runs the program *a.out* but discards its output. The RAM disk driver effectively treats this minor device as having zero size, so no data are ever copied to or from it. If you read from it you will get an immediate EOF (End of File).

If you have looked at the directory entries for these files in */dev/* you may have noticed that, of those mentioned so far, only */dev/ram* is a block special file. All the others are character devices. There is one more block device supported by the memory driver. This is */dev/boot*. From the point of view of the device driver it is another block device implemented in RAM, just like */dev/ram*. However, it is meant to be initialized by copying a file appended to the boot image after *init* into memory, rather than starting with an empty block of memory, as is done for */dev/ram*. Support for this device is provided for future use and it is not used in MINIX 3 as described in this text.

Finally, the last device supported by the memory driver is another character special file, */dev/zero*. It is sometimes convenient to have a source of zeros. Writing to */dev/zero* is like writing to */dev/null*; it throws data away. But reading */dev/zero* gives you zeros, in any quantity you want, whether a single character or a disk full.

At the driver level, the code for handling */dev/ram*, */dev/mem*, */dev/kmem*, and */dev/boot* is identical. The only difference among them is that each one corresponds to a different region of memory, indicated by the arrays *ram_origin* and *ram_limit*, each indexed by minor device number. The file system manages devices at a higher level. The file system interprets devices as character or block devices, and thus can mount */dev/ram* and */dev/boot* and manage directories and files on these devices. For the devices defined as character devices the file system can only read and write streams of data (although a stream read from */dev/null* gets only EOF).

3.6.3 Implementation of the RAM Disk Driver in MINIX 3

As with other disk drivers, the main loop of the RAM disk driver is in the file *driver.c*. The device-specific support for memory devices is in *memory.c* (line 10800). When the memory driver is compiled, a copy of the object file called *drivers/libdriver/driver.o*, produced by compiling *drivers/libdriver/driver.c*, is linked with the object file *drivers/memory/memory.o*, the product of compiling *drivers/memory/memory.c*.

It may be worth taking a moment to consider how the main loop is compiled. The declaration of the *driver* structure in *driver.h* (lines 10829 to 10845) defines a data structure, but does not create one. The declaration of *m_dtab* on lines 11645 to 11660 creates an instance of this with each part of the structure filled in with a pointer to a function. Some of these functions are generic code compiled when *driver.c* is compiled, for instance, all of the *nop* functions. Others are code compiled when *memory.c* is compiled, for instance, *m_do_open*. Note that for the memory driver seven of the entries are do-little or do-nothing routines and the last two are defined as *NULL* (which means these functions will never be called, there is no need even for a *do_nop*). All this is a sure clue that the operation of a RAM disk is not terribly complicated.

The memory device does not require definition of a large number of data structures, either. The array *m_geom[NR_DEVS]* (line 11627) holds the base and size of each of the six memory devices in bytes, as 64 bit unsigned integers, so there is no immediate danger of MINIX 3 not being able to have a big enough RAM disk. The next line defines an interesting structure that will not be seen in other drivers. *M_seg[NR_DEVS]* is apparently just an aray of integers, but these integers are indices that allow segment descriptors to be found. The memory device driver is unusual among user-space processes in having the ability to access regions of memory outside of the ordinary text, data, and stack segments every process owns. This array holds the information that allows access to the designated additional memory regions. The variable *m_device* just holds the index into these arrays of the currently active minor device.

To use */dev/ram* as the root device the memory driver must be initialized very early during startup of MINIX 3. The *kinfo* and *machine* structures that are defined next will hold data retrieved from the kernel during startup that is necessary for initializing the memory driver.

One other data structure is defined before the executable code begins. This is *dev_zero*, an array of 1024 bytes, used to supply data when a read call is made to */dev/zero*.

The main procedure *main* (line 11672) calls one function to do some local initialization. After that, it calls the main loop, which gets messages, dispatches to the appropriate procedures, and sends the replies. There is no return to *main* upon completion.

The next function, *m_name*, is trivial. It returns the string "memory" when called.

