requires that the door be unlocked and the CD-ROM ejected. There are other complications of removable media that are more applicable to floppy drives, so we will discuss these in a later section. For CD-ROMs a *DEV_IOCTL* operation is used to set a flag to mark that the medium should be ejected from the drive upon a *DEV_CLOSE*. A *DEV_IOCTL* operation is also used to read and write partition tables.

DEV_READ, *DEV_WRITE*, *DEV_GATHER* and *DEV_SCATTER* requests are each handled in two phases, prepare and transfer, as we saw previously. For the hard disk *DEV_CANCEL* and *DEV_SELECT* calls are ignored.

No scheduling is done by the hard disk device driver at all, that is done by the file system, which assembles the vector requests for gather/scatter I/O. Requests come from the file system cache as *DEV__GATHER* or *DEV_SCATTER* requests for multiples of blocks (4-KB in the default configuration of MINIX 3), but the hard disk driver is able to handle requests for any multiple of a sector (512 bytes). In any case, as we have seen, the main loop of all disk drivers transforms requests for single blocks of data into one element vector requests.

Requests for reading and writing are not mixed in a vector of requests, nor can requests be marked as optional. The elements of a request vector are for contiguous disk sectors, and the vector is sorted by the file system before being passed to the device driver, so it suffices to specify just the starting position on the disk for an entire array of requests.

The driver is expected to succeed in reading or writing at least the first request in a request vector, and to return when a request fails. It is up to the file system to decide what to do; the file system will try to complete a write operation but will return to the calling process only as much data as it can get on a read.

The file system itself, by using scattered I/O, can implement something similar to Teory's version of the elevator algorithm—recall that in a scattered I/O request the list of requests is sorted on the block number. The second step in scheduling takes place in the controller of a modern hard disk. Such controllers are ''smart'' and can buffer large quantities of data, using internally programmed algorithms to retrieve data in the most efficient order, irrespective of the order of receipt of the requests.

3.7.5 Implementation of the Hard Disk Driver in MINIX 3

Small hard disks used on microcomputers are sometimes called ''winchester'' disks. The term was IBM's code name for the project that developed the disk technology in which the read/write heads fly on a thin cushion of air and land on the recording medium when the disk stops spinning. The explanation of the name is that an early model had two data modules, a 30-Mbyte fixed and a 30-Mbyte removable one. Supposedly this reminded the developers of the Winchester 30-30 firearm which figures in many tales of the United States' western frontier. Whatever the origin of the name, the basic technology remains the same, although

today's typical PC disk is much smaller and the capacity is much larger than the 14-inch disks that were typical of the early 1970s when the winchester technology was developed.

The MINIX 3 AT-style hard disk driver is in *at_wini.c* (line 12100). This is a complicated driver for a sophisticated device, and there are several pages of macro definitions specifying controller registers, status bits and commands, data structures, and prototypes. As with other block device drivers, a *driver* structure, *w_dtab* (lines 12316 to 12331), is initialized with pointers to the functions that actually do the work. Most of them are defined in *at_wini.c*, but as the hard disk requires no special cleanup operation, its *dr_cleanup* entry points to the common *nop_cleanup* in *driver.c*, shared with other drivers that have no special cleanup requirement. Several other possible functions are also irrelevant for this driver and also are initialized to point to *nop_* functions. The entry function, called *at_winchester_task* (line 12336), calls a procedure that does hardware-specific initialization and then calls the main loop in *driver.c*, passing the address of *w_dtab*. The main loop, *driver_task* in *libdriver/driver.c*, runs forever, dispatching calls to the various functions pointed to by the *driver* table.

Since we are now dealing with real electromechanical storage devices, there is a substantial amount of work to be done by *init_params* (line 12347) to initialize the hard disk driver. Various parameters about the hard disks are kept in the *wini* table defined on lines 12254 to 12276, which has an element for each of the *MAX_DRIVES* (8) drives supported, up to four conventional IDE drives, and up to four drives on the PCI bus, either plug-in IDE controllers or **SATA** (**Serial AT Attachment**) controllers.

