CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

APPLIED PCs

Fred Eady

Construct an ATA Hard Drive Controller

It's about time you had full control of your hard drive. The controller you've been waiting for is just one project away. This month, Fred shows you how easy it is to build an ATA hard drive controller. Amazingly, all you need is a good micro and a few everyday parts.

Do you remember your first computer with a hard drive? What about when 5 MB was a lot of hard drive space? Have you ever wanted to put something of your own on a hard drive without having to rely on somebody else's expensive and proprietary hardware and driver code? Ever want to read, write, and control a hard drive with a microcontroller?

If you answered "yes" to any one of these questions, then this project is for you. I'll show you how to build an ATA hard drive controller with a microcontroller and a few common parts. In addition, you'll learn how to write simple code that will form the basis for deploying a stand-alone, networkable, microcontroller-based data storage system.

Even if you aren't interested in communicating with hard drives, there's something here for those of you who don't need a microcontroller-driven hard drive controller. The bonus track is in the hardware. Do you need an in-system programming-capable test stand for an Atmel ATmega128 with RS-232, Ethernet, and 64-KB of external 16-bit memory? Well, this project is for you too. Hard drives or no hard drives, let's get started.

THE HARDWARE

Hanging a standard PC or laptop hard drive from the 40-pin connector shown in Photo 1 is the reason why we're gathered here today. I wanted the ATA hard drive controller's electronics to be flexible. So, in addition to the standard RS-232 port, which is driven by a Sipex SP233ECT, I added 10-Mbps Ethernet capabilities with the RTL8019AS/LF1S022 combination.

The ATmega128 has plenty of internal SRAM (4 KB), but I thought adding 64 KB of 16-bit external SRAM would be nice. Adding the SRAM is sort of like buying rope: you can always make the rope shorter, but it's a pain to add rope later if you need it.

The ATmega128 has enough I/O structure to service the big SRAM with some help from a couple of 74HCT573 latches. As you can see in Figure 1, the external SRAM is attached to the ATmega128 in the standard manner. This allows those of you who aren't interested in placing bits on a spinning piece of magnetically coated aluminum to do your thing with the big chunk of SRAM and the raw power of the ATmega128. With the SRAM in this configuration, the results of the hard drive I/O operations can be buffered by the external SRAM or operated on by the AVR directly.

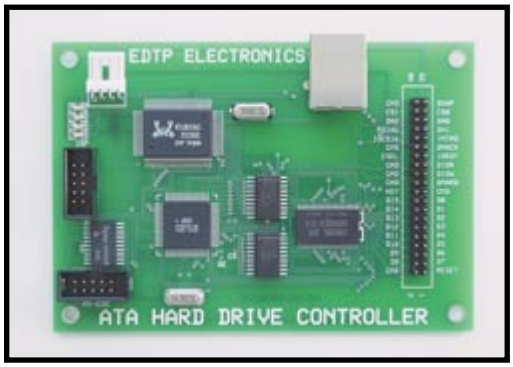


Photo 1—*The board is clean and simple. All of the supporting capacitors and resistors are SMT parts mounted on the opposite side of the board.*

Listing 1—*A core set of registers is the gateway to the hard drive platters. Everything needed to access a particular data point on the hard drive is represented here.*

```
//ATA I/O port functions and address definitions
//Control block registers
//                                        RESET
//                                        |DIOW
//                                        ||DIOR
//                                        |||DA0
//                                        ||||DA1
//                                        |||||DA2
//                                        ||||||CS0
//                                        |||||||CS1
//                                        ||||||||
#define ATA_IO_HIZ                      0b11111111
#define ATA_IO_ASTAT                    0b11101110
#define ATA_IO_DEVICECNTL               0b11101110
//Command block register addresses
#define ATA_IO_DATA                     0b11100001
#define ATA_IO_ERROR                    0b11110001
#define ATA_IO_FEATURES                 0b11110001
#define ATA_IO_SECTORCNT                0b11101001
#define ATA_IO_SECTORNUM                0b11111001
#define ATA_IO_CYL_L                    0b11100101
#define ATA_IO_CYL_H                    0b11110101
#define ATA_IO_DEVICE_HEAD              0b11101101
#define ATA_IO_STATUS                   0b11111101
#define ATA_IO_CMD                      0b11111101
```

The Ethernet interface, shown in Figure 2, is actually a wire-by-wire copy of the Packet Whacker microcontroller NIC. The original Packet Whacker firmware that was written for the ATmega series of AVRs is used in the ATA hard drive controller code, as well. The Packet Whacker code already has hooks to allow for the easy transmission and reception of the hard drive data.

