This is a archived copy of a text written by Peter Faasse, and later turned into nice html on a web page created by Wesley Moore, October, 1999. Wesley's page has been hosted at RMIT University. This archive was made on March 31, 2000, to prepare for the event where RMIT removes Wesley's pages, perhaps when he leaves the university. Wesley has done a very nice job of turning Peter's original text file into a nice HTML document. It would be a shame if the only copy were to be suddently removed from the net.

Wesley's PIC Pages

IDE Controller

A description of an IDE interface for a microcontroller

How to connect an IDE disk to a microcontroller using an 8255

By Peter Faasse
faasse AT nlr DOT nl

Contents

- 63B03 assembly listing

Introduction

Some time ago I have dropped that I had connected an IDE harddisk to one of my microcontrollers. This has provoked a response that I had not foreseen. Since that day I have received some one to two e-mails a day requesting more details about what I had done. At first I have mailed a more or less cryptic description of my interface to some of the requestors. That only resulted in more e-mail asking for more details. As it seems some people out there are really interested in how my contraption is made. In this description I will attempt to satisfy the information-hunger of all you out there who's appetite I seem to have awakened.

This interface first came to my mind when I re-read some old, old computer magazines. In one of them, the German magazine called C't there was a short description of how and IDE interface is put together. This is in the November issue of 1990. The article describes an IDE interface for both the PC-XT(!) and the PC-AT. The circuit diagrams of the article indicate that the hardware of an IDE interface is in fact very simple. It is essentially a data bus extension from the PC-AT bus to an IDE device. For a PC the hardware comes down to some bus buffers and some decoding. When a disk is connected as an IDE device the PC-AT still 'sees' the old-type control registers of the ancient MFM disk controller. In the article the entire interface is implemented using simple TTL chips. The main problem in a PC seems to be how to keep the harddisk interface and the floppy interface from colliding on some register addresses. If the IDE interface is implemented based on some controller system this is of course no problem.

>From a controller point of view an IDE interface could be described as a set of I/O ports. The IDE interface has a 8/16 bits I/O bus, two /CS lines, a /WR and /RD line, three address bits and one interrupt. In this description I assume the most traditional IDE interface. In later IDE interfaces a series of nice so-called PIO modes where added. These PIO modes add things like a ready line, DMA facilities and higher speed data transfers. As you read on you will understand that I only use the so-called PIO mode 0. This is the slowest communication modus on an IDE bus. It is also the easiest one to implement. The data bus on an IDE interface is used mostly for 8-bits transfers. Only the real disk data reads and writes use the 16-bits bus in full width. You COULD even implement an IDE interface with an 8-bits only data bus. That would mean that you use only half the disk capacity (the lower bytes of the 16-bits-wide bus) but that should work.

When scanning the net I did find an implementation of an IDE interface for 8-bit controllers. This interface was for a (hope I have this correct..) CoCo bus. It was implemented in TTL, just like the magazine's interface. The main idea was that whenever a (16-bits) word was read from the IDE bus the upper 8 bits where stored in a latch. The controller could retrieve them from the latch later. Writing to the IDE bus was implemented in the same manner. The IDE bus read/write cycles where in fact simple bus read and write cycles. At first I was about to copy this design. When thinking about it I thought that this TTL design was too complex for what I wanted to do. You need quite some TTL to implement a 16-bits read/write I/O port in TTL on an 8-bits controller.

Hardware description

When implementing a 16-bits I/O port all you need is a bidirectional I/O port and some control bits to generate the /RD, /WR etc... That is when the 8255 came in view. An 8255 has 3 8-bits I/O ports. It can be switched from output to input and back under software control. I used 2 of the 8-bits I/O ports for the data path and use port to generate the IDE control signals. The 74HC04 came into the design later. Once I had the controller and the 8255 strapped together with the IDE connector and a disk I found out that the 8255 has a nasty trait. Whenever you switch the I/O modus of the chip it resets ALL its memory bits. That includes ALL output signals too. For the data bus that is not so much of a problem. The control signals get a real shake when this happens. In particular: The /RESET line of the interface is activated. That makes all control of a disk on this interface impossible (the disk gets a reset at all kinds of odd moments...). I have solved this by simply inverting all the control signals from the 8255 to the IDE bus. When the modus of the 8255 is switched all outputs of the chip go to '0'. That means that all the (low-active) control signals are made inactive by the inversion. That is -in fact- the state where I have them already when I'm about to change the 8255's modus.