Following the policy of postponing initialization steps that could fail until the first time they are truly necessary, *init_params* does not do anything that requires accessing the disk devices themselves. The main thing it does is to copy information about the hard disk logical configuration into the *wini* array. The ROM BIOS on a Pentium-class computer retrieves basic configuration information from the CMOS memory used to preserve basic configuration data. The BIOS does this when the computer is first turned on, before the first part of the MINIX 3 loading process begins. On lines 12366 to 12392 the information is copied from the BIOS. Many of the constants used here, such as *NR_HD_DRIVES_ADDR* are defined in *include/ibm/bios.h*, a file which is not listed in Appendix B but which can be found on the MINIX 3 CD-ROM. It is not necessarily fatal if this information cannot be retrieved. If the disk is a modern one, the information can be retrieved directly from the disk when it is accessed for the first time. Following the entry of data obtained from the BIOS, additional disk information is filled in for each drive using a call to the next function, *init_drive*.

On older systems with IDE controllers, the disk functions as if it were an AT-style peripheral card, even though it may be integrated on the parentboard. Modern drive controllers usually function as PCI devices, with a 32-bit data path to the CPU, rather than the 16-bit AT bus. Fortunately for us, once initialization

is complete, the interface to both generations of disk controller appears the same to the programmer. To make this work, *init_params_pci* (line 12437) is called if necessary to get the parameters of the PCI devices. We will not describe the details of this routine, but a few points should be mentioned. First, the boot parameter *ata_instance* is used on line 12361 to set the value of the variable *w_instance*. If the boot parameter is not explicitly set the value will be zero. If it is set and greater than zero the test on line 12365 causes querying the BIOS and initialization of standard IDE drives to be skipped. In this case only drives found on the PCI bus will be registered.

The second point is that a controller found on the PCI bus will be identified as controlling devices *c0d4* through *c0d7*. If *w_instance* is non-zero the drive identifiers *c0d0* through *c0d3* will be skipped, unless a PCI bus controller identifies itself as "compatible." Drives handled by a compatible PCI bus controller will be designated *c0d0* through *c0d3*. For most MINIX 3 users all of these complications can probably be ignored. A computer with less than four drives (including the CD-ROM drive), will most likely appear to the user to have the classical configuration, with drives designated *c0d0* to *c0d3*, whether they are connected to IDE or PCI controllers, and whether or not they use the classic 40-pin parallel connectors or the newer serial connectors. But the programming required to create this illusion is complicated.

After the call to the common main loop, nothing may happen for a while until the first attempt is made to access the hard disk. When the first attempt to access a disk is made a message requesting a *DEV_OPEN* operation will be received by the main loop and *w_do_open* (line 12521) will be indirectly called. In turn, *w_do_open* calls *w_prepare* to determine if the device requested is valid, and then *w_identify* to identify the type of device and initialize some more parameters in the *wini* array. Finally, a counter in the *wini* array is used to test whether this is first time the device has been opened since MINIX 3 was started. After being examined, the counter is incremented. If it is the first *DEV_OPEN* operation, the *partition* function (in *drvlib.c*) is called.

The next function, *w_prepare* (line 12577), accepts an integer argument, *device*, which is the minor device number of the drive or partition to be used, and returns a pointer to the *device* structure that indicates the base address and size of the device. In the C language, the use of an identifier to name a structure does not preclude use of the same identifier to name a variable. Whether a device is a drive, a partition, or a subpartition can be determined from the minor device number. Once *w_prepare* has completed its job, none of the other functions used to read or write the disk need to concern themselves with partitioning. As we have seen, *w_prepare* is called when a *DEV_OPEN* request is made; it is also one phase of the prepare/transfer cycle used by all data transfer requests.