The RS-232 port has a dual purpose. Running at 57.6 kbps, it's fast enough to spit out a sector's worth of ASCII data to a terminal emulator for debugging. In addition, it can be used efficiently in an application to transfer data and commands between the AVR-based hard drive controller and a peripheral device. The 10-pin header makes assembling a serial cable easy, if you use 9- or 25-pin IDC shell connector parts and ribbon cable.

I've standardized with 10-pin male headers for all of the external ports with the obvious exceptions of the Ethernet and hard drive I/O ports. As long as you put the right connector in the correct header socket, using the 10-pin headers with the keyed shrouds eliminates the possibility of inserting the ISP and serial connectors incorrectly.

The 10-pin arrangement is also standard for most of the ISP dongles that support the AVR series of ISP-

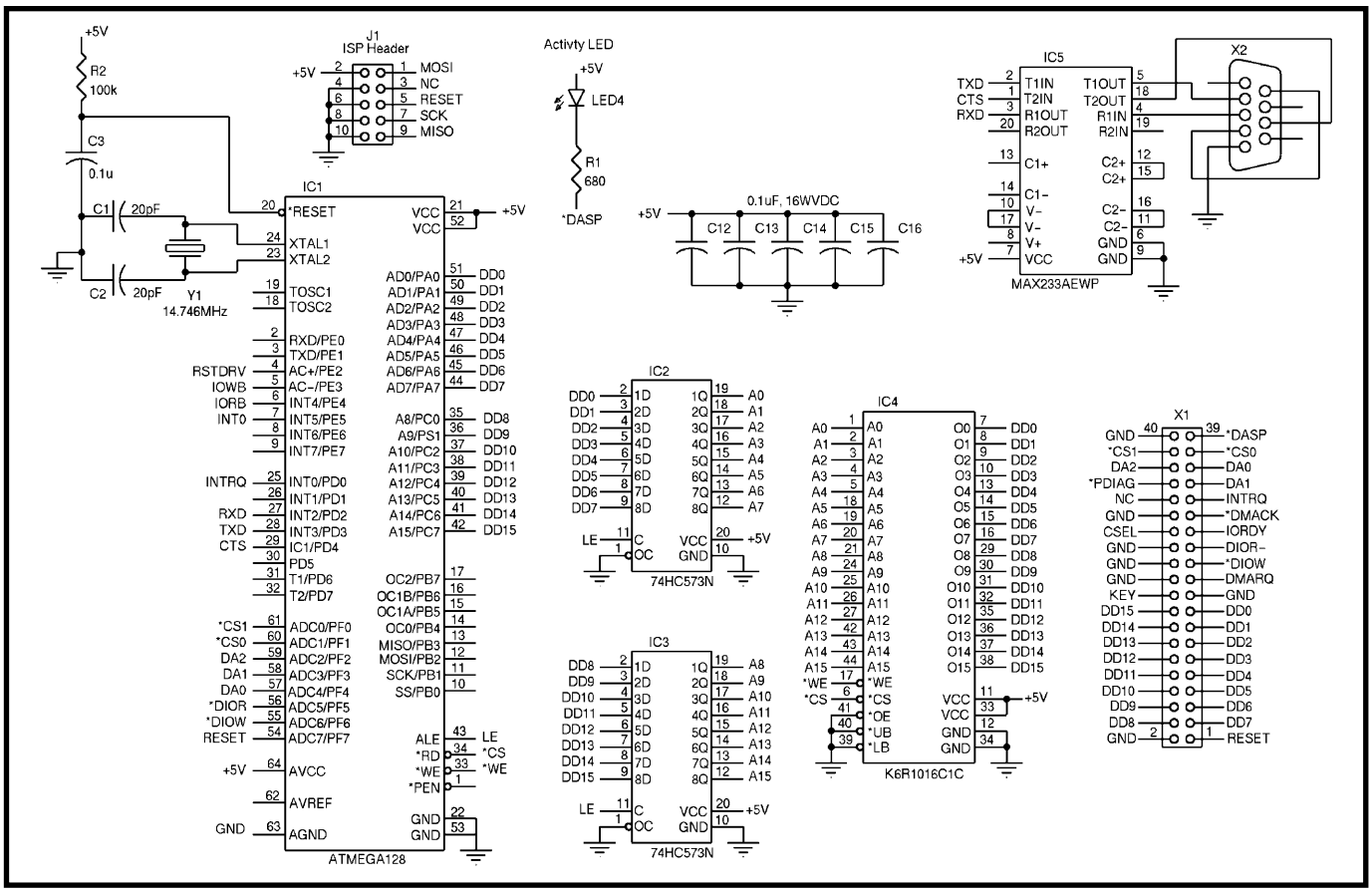


Figure 1—*No surprises here, either. The craftiness of this design lies in the way the firmware utilizes the hardware resources.*

capable microcontrollers. I used a Kanda AVR ISP dongle and a version of the company's ISP software to program the hard drive controller's ATmega128. Because the dongle interface is dedicated to certain pins on the AVR and power isn't transferred within the programming cable, I was able to keep the dongle attached to the hard drive controller throughout the programming and debugging process.