At this point I would like to present a nice circuit diagram to show what the contraption I have made looks like. Unfortunately I know of no easy way to do that. This beautiful net is a marvel when it comes down to transporting text, graphics is another matter. A GIF picture would do the work; I do not have any means to produce one. Some schematics drawing package could give a good picture; I have no schematics package and I am not sure what package would be universal enough to be useable by everyone. So I am restricted to a more or less cryptic ASCII description of the hardware. Please, the cryptology is out of need, not out of my liking. Well here it comes:

1) The IDE bus pin connections themselves:

The IDE connector itself is a 40-pins two-row connector:

```
    1                   39    odd-numbered pins
    ....................
    ....................
    2                   40    even-numbered pins
```

In an IDE bus this connector is used as follows:

```
pin no:      name:     function:
---------   -------    ---------

1            /RESET    Al low signal level on this pin will reset
                       all connected devices

2,19,22      GND       ground, interconnect them all and tie to
24,26,30               controller's ground signal
40

3,5,7,9,11   D7..D0    low data bus, 3=D7 .. 17=D0. This part of
13,15,17               the bus is used for the command and
                       parameter transfer. It is also used for
                       the low byte in 16-bits data transfers.

4,6,8,10     D8..D15   high data bus, 4=D8 .. 18=D15. This part
12,14,16,18            of the bus is used only for the 16-bits
                       data transfer.

20           -         This pin is usually missing. It is used to
                       prevent mis-connecting the IDE cable.

21 and       /IOREADY  I do not use or connect to this pin. It is
27                     there to slow down a controller when it is
                       going too fast for the bus. I do not have
```

		that problem...
23	/WR	Write strobe of the bus.
25	/RD	Read strobe of the bus.
28	ALE	Some relic from the XT time. I do not use it, and I'm not the only one...
31	IRQ	Interrupt output from the IDE devices. At this moment I do not use it. This pin could be connected to a controller to generate interrupts when a command is finished. I have an inverter ready for this signal (I need a /IRQ for my controller, an IRQ is of no use to me..)
32	IO16	Used in an AT interface to enable the upper data bus drivers. I do not use this signal. It is redundant anyway, the ATA-3 definition has scrapped it.
34	/PDIAG	Master/slave interface on the IDE bus itself. Leave it alone or suffer master/slave communications problems. Not used (or connected to ANYTHING) by me.
35 33 36	A0 A1 A2	Addresses of the IDE bus. With these you can select which register of the IDE devices you want to communicate.
37 38	/CS0 /CS1	The two /CS signals of the IDE bus. Used in combination with the A0 .. A2 to select the register on the IDE device to communicate with.
39	/ACT	A low level on this pin indicates that the IDE device is busy. I have connected a LED on this pin. The real busy signal for the controller I get from the IDE status register.

Input/Output status of these signals:

The signals:
A0, A1, A2, /CS0, /CS1, /WR, /RD and /RESET are always outputs from the controller to the IDE bus.

The signals:
IRQ and /ACT are always outputs from the IDE bus to the controller (IRQ can be tri-stated by the IDE device when two devices are connected to the IDE bus.) /ACT can drive a LED (with resistor of course).

The signals:
D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15 are bi-directional. They are output from the controller to the IDE bus when writing, output from the IDE device to the controller when reading information.

2) The 8255 <-> controller

The 8255 is connected to the controller by means of its 8-bit data bus, the A0 and A1 lines and the /CS, /WR and /RD lines. I can not give that much info about this. If in doubt: consult the 8255 data sheet. This part depends as much on the controller you use as anything else. I have the 8255 connected to my controller

(HD63B03R1CP, a Hitachi 6803-derivate..) as an I/O port. Perhaps some of you have seen my previous e-mails asking for the pinout of the HD63B03R1CP (52-pins PLCC) chip, well I did find it (Some department of my work had an old Hitachi databook, voila the pinout was there). My address decoding puts the 8255 on address 0500H to 0503H in the controller's memory map. That may help if you decide to try to make sense of my software listing. On this point you are on your own as to the how to connect a 8255 to your controller.