Software-compatible AT-style disks have been in use for quite a while, and *w_identify* (line 12603) has to distinguish between a number of different designs that have been introduced over the years. The first step is to see that a readable

and writeable I/O port exists where one should exist on all disk controllers in this family. This is the first example we have seen of I/O port access by a user-space driver, and the operation merits a description. For a disk device I/O is done using a *command* structure, defined on lines 12201 to 12208, which is filled in with a series of byte values. We will describe this in a bit more detail later; for the moment note that two bytes of this structure are filled in, one with a value *ATA_IDENTIFY*, interpreted as a command that asks an **ATA** (**AT Attached**) drive to identify itself, and another with a bit pattern that selects the drive. Then *com_simple* is called.

This function hides all the work of constructing a vector of seven I/O port addresses and bytes to be written to them, sending this information to the system task, waiting for an interrupt, and checking the status returned. This tests that the drive is alive and allows a string of 16-bit values to be read by the sys_insw kernel call on line 12629. Decoding this information is a messy process, and we will not describe it in detail. Suffice it to say that a considerable amount of information is retrieved, including a string that identifies the model of the disk, and the preferred physical cylinder, head, and sector parameters for the device. (Note that the "physical" configuration reported may not be the true physical configuration, but we have no alternative to accepting what the disk drive claims.) The disk information also indicates whether or not the disk is capable of **Logical Block Addressing** (**LBA**). If it is, the driver can ignore the cylinder, head, and sector parameters and can address the disk using absolute sector numbers, which is much simpler.

As we mentioned earlier, it is possible that *init_params* may not recover the logical disk configuration information from the BIOS tables. If that happens, the code at lines 12666 to 12674 tries to create an appropriate set of parameters based on what it reads from the drive itself. The idea is that the maximum cylinder, head, and sector numbers can be 1023, 255, and 63 respectively, due to the number of bits allowed for these fields in the original BIOS data structures.

If the *ATA_IDENTIFY* command fails, it may simply mean that the disk is an older model that does not support the command. In this case the logical configuration values previously read by *init_params* are all we have. If they are valid, they are copied to the physical parameter fields of *wini*; otherwise an error is returned and the disk is not usable.

Finally, MINIX 3 uses a *u32_t* variable to count addresses in bytes. This limits the size of a partition to 4 GB. However, the *device* structure used to record the base and size of a partition (defined in *drivers/libdriver/driver.h* on lines 10856 to 10858) uses *u64_t* numbers, and a 64 bit multiplication operation is used to calculate the size of the drive on (line 12688), and the base and size of the whole drive are then entered into the *wini* array, and *w_specify* is called, twice if necessary, to pass the parameters to be used back to the disk controller (line 12691). Finally, more kernel calls are made: a sys_irqsetpolicy call (line 12699) ensures that when a disk controller interrupt occurs and is serviced the interrupt

will be automatically reenabled in preparation for the next one. Following that, a sys_irqenablecall actually enables the interrupt.

W_name (line 12711) returns a pointer to a string containing the device name, which will be either "AT-D0," "AT-D1" "AT-D2," or "AT-D3." When an error message must be generated this function tells which drive produced it.

It is possible that a drive will turn out to be incompatible with MINIX 3 for some reason. The function *w_io_test* (line 12723) is provided to test each drive the first time an attempt is made to open it. This routine tries to read the first block on the drive, with shorter timeout values than are used in normal operation. If the test fails the drive is permanently marked as unavailable.

W_specify (line 12775), in addition to passing the parameters to the controller, also recalibrates the drive (if it is an older model), by doing a seek to cylinder zero.

Do_transfer (line 12814) does what its name implies, it assembles a *command* structure with all the byte values needed to request transfer of a chunk of data (possibly as many as 255 disk sectors), and then it calls *com_out*, which sends the command to the disk controller. The data must be formatted differently depending upon how the disk is to be addressed, that is, whether by cylinder, head, and sector or by LBA. Internally MINIX 3 addresses disk blocks linearly, so if LBA is supported the first three byte-wide fields are filled in by shifting the sector count an appropriate number of bits to the right and then masking to get 8-bit values. The sector count is a 28 bit number, so the last masking operation uses a 4-bit mask (line 12830). If the disk does not support LBA then cylinder, head, and sector values are calculated, based on the parameters of the disk in use (lines 12833 to 12835).