The ATmega128 is clocked at 14.746 MHz to keep the data rate error percentage at a minimum for the 57.6-kbps serial port. For this project, the ATmega128 I used was a 5-V part that can run at 16 MHz. The 14.746 MHz is the closest standard crystal to the maximum clock speed that will clock the big AVR with the least amount of serial data bit error rate.

The first spin of the ATA hard drive controller used a 2-mm header that mated directly to the 44 I/O pins found on 2.5″ laptop drives. My experiences with the 2-mm parts were not good ones. The pins and connectors are fragile, and I really don't like working with the 1-mm ribbon cable.

In the process of attempting to work at 2 mm, I purchased a gaggle of new surplus 2.5″ Hitachi 540-MB drives. That purchase, as it turns out, was a good thing. After junking the 2-mm idea, I purchased some surplus 850-MB, 3.5″ drives that turned out to be mostly junk. I never really had any inclination to put real data on them, so it's not a total loss. At less than $10 per drive, what did I expect?

When I bought the laptop drives, I also purchased some 2-mm to 0.1″ (or 2.5″ to 3.5″) converter boards. The idea was to be able to attach the laptop drives to a PC for the debugging and verification of the ATA drive controller's firmware and hardware.

The moral of the 2-mm hard drive story is that, thanks to my foresight, you'll see how I brought the ATA hard drive controller to life with the 2.5″ Hitachi drives and converter boards. This may sound funny, but when I was formatting the 3.5″ 850-MB drives, I was hoping that a few of them would show some errors. I wanted to verify that the hard drive controller could detect them, and then show you what they looked like. I really didn't expect them to be trashed so badly. So, the drive error examples will come from the 3.5″ drives, and the good data examples will feature the smaller Hitachi drives.

The ATA hard drive controller requires a single 5-VDC power source. Also, the Hitachi drives require only