3) 8255 <-> IDE connector

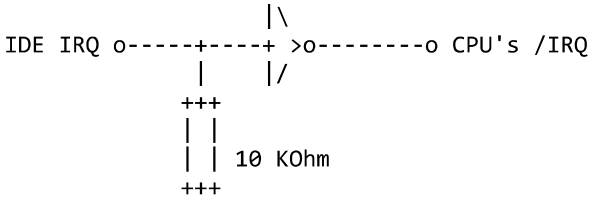
I have used the 8255's port A to generate the IDE bus control signals. Some of these control signals pass through an inverter before I connected them to the IDE connector itself. All of these signals are always used as output from the 8255 to the IDE bus. When a control signal is inverted the 8255 pin is connected to one of the inputs of the 74HC04 and the (corresponding) output of the 74HC04 is connected to the IDE bus connector.

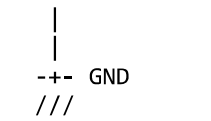
The ports B and C are used as the 16-bits data bus. There are no special things in this, it's just a simple interconnection of the 8255 I/O pins to the D0..D15 pins of the IDE connector.

I have connected this as follows:

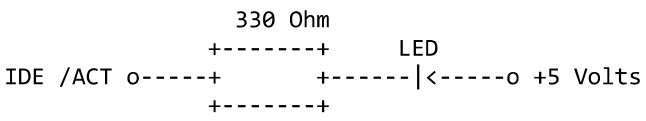
```
PA.7  --> inverter --> IDE bus /RESET
PA.6  --> inverter --> IDE bus /RD
PA.5  --> inverter --> IDE bus /WR
PA.4  --> inverter --> IDE bus /CS1
PA.3  --> inverter --> IDE bus /CS0
PA.2  -->              IDE bus A2
PA.1  -->              IDE bus A1
PA.0  -->              IDE bus A0

PB.7 <--> IDE bus D7
PB.6 <--> IDE bus D6
PB.5 <--> IDE bus D5
PB.4 <--> IDE bus D4
PB.3 <--> IDE bus D3
PB.2 <--> IDE bus D2
PB.1 <--> IDE bus D1
PB.0 <--> IDE bus D0

PC.7 <--> IDE bus D15
PC.6 <--> IDE bus D14
PC.5 <--> IDE bus D13
PC.4 <--> IDE bus D12
PC.3 <--> IDE bus D11
PC.2 <--> IDE bus D10
PC.1 <--> IDE bus D9
PC.0 <--> IDE bus D8
```

I have put a 10 KOhm pull-down resistor from the IDE bus IRQ to ground. The IDE IRQ signal is also connected to the input of the (one-remaining) inverter. The output of the inverter is connected to the controller's /IRQ input. As you can see, I do have the hardware for interrupts here, I do not use it. I tried to use it, but got unexplained errors (I probably did something wrong, I have not yet found what)..





The IDE /ACT signal is connected to a 330 Ohm resistor, the other end of the resistor is connected to a LED, the other end of the LED is connected to the +5 Volts. This gives a nice LED indication of when I'm using the disk. This is -as far as I know- the same hardware a PC uses to produce the disk busy LED you may find on the front of a PC box.



So much about the hardware of the IDE interface. I hope that is all clear now. If I have been less than clear, please ask. If I have made mistakes, please complain (I said complain, NOT FLAME!!!).

IDE read/write and register description

The IDE device appears to the IDE bus as a set of registers. The register selection is done with the /CS0, CS1 and A0, A1, A2 lines. Reading/writing is done with the /RD and /WR signals. I have used the 8255 port A signals to make read/write cycles on the IDE bus. What I do is the following:

read cycle:

1) put the port B and C of the 8255 in input modus.

2) set an address and /CS0 and /CS1 signal on the port A of the 8255.