The code contains a hint of a future enhancement. LBA addressing with a 28-bit sector count limits MINIX 3 to fully utilizing disks of 128 GB or smaller size. (You can use a bigger disk, but MINIX 3 can only access the first 128 GB). The programmers have been thinking about, but have not yet implemented, use of the newer **LBA48** method, which uses 48 bits to address disk blocks. On line 12824 a test is made for whether this is enabled. The test will always fail with the version of MINIX 3 described here. This is good, because no code is provided to be executed if the test succeeds. Keep in mind if you decide to modify MINIX 3 yourself to use LBA48 that you need to do more than just add some code here. You will have to make changes in many places to handle the 48-bit addresses. You might find it easier to wait until MINIX 3 has been ported to a 64-bit processor, too. But if a 128 GB disk is not big enough for you, LBA48 will give you access to 128 PB (Petabytes).

Now we will briefly look at how a data transfer takes place at a higher level. *W_prepare*, which we have already discussed, is called first. If the transfer operation requested was for multiple blocks (that is, a *DEV_GATHER* or *DEV_SCATTER* request), *w_transfer* line 12848 is called immediately afterward. If the transfer is for a single block (a *DEV_READ* or *DEV_WRITE* request), a one

element scatter/gather vector is created, and then *w_transfer* is called. Accordingly, *w_transfer* is written to expect a vector of *iovec_t* requests. Each element of the request vector consists of a buffer address and the size of the buffer, constrained that the size must be a multiple of the size of a disk sector. All other information needed is passed as an argument to the call, and applies to the entire request vector.

The first thing done is a simple test to see if the disk address requested for the start of the transfer is aligned on a sector boundary (line 12863). Then the outer loop of the function is entered. This loop repeats for each element of the request vector. Within the loop, as we have seen many times before, a number of tests are made before the real work of the function is done. First the total number of bytes remaining in the request is calculated by summing the *iov_size* fields of each element of the request vector. This result is checked to be sure it is an exact multiple of the size of a sector. Other tests check that the starting position is not at or beyond the end of the device, and if the request would end past the end of the device the size of the request is truncated. All calculations so far have been in bytes, but on line 12876 a calculation is made of the block position on the disk, using 64 bit arithmetic. Note that although the variable used is named *block*, this is a number of disk blocks, that is, 512 byte sectors, not the "block" used internally by MINIX 3, normally 4096 bytes. After this one more adjustment is made. Every drive has a maximum number of bytes that can be requested at one time, and the request is scaled back to this quantity if necessary. After verifying that the disk has been initialized, and doing so again if necessary, a request for a chunk of data is made by calling *do_transfer* (line 12887).

After a transfer request has been made the inner loop is entered, which repeats for each sector. For a read or write operation an interrupt will be generated for each sector. On a read the interrupt signifies data is ready and can be transferred. The *sys_insw* kernel call on line 12913 asks the system task to read the specified I/O port repeatedly, transferring the data to a virtual address in the data space of the specified process. For a write operation the order is reversed. The *sys_outsw* call a few lines further down writes a string of data to the controller, and the interrupt comes from the disk controller when the transfer to the disk is complete. In the case of either a read or a write, *at_intr_wait* is called to receive the interrupt, for example, on line 12920 following the write operation. Although the interrupt is expected, this function provides a way to abort the wait if a malfunction occurs and the interrupt never arrives. *At_intr_wait* also reads the disk controller's status register and returns various codes. This is tested on line 12933. On an error when either reading or writing, there is a break which skips over the section where results are recorded and poiners and counters adjusted for the next sector, so the next time through the inner loop will be a retry of the same sector, if another try is allowed. If the disk controller reports a bad sector *w_transfer* terminates immediately. For other errors a counter is incremented and the function is allowed to continue if *max_errors* has not been reached.