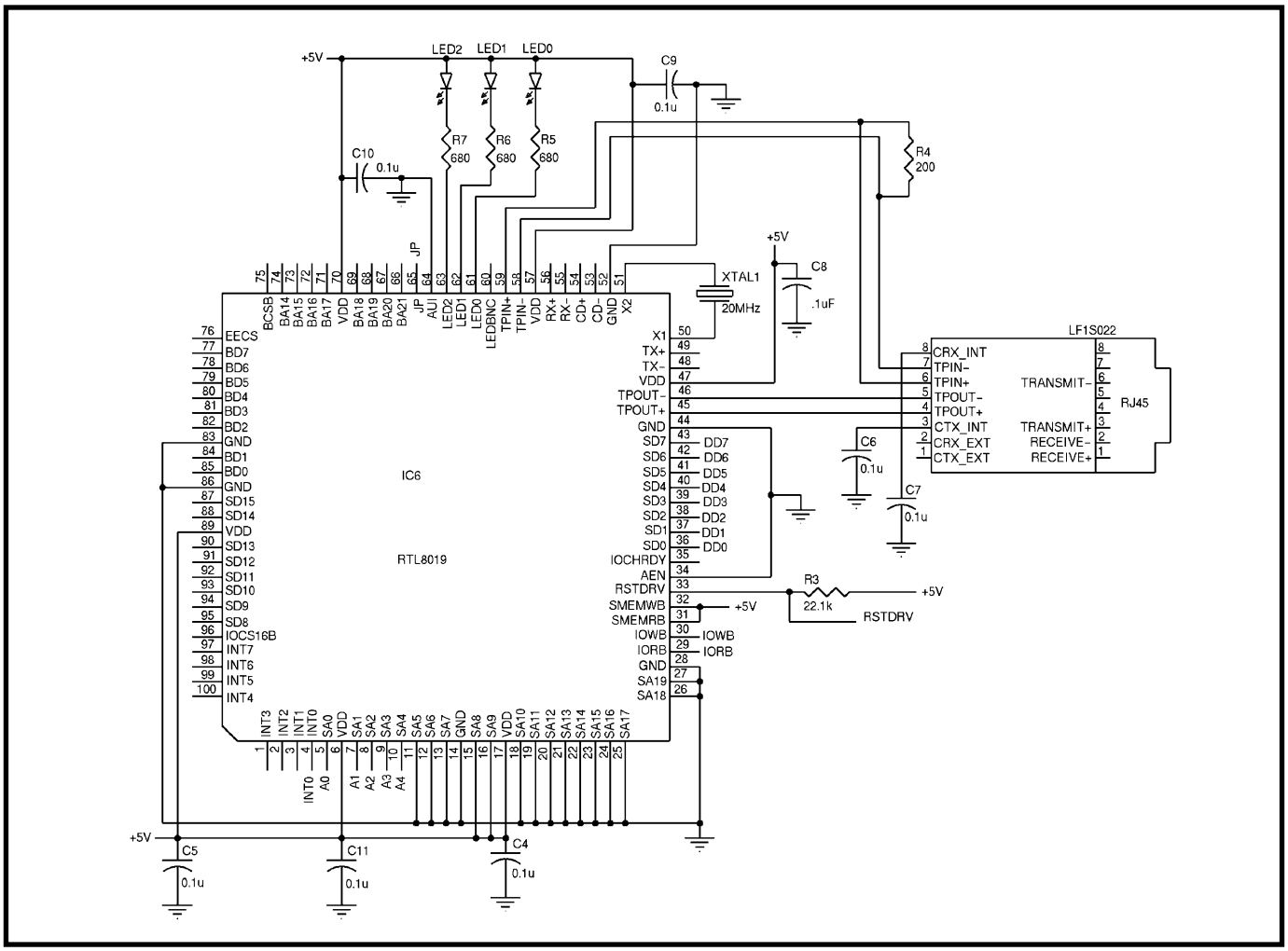


Figure 2—*If this looks familiar, it's because it's actually a Packet Whacker that's been melded into the ATA controller design. The Packet Whacker code was reused, as well; it can be found wound into the ATA controller source.*

Listing 2—*Because the routines are identical, with exception to the status bit that's checked, I took some liberties and squashed all three* `ready`, `busy`, *and* `error` *routines into a single function to save some space. The unsigned* `int ata_bsy(void)` *function includes the* `if(ata_byte_read & ATA_STAT_BSY)` *line and the other two functions follow the same logic.*

```
#define recalibrate          ata_send_cmd(CMD_RECALIBRATE)
#define CMD_RECALIBRATE      0x10
#define PORT_ATA_IO_CNTL     PORTF
#define ATA_DIOR             0x20
#define PORT_ATA_DATA_L_IN   PINA
#define ATA_STAT_BSY         0x80  //ATA busy
#define ATA_STAT_RDY         0x40  //ATA ready
#define ATA_STAT_ERR         0x01  //ATA error
#define busy                 ata_bsy()
#define drq                  ata_drq()
#define error                ata_err()
#define ready                ata_rdy()
#define hard_reset           ata_hard_reset()
#define select_device_0      ata_select_device(0x00)
#define select_device_1      ata_select_device(0x01)
#define recalibrate          ata_send_cmd(CMD_RECALIBRATE)
#define identify_device      ata_send_cmd(CMD_IDENTIFY_DEVICE)
*****************************************************************
Initialize drive. This routine assumes drive 0 is the only drive
that is attached.
*****************************************************************
void init_ata(void)
{
 while(!ready & busy);
 hard_reset;
 delay_ms(10);
 while(!ready & busy);
 select_device_0;
 while(!ready & busy);
 recalibrate;
 while(busy);
 if(error)
   printf("ERROR!");
 printf("\r\nDrive is READY!\r\n");
//Functions are squashed for space savings
unsigned int ata_bsy(void)
unsigned int ata_rdy(void)
unsigned int ata_err(void)
{
   unsigned char ata_byte_read;
   avr_databus_in;
   delay_us(1);
   PORT_ATA_IO_CNTL = ATA_IO_STATUS;
   PORT_ATA_IO_CNTL &= ~ATA_DIOR;
   delay_us(1);
   ata_byte_read = PORT_ATA_DATA_L_IN;
   PORT_ATA_IO_CNTL |= ATA_DIOR;
   PORT_ATA_IO_CNTL = ATA_IO_HIZ;
   if(ata_byte_read & ATA_STAT_BSY)
   if(ata_byte_read & ATA_STAT_RDY)
   if(ata_byte_read & ATA_STAT_ERR)
     return 1;
   else
    return 0;
}                                  //End of squashed functions

void ata_hard_reset(void)
{
   avr_databus_in;
   PORT_ATA_IO_CNTL = ATA_IO_HIZ;
   PORT_ATA_IO_CNTL &= ~ATA_RESET;
```

(Continued)

5 VDC. However, the larger 3.5″ drives need 12 VDC in addition to the 5 VDC. The original spin of the hard drive controller used a 2-mm, 44-pin hard drive I/O attachment point. The extra four pins on the 2-mm connector provided 5 VDC and ground for the 2.5″ drives right at the hard drive I/O connector. In this spin, the 44-pin, 2-mm pin set is replaced with the standard 40-pin 0.1″ pin set, and there isn't a power supply outlet at the hard drive I/O connector.