3) activate the /RD of the IDE bus by setting the equivalent bit in the port A of the 8255.

4) read the data from port B (and C) of the 8255.

5) deactivate the /RD signal on the IDE bus by resetting the equivalent bit of port A of the 8255.

write cycle:

1) put the port B and C of the 8255 in output modus.

2) set an address and /CS0 and /CS1 signal on the port A of the 8255.

3) set a data word (or byte) on port B (and C) of the 8255.

4) activate the /WR of the IDE bus by setting the equivalent bit in the port A of the 8255.

5) deactivate the /WR signal on the IDE bus by resetting the equivalent bit of port A of the 8255.

The only difference between 8 bits and 16 bits transfers is the following:

- In an 8-bit transfer I use only the port B of the 8255 for data transfer. When writing I put data only on the lower byte of the IDE bus; the upper byte is ignored anyway

by the IDE device. When reading I read only port B of the 8255; I never even bother to look at the upper byte.

- In an 16-bit transfer I read/write to both the upper and the lower byte of the IDE bus; thus using both port B and port C.

The IDE device appears as the following registers:

/CS0=0, /CS1=1, A2=A1=A0=0: data I/O register (16-bits) This is the data I/O register. This is in fact the only 16-bits wide register of the entire IDE interface. It is used for all data block transfers from and to the IDE device.

/CS0=0, /CS1=1, A2..A0=001B: This is the error information register when read; the write precompensation register when written. I have never bothered with the write precompensation at all, I only read this register when an error is indicated in the IDE status register (see below for the IDE status register).

/CS0=0, /CS1=1, A2..A0=010B: Sector counter register. This register could be used to make multi-sector transfers. You'd have to write the number of sectors to transfer in this register. I use one-sector only transfers; So I'll only write 01H into this register. I do use this register to pass the parameter the timeout for idle modus command via this register.

/CS0=0, /CS1=1, A2..A0=011B: Start sector register. This register holds the start sector of the current track to start reading/ writing to. For each transfer I write the start sector in this register. For some fancy reason the sector count starts at 1 and runs up to 256, so writing 00H results in sector number 256. Why that is done is a mystery to me, all other counting in the IDE interface starts at 0.....

/CS0=0, /CS1=1, A2..A0=100B: Low byte of the cylinder number. This register holds low byte of the cylinder number for a disk transfer.

/CS0=0, /CS1=1, A2..A0=101B: High two bits of the cylinder number. The traditional IDE interface allows only cylinder numbers in the range 0..1023. This register gets the two upper bits of this number. I write the cylinder number's upper two bits into this register before each transfer.

/CS0=0, /CS1=1, A2..A0=110B: Head and device select register. The bits 3..0 of this register hold the head number (0..15) for a transfer. The bit 4 is to be written 0 for access to the IDE master device, 1 for access to the IDE slave device. The bits 7..5 are fixed at 101B in the traditional interface.

/CS0=0, /CS1=1, A2..A0=111B: command/status register. When written the IDE device regards the data you write to this register as a command. When read you get the status of the IDE device. Reading his register also clears any interrupts from the IDE device to the controller.

/CS0=1, /CS1=0, A2..A0=110B: 2nd status register/interrupt and reset register. When read this register gives you the same status byte as the primary (/CS0=0, /CS1=1, A2..A0=111B) status register. The only difference is that reading this register does not clear the interrupt from the IDE device when read. When written you can enable/disable the interrupts the IDE device generates; Also you can give a software reset to the IDE device.

/CS0=1, /CS1=0, A2..A0=111B: active status of the IDE device. In this register (read-only) you can find out if the IDE master or slave is currently active and find the currently selected head number. In a PC environment you can also find out if the floppy is currently in the drive. I have not used this register yet.