The next function we will discuss is *com_out*, which sends the command to the disk controller, but before we look at its code let us first look at the controller as it is seen by the software. The disk controller is controlled through a set of registers, which could be memory mapped on some systems, but on an IBM compatible appear as I/O ports. We will look at these registers and discuss a few aspects of how they (and I/O control registers in general) are used. In MINIX 3 there is the added complication that drivers run in user space and cannot execute the instructions that read or write registers. This will provide an opportunity to look at how kernel calls are used to work around this restriction.

The registers used by a standard IBM-AT class hard disk controller are shown in Fig. 3-23.

Register	Read Function	Write Function
0	Data	Data
1	Error	Write Precompensation
2	Sector Count	Sector Count
3	Sector Number (0-7)	Sector Number (0-7)
4	Cylinder Low (8-15)	Cylinder Low (8-15)
5	Cylinder High (16-23)	Cylinder High (16-23)
6	Select Drive/Head (24-27)	Select Drive/Head (24-27)
7	Status	Command

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Cylinder/Head/Sector Mode
1 = Logical Block Addressing Mode
D: 0 = master drive
1 = slave drive
HSn: CHS mode: Head select in CHS mode
LBA mode: Block select bits 24 - 27

(b)

Figure 3-23. (a) The control registers of an IDE hard disk controller. The numbers in parentheses are the bits of the logical block address selected by each register in LBA mode. (b) The fields of the Select Drive/Head register.

We have mentioned several times reading and writing to I/O ports, but we tacitly treated them just like memory addresses. In fact, I/O ports often behave differently from memory addresses. For one thing, input and output registers that

happen to have the same I/O port address are not the same register. Thus, the data written to a particular address cannot necessarily be retrieved by a subsequent read operation. For example, the last register address shown in Fig. 3-23 shows the status of the disk controller when read and is used to issue commands to the controller when written to. It is also common that the very act of reading or writing an I/O device register causes an action to occur, independently of the details of the data transferred. This is true of the command register on the AT disk controller. In use, data are written to the lower-numbered registers to select the disk address to be read from or written to, and then the command register is written last with an operation code. The data written to the command register determines what the operation will be. The act of writing the operation code into the command register starts the operation.

It is also the case that the use of some registers or fields in the registers may vary with different modes of operation. In the example given in the figure, writing a 0 or a 1 to the LBA bit, bit 6 of register 6, selects whether CHS (Cylinder-Head-Sector) or LBA (Logical Block Addressing) mode is used. The data written to or read from registers 3, 4, and 5, and the low four bits of register 6 are interpreted differently according to the setting of the LBA bit.

Now let us take a look at how a command is sent to the controller by calling *com_out* (line 12947). This function is called after setting up a *cmd* structure (with *do_transfer*, which we saw earlier). Before changing any registers, the status register is read to determine that the controller is not busy. This is done by testing the *STATUS_BSY* bit. Speed is important here, and normally the disk controller is ready or will be ready in a short time, so busy waiting is used. On line 12960 *w_waitfor* is called to test *STATUS_BSY*. *W_waitfor* uses a kernel call to ask the system task to read an I/O port so *w_waitfor* can test a bit in the status register. It loops until the bit is ready or until there is a timeout. The loop is programmed for a quick return when the disk is ready. Thus the returned value will be true with the minimum possible delay if the controller is ready, true after a delay if it is temporarily unavailable, or false if it is not ready after the timeout period. We will have more to say about the timeout when we discuss *w_waitfor* itself.

