/kernel.out  /usr/src/kernel
cd  /usr/src/tools
```

```
make                    # make minix.img
cp  minix.img  /tmp
cd  /tmp
/etc/umount  /dev/fd0
```

Remove the diskette and insert an empty, formatted diskette. Copy the MINIX image to it:

```
cp  minix.img  /dev/rfd0
```

If space is getting scarce, remove as much temporary and '*.o*' files as necessary.

8.5.4. 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 *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*.

Writing device drivers for MINIX-SPARC is somewhat more difficult than writing device drivers for the PC or Atari versions of MINIX. The first problem is that the internal hardware of the SparcStation is not well documented. Secondly, the SparcStation 1, 1+ and IPC use an MMU with contexts and a cache. A context is a translation table of virtual to physical addresses. Of the 8 available contexts, MINIX-SPARC uses only the first 3 contexts. The MINIX memory manager runs at virtual address 0 in context 0, the file system at virtual address 0 in context 1 and the currently (or last) running user process is mapped at virtual address 0 in context 2. The kernel is mapped in **all** contexts in high virtual memory at address 0xE0004000. When a device driver wants to access data in the memory of another process, it has to map that memory in its own address space. But by double mapping of the same physical memory, the cache of the SparcStation could present problems. Therefore, a device driver should carefully flush the cache. If you want to write a device driver, study the present device drivers closely, until you understand them. The file */usr/src/kernel/const.h* gives information on the Sparc and SparcStation specific constants.

8.5.5. 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* 60000 bytes of memory by issuing the command:

```
chmem =60000 /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*. Especially the GNU C-compiler proper (*/usr/lib/cc1*) is known to require a gap of 1200000 bytes to compile some programs. When compiling the libraries via

```
cc –O –D_MINIX –D _POSIX_SOURCE –c *.c
```

the dispatcher, *cc*, will need more memory.

8.5.6. Recompiling GNU Sources

Rebuilding the GNU programs is somewhat different from rebuilding the ordinary MINIX commands. This is mainly due to the size and the form of distribution of the GNU sources.

Some gcc and gas source files are so big, that */usr/lib/cc1* and */usr/bin/as* need more memory space to compile/assemble them. When compiling gcc or gas, it's best to temporarily increase the gap size of */usr/lib/cc1* to 1600000, and the gap size of */usr/bin/as* to 800000. This will be enough to compile the most demanding sources. To link the assembler object files, */usr/bin/ld* needs a slightly bigger gap of 150000 bytes. You should restore the gaps to their original sizes after you've finished recompiling gcc and gas to preserve memory space.

Distributed are original GNU source files, stored in the *Setup.dir* directories, and the changes made to some source files in *cdiff* files in the *Setup.fix* directories. There are three GNU source directories, *gcc*, *gas* and *binutils* in the */usr/src/gnu* directory.

To compile, for instance, *gcc*, you should have installed the GNU sources via */usr/setup_usr*. You also should have booted MINIX-SPARC with a root file system, big enough to hold the temporary files from *patch*. The distribution 02.ROOT is not big enough. Use a root of at least 768 KB, created as shown in Step 4 of section 6.3.4. Log in and type:

```
chmem =1600000 /usr/lib/cc1
chmem =800000 /usr/bin/as
cd /usr/src/gnu/gcc
make
```

Now, the MINIX-SPARC sources of the GNU C compiler are extracted and compiled, which may take a while, even on a fast computer like the SparcStation. The new *cc*, *cpp* and *cc1* can be tested and copied to */usr/bin* and */usr/lib* if they're all right. Afterwards, you should *chmem cc1* and *as* to their original gap sizes. If you do not, both of them will allocate an enormous amount of memory when run, which is almost never used.