The hard drive controller is equipped with a standard 4-pin floppy drive power plug. As you might have figured out, the inclusion of a standard PC power connector on the hard drive controller allows you to power the 3.5″ drive and the hard drive controller's electronics from a common off-the-shelf PC power supply.

If the 2.5″ drives are used, you'll have to provide an attachment to supply power to the extra I/O-based power pins on the drive. That's where the 2.5″ to 3.5″ drive I/O adapters come in. The adapters I purchased have a 3.5″ drive power connector that has only the 5-VDC lines tapped into the 44-pin, 2-mm drive connector. The drive converter board allows you to use the smaller 2.5″ drives with the standard 40-pin 0.1″ cables and a PC power supply. Although using a commodity power supply is the easiest way to go, any other suitable power supply method will work just as well. Photo 2 is a shot of an Hitachi 2.5″ drive and its associated converter board attached to the ATA hard drive controller.

THE FIRMWARE

I wanted the ATA hard drive controller to be capable of interfacing to any standard ATA device. With that design requirement mind, I wrote the hard drive controller's AVR firmware with ImageCraft's ICCAVR C compiler and guidance from the ATA-3 specification. What I ended up with was a basic set of routines that allows you to exercise the standard ATA command set, query the hard drive register set, and exchange data with the attached ATA hard drive.

As soon as I had access to the hard drive's register set and data, I set out to write code to move the data that

was harvested from the hard drive to the outside world. The first logical choice of data transport was a serial port. After thinking it over, I decided that an Ethernet interface would be an excellent way to move data in and out of the hard drive controller. The Ethernet port would allow the hard drive controller to be networked and provide a high-speed data connection for transfer rates beyond the capabilities of a serial port. In either case (serial or Ethernet), you could use any of the Visual (e.g., Visual Basic, Visual C) or Borland compilers to build an embedded or PC interface to the ATA hard drive controller.

The first order of business as I started to develop the AVR firmware was to define all of the functions that would run against the hard drive. With respect to the software, a hard drive looks like an 8- or 16-bit I/O port that leads to an internal register set. Register I/O is normally achieved in 8-bit mode, while data transfers are typically performed in 16-bit operations.

If you review Figure 1, you'll see that a 16-bit data bus is pinned out on the 40-pin hard drive I/O connector. The data bus signals are supported by a set of I/O read and write signals. Access to the internal hard drive register set is accomplished using the I/O read/write signals and data bus signals in conjunction with the address and select signals found on the 40-pin hard drive I/O connector. The control block is the core set of hard drive internal registers. Listing 1 is my definition of how the control block registers are addressed.

If you take a close look at the control block register definitions in Listing 1, you'll notice that the basic components (in register form) of addressing data on a hard drive are represented. Cylinder head sector (CHS) addressing is implied in the control block register names; however, in the ATA hard drive controller firmware, I'll use these

Listing 2—*Continued*

```
    delay_ms(10);
    PORT_ATA_IO_CNTL |= ATA_RESET;
}

void ata_select_device(unsigned char device)
{
  PORT_ATA_IO_CNTL = ATA_IO_DEVICE_HEAD;
  switch (device)
  {
              case 0x00:
                      ata_write_byte(ATA_DH_DEV0);
                      break;
              case 0x01:
                      ata_write_byte(ATA_DH_DEV1);
                      break;
              default:
                      ata_write_byte(ATA_DH_DEV0);
                      break;
  }
}

void ata_send_cmd(unsigned char atacmd)
{
  PORT_ATA_IO_CNTL = ATA_IO_CMD;
  avr_databus_out;
  PORT_ATA_DATA_L_OUT = atacmd;
  ata_write_pulse;
  PORT_ATA_IO_CNTL = ATA_IO_HIZ;
  avr_databus_in;
}
```

Word	F/V	Identify device information	Word	F/V	Identify device information
0		General configuration of bit-significant information:	1	F	Number of logical cylinders
	F	15 0 = ATA device 1 = ATAPI device	2	R	Reserved
	F	14 Obsolete	3	F	Number of logical heads
	F	13 Obsolete	4	X	Obsolete
	F	12 Obsolete	5	X	Obsolete
	F	11 Obsolete	6	F	Number of logical sectors per logical track
	F	10 Obsolete	7–9	X	Vendor specific
	F	9 Obsolete	10–19	F	Serial number (20 ASCII characters)
	F	8 Obsolete	20	X	Obsolete
	F	7 1 = Removable media device	21	X	Obsolete
	F	6 1 = Not removable controller and/or device	22	F	Number of vendor-specific bytes available on read/write long commands
	F	5 Obsolete	23–26	F	Firmware revision (eight ASCII characters)
	F	4 Obsolete	27–46	F	Model number (40 ASCII characters)
	F	3 Obsolete	47	X	15–8 Vendor specific
	F	2 Obsolete		R	7–0 00h = Reserved
	F	1 Obsolete		F	01h–FFh = Maximum number of sectors that can be transferred per interrupt on read/write multiple commands
	F	0 Reserved	48	R	Reserved