A controller can handle more than one drive, so once it is determined that the controller is ready, a byte is written to select the drive, head, and mode of operation (line 12966) and *w_waitfor* is called again. A disk drive sometimes fails to carry out a command or to properly return an error code—it is, after all, a mechanical device that can stick, jam, or break internally—and as insurance a sys_setalarm kernel call is made to have the system task schedule a call to a wakeup routine. Following this, the command is issued by first writing all the parameters to the various registers and finally writing the command code itself to the command register. This is done with a sys_voutb kernel call, which sends a vector of (*value*, *address*) pairs to the system task. The system task writes each *value* to the I/O port specified by the *address* in order. The vector of data for the

sys_voutb call is constructed by use of a macro, *pv_set*, which is defined in *include/minix/devio.h*. The act of writing the operation code to the command register makes the operation begin. When it is complete, an interrupt is generated and a notification message is sent. If the command times out the alarm will expire and a synchronous alarm notification will wake up the disk driver.

The next several functions are short. *W_need_reset* (line 12999) is called when timeouts occur while waiting for the disk to interrupt or become ready. The action of *w_need_reset* is just to mark the *state* variable for every drive in the *wini* array to force initialization on the next access.

W_do_close (line 13016) has very little to do for a conventional hard disk. Additional code is needed to support CD-ROMs.

Com_simple is called to issue controller commands that terminate immediately without a data transfer phase. Commands that fall into this category include those that retrieve the disk identification, setting of some parameters, and recalibration. We saw an example of its use in *w_identify*. Before it is called the *command* structure must be correctly initialized. Note that immediately after the call to *com_out* a call to *at_intr_wait* is made. This eventually does a receive which blocks until a notification arrives signifying that an interrupt has occurred.

We noted that *com_out* does a sys_setalarm kernel call before asking the system task to write the registers which set up and execute a command. As we mentioned in the overview section, the next receive operation normally should receive a notification indicating an interrupt. If an alarm has been set and no interrupt occurs, the next message will be a *SYN_ALARM*. In this case *w_timeout* line 13046 is called. What needs to be done depends on the current command in *w_command*. The timeout might have been left over from a previous operation, and *w_command* may have the value *CMD_IDLE*, meaning the disk completed its operation. In that case there is nothing to do. If the command does not complete and the operation is a read or write, it may help to reduce the size of I/O requests. This is done in two steps, first reducing the maximum number of sectors that can be requested to 8, and then to 1. For all timeouts a message is printed and *w_need_reset* is called to force re-initialization of all drives on the next attempted access.

When a reset is required, *w_reset* (line 13076) is called. This function makes use of a library function, *tickdelay*, that sets a watchdog timer and then waits for it to expire. After an initial delay to give the drive time to recover from previous operations, a bit in the disk controller's control register is **strobed**—that is, set to a logical 1 level for a definite period, then returned to the logical 0 level. Following this operation, *w_waitfor* is called to give the drive a reasonable period to signal it is ready. In case the reset does not succeed, a message is printed and an error status returned.

Commands to the disk that involve data transfer normally terminate by generating an interrupt, which sends a message back to the disk driver. In fact, an interrupt is generated for each sector read or written. The function *w_intr_wait*

(line 13123) calls *receive* in a loop, and if a *SYN_ALARM* message is received *w_timeout* is called. The only other message type this function should see is *HARD_INT*. When this is received the status register is read and *ack_args* is called to reinitialize the interrupt.

W_intr_wait is not called directly; when an interrupt is expected the function called is the next one, *at_intr_wait* (line 13152). After an interrupt is received by *at_intr_wait* a quick check is made of the drive status bits. All is OK if the bits corresponding to busy, write fault, and error are all clear. Otherwise a closer look is taken. If the register could not be read at all, it is panic time. If the problem was a bad sector a specific error is returned, any other problem results in a general error code. In all cases the *STATUS_ADMBSY* bit is set, to be reset later by the caller.

We have seen several places where *w_waitfor* (line 13177) is called to do busy waiting on a bit in the disk controller status register. This is used in situations where it is expected the bit might be clear on the first test, and a quick test is desirable. For the sake of speed, a macro that read the I/O port directly was used in earlier versions of MINIX—this is, of course, not allowable for a user-space driver in MINIX 3. The solution here is to use a do ... while loop with a minimum of overhead before the first test is made. If the bit being tested is clear there is an immediate return from within the loop. To deal with the possibility of failure a timeout is implemented within the loop by keeping track of clock ticks. If a timeout does occur *w_need_reset* is called.