On a read or write operation, the main loop makes three calls: one to prepare a device, one to do the actual data transfer, and one to do cleanup. For a memory device, a call to *m_prepare* is the first of these. It checks that a valid minor device has been requested and then returns the address of the structure that holds the base address and size of the requested RAM area. The second call is for *m_transfer* (line 11706). This does all the work. As we saw in *driver.c*, all calls to read or write data are transformed into calls to read or write multiple contiguous blocks of data—if only one block is needed the request is passed on as a request for multiple blocks with a count of one. So only two kinds of transfer requests are passed on to the driver, *DEV_GATHER*, requesting a read of one or more blocks, and *DEV_SCATTER*, a request to write one or more blocks. Thus, after getting the minor device number, *m_transfer* enters a loop, repeated for the number of transfers requested. Within the loop there is a switch on the device type.

The first case is for */dev/null*, and the action is to return immediately on a *DEV_GATHER* request or on a *DEV_SCATTER* request to fall through to the end of the switch. This is so the number of bytes transferred (although this number is zero for */dev/null*) can be returned, as would be done for any write operation.

For all of the device types that refer to real locations in memory the action is similar. The requested offset is checked against the size of the device to determine that the request is within the bounds of the memory allocated to the device. Then a kernel call is made to copy data either to or from the memory of the caller. There are two chunks of code that do this, however. For */dev/ram*, */dev/kmem*, and */dev/boot* virtual addresses are used, which requires retrieving the segment address of the memory region to be accessed from the *m_seg* array, and then making a sys_vircopy kernel call (lines 11640 to 11652). For */dev/mem* a physical address is used and the call is to sys_physcopy.

The remaining operation is a read or write to */dev/zero*. For reading the data is taken from the *dev_zero* array mentioned earlier. You might ask, why not just generate zero values as needed, rather than copying from a buffer full of them? Since the copying of the data to its destination has to be done by a kernel call, such a method would require either an inefficient copying of single bytes from the memory driver to the system task, or building code to generate zeros into the system task. The latter approach would increase the complexity of kernel-space code, something that we would like to avoid in MINIX 3.

A memory device does not need a third step to finish a read or write operation, and the corresponding slot in *m_dtab* is a call to *nop_finish*.

Opening a memory device is done by *m_do_open* (line 11801). The job is done by calling *m_prepare* to check that a valid device is being referenced. More interesting than the code that exists is a comment about code that was found here in older versions of MINIX. Previously a trick was hidden here. A call by a user process to open */dev/mem* or */dev/kmem* would also magically confer upon the caller the ability to execute instructions which access I/O ports. Pentium-class CPUs implement four privilege levels, and user processes normally run at the least-privileged level. The CPU generates a general protection exception when an process tries to execute an instruction not allowed at its privilege level. Providing a way to get around this was considered safe because the memory devices could only be accessed by a user with root privileges. In any case, this possibly risky "feature" is absent from MINIX 3 because kernel calls that allow I/O access via the system task are now available. The comment remains, to point out that if MINIX 3 is ported to hardware that uses memory-mapped I/O such a feature might need to be reintroduced. The function to do this, *enable_iop*, remains in the kernel code to show how this can be done, although it is now an orphan.

The next function, *m_init* (line 11817), is called only once, when *mem_task* is called for the first time. This routine uses a number of kernel calls, and is worth study to see how MINIX 3 drivers interact with kernel space by using system task services. First a sys_getkinfo kernel call is made to get a copy of the kernel's *kinfo* data. From this data it copies the base address and size of */dev/kmem* into the corresponding fields of the *m_geom* data structure. A different kernel call, sys_segctl, converts the physical address and size of */dev/kmem* into the segment descriptor information needed to treat the kernel memory as a virtual memory space. If an image of a boot device has been compiled into the system boot image, the field for the base address of */dev/boot* will be non-zero. If this is so, then information to access the memory region for this device is set up in exactly the same way it was done for */dev/kmem*. Next the array used to supply data when */dev/zero* is accessed is explicitly filled with zeros. This is probably unnecessary; C compilers are supposed to initialize newly created static variables to all zeros.