Table 1—*If you like to write code that parses data, then writing ATA hard drive code will keep you happy (and busy) for days. The data in Photo 3 was culled from this table of words spoken by the little Hitachi DK211A-54.* F *represents a fixed value,* V *represents a variable value,* X *represents a vendor-specific value, and* R *represents a reserved value.*

same CHS-based registers to perform logical block address (LBA) mode addressing operations. LBA mode is a means set forth by the ATA standards to allow for the linear addressing of sectors. LBA addressing is derived from the CHS addressing format as follows:

LBA = ((cylinder × heads_per_cylinder + heads) × sectors_per_track) + sector – 1

Photo 2—*The 540-MB drive formats quickly and is easy to handle even with the converter boards attached. This made for quick turnarounds in the initial development stages when I was experimenting, debugging, and doing a lot of hard drive formatting.*

For instance, cylinder 0, head 0, sector 1 is LBA address 0. Therefore, for LBA mode to function, the hard drive must support LBA mode internally, and that's the case for the 540-MB laptop drives as well as the larger 850-MB 3.5" drives.

The ultimate goal is to use LBA mode to read and write to sectors on the hard drive. To do this, you must first be able to read and write to the hard drive's register set. A good place to start with the firmware description of this process is with the hard drive initialization routine, whose source code is included in Listing 2.

The first register access occurs when the `while(!ready & busy)` statement executes. Note that `ready` and `busy` are macros that call the `ata_rdy(void)` and `ata_busy(void)` functions. The `ata_rdy(void)` and `ata_busy(void)` functions are identical with the exception of the status bit they check. In both cases the AVR data bus pins are put in Input mode, the status register is addressed, the I/O read pin is toggled, and the status register data is read (8 bits). Additionally, the hard drive I/O port is put into a high-impedance state, the status condition is determined, and a return code is generated. Note that the external buffer SRAM is not used by these functions.

After the hard drive has done its own power-on reset, the ready bit will show that the hard drive is ready for a command, and the busy bit will indicate a "not busy" status. At this point, a hard reset is toggled using the RESET pin on the hard drive I/O bus, and time is marked to allow the physical and electrical hard drive reset process to finish. Because I was attaching a single hard drive that's strapped as master drive 0, I selected drive 0 in LBA mode using the

`select_drive_0` macro. The next step was to issue a recalibrate command and check the error status register. In instances like this, a "drive ready" banner is sent to the serial port if all is well.

At that point, I wasn't ready to start reading hard drive sectors, because I needed to make sure I could address and command the hard drive interface accurately. The easiest way to verify this was to execute an ATA Identify Device command.

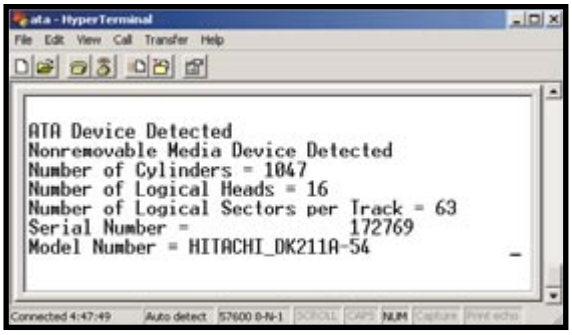


Photo 3—*Things are good when the numbers in this photo match the numbers written on the hard drive.*

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

Basically, the Identify Device command instructs the hard drive to divulge its factory-loaded identifiers, and 255 words are returned. All I had to do was pick up the words from the hard drive I/O port, parse them, and send the results to the serial port. All 255 words weren't needed. As you can see in Table 1, the first 46 words tell you if things are working correctly. Photo 3 is a HyperTerminal shot showing you what the little Hitachi drive had to say about itself.

Now that you know how to get data from the hard drive, I'll show you how to read a sector. Before the code is tested, however, there's work to be done on the hard drive, and you'll need a way to verify your results.

Because I plan to develop AVR firmware to manipulate FAT32 formatted drives, it would be logical to format the hard drives that will be used with MSDOS by way of Windows 98. Formatting in this way puts master boot records, partition tables, and data in predictable places on the drive. Two drives should be formatted: one is used on the ATA hard drive controller, and the other is used on a PC for verification and as an aid in debugging.