The *timeout* parameter that is used by the *w_waitfor* function is defined by *DEF_TIMEOUT_TICKS* on line 12228 as 300 ticks, or 5 seconds. A similar parameter, *WAKEUP* (line 12216), used to schedule wakeups from the clock task, is set to 31 seconds. These are very long periods of time to spend busy waiting, when you consider that an ordinary process only gets 100 msec to run before it will be evicted. But, these numbers are based upon the published standard for interfacing disk devices to AT-class computers, which states that up to 31 seconds must be allowed for a disk to "spin up" to speed. The fact is, of course, that this is a worst-case specification, and that on most systems spin up will only occur at power-on time, or possibly after long periods of inactivity, at least for hard disks. For CD-ROMs or other devices which must spin up frequently this may be a more important issue.

There are a few more functions in *at_wini.c*. *W_geometry* returns the logical maximum cylinder, head, and sector values of the selected hard disk device. In this case the numbers are real ones, not made up as they were for the RAM disk driver. *W_other* is a catch-all for unrecognized commands and ioctls. In fact, it is not used in the current release of MINIX 3, and we should probably have removed it from the Appendix B listing. *W_hw_int* is called when a hardware interrupt is received when it is not expected. In the overview we mentioned that this can happen when a timeout expires before an expected interrupt occurs. This will satisfy a receive operation that was blocked waiting for the interrupt, but the

interrupt notification may then be found by a subsequent receive. The only thing to be done is to reenable the interrupt, which is done by calling the next function, *ack_irqs* (line 13297). It cycles through all the known drives and uses the sys_irqenable kernel call to ensure all interrupts are enabled. Finally, at the end of *at_wini.c* two strange little functions are found, *strstatus* and *strerr*. These use macros defined just ahead of them on lines 13313 and 13314 to concatenate error codes into strings. These functions are not used in MINIX 3 as described here.

3.7.6 Floppy Disk Handling

The floppy disk driver is longer and more complicated than the hard disk driver. This may seem paradoxical, since floppy disk mechanisms are simpler than those of hard disks, but the simpler mechanism has a more primitive controller that requires more attention from the operating system. Also, the fact that the medium is removable adds complications. In this section we will describe some of the things an implementer must consider in dealing with floppy disks. However, we will not go into the details of the MINIX 3 floppy disk driver code. In fact, we have not listed the floppy disk driver in Appendix B. The most important parts are similar to those for the hard disk.

One of the things we do not have to worry about with the floppy driver is the multiple types of controller to support that we had to deal with in the case of the hard disk driver. Although the high-density floppy disks currently used were not supported in the design of the original IBM PC, the floppy disk controllers of all computers in the IBM PC family are supported by a single software driver. The contrast with the hard disk situation is probably due to lack of motivation to increase floppy disk performance. Floppy disks are rarely used as working storage during operation of a computer system; their speed and data capacity are too limited compared to those of hard disks. Floppy disks at one time were important for distribution of new software and for backup, but as networks and larger-capacity removable storage devices have become common, PCs rarely come standard with a floppy disk drives any more.

The floppy disk driver does not use the SSF or the elevator algorithm. It is strictly sequential, accepting a request and carrying it out before even accepting another request. In the original design of MINIX it was felt that, since MINIX was intended for use on personal computers, most of the time there would be only one process active. Thus the chance of a disk request arriving while another was being carried out was small. There would be little to gain from the considerable increase in software complexity that would be required for queueing requests. Complexity is even less worthwhile now, since floppy disks are rarely used for anything but transferring data into or out of a system with a hard disk.

That said, the floppy driver, like any other block driver, can handle a request for scattered I/O. However, in the case of the floppy driver the array of requests