Finally, *m_init* uses a sys_getmachine kernel call to get another set of data from the kernel, the *machine* structure which flags various possible hardware alternatives. In this case the information needed is whether or not the CPU is capable of protected mode operation. Based on this information the size of */dev/mem* is set to either 1 MB, or 4 GB − 1, depending upon whether MINIX 3 is running in 8088 or 80386 mode. These sizes are the maximum sizes supported by MINIX 3 and do not have anything to do with how much RAM is installed in the machine. Only the size of the device is set; the compiler is trusted to set the base address correctly to zero. Also, since */dev/mem* is accessed as physical (not virtual) memory there is no need to make a sys_segctl kernel call to set up a segment descriptor.

Before we leave *m_init* we should mention another kernel call used here, although it is not obvious in the code. Many of the actions taken during initialization of the memory driver are essential to proper functioning of MINIX 3, and thus several tests are made and *panic* is called if a test fails. In this case *panic* is a library routine which ultimately results in a sys_exit kernel call. The kernel and (as we shall see) the process manager and the file system have their own *panic* routines. The library routine is provided for device drivers and other small system components.

Surprisingly, the function we just examined, *m_init*, does not initialize the quintessential memory device, */dev/ram*. This is taken care of in the next function, *m_ioctl* (line 11863). In fact, there is only one ioctl operation defined for the RAM disk; this is *MIOCRAMSIZE*, which is used by the file system to set the RAM disk size. Much of the job is done without requiring any services from the kernel. The call to allocmem on line 11887 is a system call, but not a kernel call. It is handled by the process manager, which maintains all of the information necessary to find an available region of memory. However, at the end one kernel call is needed. At line 11894 a sys_segctl call is made to convert the physical address and size returned by allocmem into the segment information needed for further access.

The last function defined in *memory.c* is *m_geometry*. This is a fake. Obviously, cylinders, heads, and sectors are irrelevant in addressing semiconductor memory, but if a request is made for such information for a memory device this function pretends it has 64 heads and 32 sectors per track, and calculates from the size how many cylinders there are.

3.7 DISKS

All modern computers except embedded ones have disk drives. For that reason, we will now study them, starting with the hardware, then moving on to say some general things about disk software. After that we will delve into the way MINIX 3 controls its disks.

3.7.1 Disk Hardware

All real disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on some hard disks. The simplest designs have the same number of sectors on each track. All sectors contain the same number of bytes, although a little thought will make it clear that sectors close to the outer rim of the disk will be physically longer than those close to the hub. The time to read or write each sector will be same, however. The data density is obviously higher on the innermost cylinders, and some disk designs require a change in the drive current to the read-write heads for the inner tracks. This is handled by the disk controller hardware and is not visible to the user (or the implementer of an operating system).

The difference in data density between inner and outer tracks means a sacrifice in capacity, and more sophisticated systems exist. Floppy disk designs that rotate at higher speeds when the heads are over the outer tracks have been tried. This allows more sectors on those tracks, increasing disk capacity. Such disks are not supported by any system for which MINIX 3 is currently available, however. Modern large hard drives also have more sectors per track on outer tracks than on inner tracks. These are **IDE (Integrated Drive Electronics)** drives, and the sophisticated processing done by the drive's built-in electronics masks the details. To the operating system they appear to have a simple geometry with the same number of sectors on each track.

The drive and controller electronics are as important as the mechanical hardware. The main element of the disk controller is a specialized integrated circuit, really a small microcomputer. Once this would have been on a card plugged into the computer's backplane, but on modern systems, the disk controller is on the parentboard. For a modern hard disk this disk controller circuitry may be simpler than for a floppy disk, since a hard drive has a powerful electronic controller integrated into the drive itself.

A device feature that has important implications for the disk driver is the possibility of a controller doing seeks on two or more drives at the same time. These are known as **overlapped seeks**. While the controller and software are waiting for a seek to complete on one drive, the controller can initiate a seek on another drive. Many controllers can also read or write on one drive while seeking on one