The verification program for the PC is called WinHex. Normally, WinHex is used to inspect and repair files on PC hard drives. This program does it all as far as hard drives are concerned; it understands FAT12, FAT16, FAT32, NTFS, and CDFS. In addition, WinHex includes a disk editor that allows you to become a dangerous hard drive technician. You can also use WinHex to create templates that automatically parse known data areas of the hard drive.

Photo 4 is a screen shot of an actual

WinHex panel that's aimed at the 2.5" Hitachi drive attached to the PC. I've dialed in the MBR master boot record (MBR), which resides at cylinder 0, head 0, and sector 1 or LBA 0. If all goes well with the sector read on the hard drive controller, the data in the HyperTerminal window should be identical to the bytes found in the WinHex window.

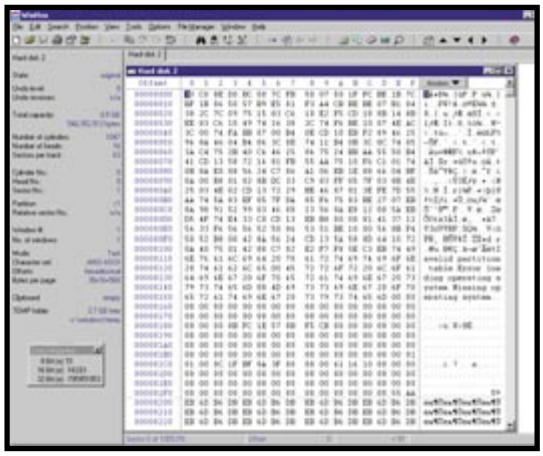
The HyperTerminal readout in Photo 5 matches the numbers picked up by WinHex from the clone drive in Photo 4. As Spock would say, "Random chance seems to have operated in our favor." The `ata_read_sector()` function shown in Listing 3 works as designed. Reading a sector in LBA mode entails loading the Device/Head register with the LBA mode bit set, loading the cylinder High/Low and Sector registers, and issuing the Read Sectors command.

Here's where that big chunk of external 16-bit SRAM is handy. Instead of pulling the data directly into the AVR, I used the AVR to generate address information for the SRAM and manipulate the SRAM's write enable and chip select lines to store the incoming data in the external 64 KB of SRAM.

I divided the SRAM into logical pages of 256 words each and wrote routines to read and write these pages. Each page of external SRAM holds one sector of hard drive data, allowing up to 256 sectors to be buffered. That pretty much takes care of verifying the ATA hard drive controller's read functionality.

Writing to the hard drive is a similar process. The external SRAM is filled with a sector's worth of data (256 words), and then that particular SRAM page is written to the hard drive I/O port's data bus. Instead of performing an ATA I/O read, an ATA I/O write is performed when the data is presented on the external SRAM data pins. I tested the write sector code successfully on random sectors of the hard drive attached to the ATA hard drive controller. Additionally, I verified the writes by moving the hard drive to the PC and reading the sectors I wrote using WinHex.

Photo 4—*There isn't much about a hard drive you will want to know that WinHex won't tell you.*



GETTING FAT

All of the reading and writing up to this point was completed with simple C routines that were teamed together to perform a much larger and more complex task. Believe it or not,

designing the ATA hard drive controller hardware and finishing the C coding for the controller I/O functions was the easy part. The Ethernet code was just as easy, because I copied AVR code that was already written for the Packet Whacker microcontroller NIC. The next step in the process of assembling a microcontroller-based networkable mass storage system was a bit more demanding.

I completed a tremendous amount of research in preparation for writing AVR code to implement Microsoft's FAT32 file system. Thanks to a series of *Circuit Cellar* articles written by our own Jeff Bachiochi (*Circuit Cellar* 143–146), I had a good idea about the roads I will travel and battles I will fight.

The journey starts with the bytes in Photo 4, the master boot record. Because I'm not executing instructions on a legacy x86 machine and using a PC BIOS or MSDOS, I'll have to interpret the data and adapt it to the AVR. For instance, there's executable code in the MBR that I don't care about. The problem is that I have to navigate through it to find markers that either give me information about where FAT-related constants and parameters reside or point me to places where I can read and write my data.

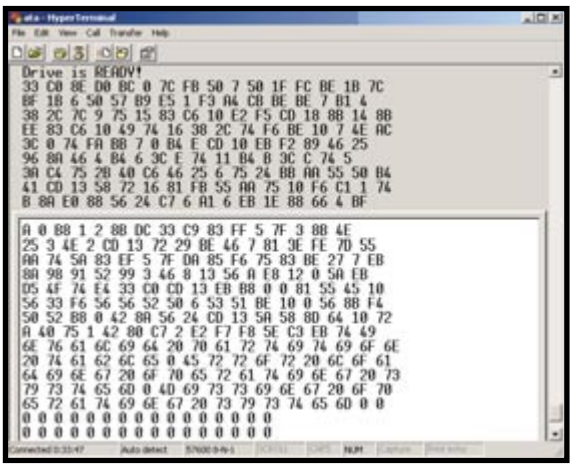


Photo 5—*The format of the data may not be pretty, but the data itself is beautiful because it matches the clone drive's MBR data read by WinHex.*

In the case of the MBR, I'm interested only in the last 66 bytes, because that's where the partition table resides. The fun starts at offset 0x1BE in the MBR, which is the first partition entry. Because the drives I formatted were secondary drives, the FDISK program couldn't make their partitions active. So, the first byte at offset 0x1BE was 0x00 (inactive) on my drives.

Other interesting information resides at offset 0x1C6 in the MBR. This is the number of sectors between the MBR and the first sector of the first partition. As you can see in Photo 4, WinHex shows that number as 0x3F. I used the WinHex program to dial in 0x3F sectors beyond the MBR and, lo and behold, there was the FAT32 boot sector with additional fields of information to parse. The plan is to collect documentation and use WinHex to obtain the actual visuals concerning how data and control areas on a FAT32 hard disk are defined and laid

out.

So, you see that I have my work cut out for me. The good news is that you'll be able to share the fruits of my labor, because I will make the ATA hard drive controller hardware

Listing 3—*There aren't any tricks in this code. It's all simple read and write I/O between the hard disk and the SRAM. The routine reads an LBA-addressed sector into an SRAM page.*

```
*****************************************************************
Read a sector. device = 0x00 or 0x01
*****************************************************************
unsigned char ata_read_sector(unsigned char device, unsigned
long lbasector \ ,unsigned int page)
{
 unsigned int i,ram_address;
 lbasector &= 0x0FFFFFFF;
 ata_set_io_addr(ATA_IO_DEVICE_HEAD);
 switch (device)
 {
  case 0x00:
        ata_write_byte(lbasector >> 24 | 0xE0);
        break;
  case 0x01:
        ata_write_byte(lbasector >> 24 | 0xF0);
        break;
  default:
        ata_write_byte(lbasector >> 24 | 0xE0);
        break;
 }
 while(busy);
 ata_set_io_addr(ATA_IO_CYL_H);
 ata_write_byte(lbasector >> 16);
 while(busy);
 ata_set_io_addr(ATA_IO_CYL_L);
 ata_write_byte(lbasector >> 8);
 while(busy);
 ata_set_io_addr(ATA_IO_SECTORNUM);
 ata_write_byte(lbasector);
 while(busy);
 ata_set_io_addr(ATA_IO_SECTORCNT);
 ata_write_byte(0x01);
 while(busy);
 ata_send_cmd(CMD_READ_SECTORS);
 while(busy);
 while(!drq);

 ram_address = page * 0x100;
 for(i=0;i<256;++i)
 {
  avr_databus_out;
  PORT_ATA_DATA_H_OUT = ram_address >> 8;
  PORT_ATA_DATA_L_OUT = ram_address;
  latch_ram_addr;
  avr_databus_in;
  ram_on;
  while(busy);
  PORT_ATA_IO_CNTL = ATA_IO_DATA;
  PORT_ATA_IO_CNTL &= ~ATA_DIOR;
  delay_us(1);
  ram_write_pulse;
  delay_us(1);
  PORT_ATA_IO_CNTL |= ATA_DIOR;
  PORT_ATA_IO_CNTL = ATA_IO_HIZ;
  while(busy);
  ram_off;
  ++ram_address;
 }
  return (error);
}
```

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2003/150/.

SOURCES

ATmega128 Microcontroller
Atmel Corp.
(408) 441-0311
www.atmel.com

DK211A-54 2.5" Disk drives
Hitachi, Ltd.
(800) 448-2244
www.hitachi.com

ICCAVR C compiler
ImageCraft Creations, Inc.
(650) 493-9326
www.imagecraft.com

AVR ISP Dongle and ISP software
Kanda Systems
+44 (0) 870 744 6807
www.kanda.com

MAX233
Maxim Integrated Products, Inc.
(408) 737-7600
www.maxim-ic.com

74HCT573 Octal D-type transparent latch
Philips Semiconductor
www.semiconductors.philips.com

RTL8019AS Ethernet controller
Realtek Semiconductor Corp.
+886 (0) 3 578 0211
www.realtek.com.tw

SP233ECT RS-232 Line drivers/receivers
Sipex Corp.
(978) 667-8700
www.sipex.com

WinHex
X-Ways Software Technology AG
www.x-ways.com