Some of these registers have bitwise meanings. I'll elaborate on that here:

head and device register:
A write register that sets the master/slave selection and the head number.

```
bits 3..0: head number [0..15]
bit  4   : master/slave select: 0=master,1=slave
bits 7..5: fixed at 101B. This is in fact the bytes/sector
           coding. In old (MFM) controllers you could specify if
           you wanted 128,256,512 or 1024 bytes/sector. In the
           IDE world only 512 bytes/sector is supported. This bit
           pattern is a relic from the MFM controllers age. The
           bit 6 of this pattern could in fact be used to access
           a disk in LBA modus.
```

Status register:
Both the primary and secondary status register use the same bit coding. The register is a read register.

```
bit 0    : error bit. If this bit is set then an error has
           occurred while executing the latest command. The error
           status itself is to be found in the error register.
bit 1    : index pulse. Each revolution of the disk this bit is
           pulsed to '1' once. I have never looked at this bit, I
           do not even know if that really happens.
bit 2    : ECC bit. if this bit is set then an ECC correction on
           the data was executed. I ignore this bit.
bit 3    : DRQ bit. If this bit is set then the disk either wants
           data (disk write) or has data for you (disk read).
bit 4    : SKC bit. Indicates that a seek has been executed with
           success. I ignore this bit.
bit 5    : WFT bit. indicates a write error has happened. I do
           not know what to do with this bit here and now. I've
           never seen it go active.
bit 6    : RDY bit. indicates that the disk has finished its
           power-up. Wait for this bit to be active before doing
           anything (execpt reset) with the disk. I once ignored
           this bit and was rewarded with a completely unusable
           disk.
bit 7    : BSY bit. This bit is set when the disk is doing
           something for you. You have to wait for this bit to
           clear before you can start giving orders to the disk.
```

interrupt and reset register:

This register has only two bits that do something (that I know of). It is a write register.

```
bit 1    : IRQ enable. If this bit is '0' the disk will give and
           IRQ when it has finished executing a command. When it
           is '1' the disk will not generate interrupts.
bit 2    : RESET bit. If you pulse this bit to '1' the disk will
           execute a software reset. The bit is normally '0'. I
           do not use it because I have full software control of
           the hardware /RESET line.
```

Active status register:
This is a read register. I have -up till now- ignored this register. I have only one IDE device (a disk) on my contraption.

```
bit 0    : master active. If this bit is set then the master IDE
           device is active.
bit 1    : slave active. If this bit is set then the slave IDE
           device is active.
bits 5..2: complement of the currently active disk head.
bit 6    : write bit. This bit is set when the device is writing.
```

```
bit 7    : in a PC environment this bit indicates if a floppy is
           present in the floppy drive. Here it has no meaning.
```

error register:
The error register indicates what went wrong when a command execution results in an error. The fact that an error has occurred is indicated in the status register, the explanation is given in the error register. This is a read register.

```
bit 0    : AMNF bit. Indicates that an address mark was not
           found. What this means I not sure of. I have never
           seen this happen.
bit 1    : TK0NF bit. When this bit is set the drive was not able
           to find track 0 of the device. I think you will have
           to throw away the disk if this happens.
bit 2    : ABRT bit. This bit is set when you have given an
           indecent command to the disk. Mostly wrong parameters
           (wrong head number etc..) cause the disk to respond
           with error flag in the status bit set and the ABRT bit
           set. I have gotten this error a lot when I tried to
           run the disk with interrupts. Something MUST have been
           wrong with my interface program. I have not (yet)
           found what.
bit 3    : MCR bit. indicated that a media change was requested.
           What that means I do not know. I've ignored this bit
           till now.
bit 4    : IDNF bit. Means that a sector ID was not found. I have
           never seen this happen, I guess it means that you've
           requested a sector that is not there.
bit 5    : MC bit. Indicates that the media has been changed. I
           ignore this bit.
bit 6    : UNC bit. Indicates that an uncorrectable data error
           happened. Some read or write errors could provoke
           this. I have never seen it happen.
bit 7    : reserved.
```

Note: I have these registers descriptions from two sources:

```
1) The C't magazine I mentioned before. It's in German, not very
   complete (the error register description is missing) and very
   old. It does have a hardware description of and IDE interface
   however....

2) The document X3T10/2008D: Information Technology-
   AT Attachment-3 Interface (ATA-3), Working draft.
   This latter document gives an exhaustive overview of the IDE
   interface. It states more details of the IDE interface than I
   can ever hope to include in this short description. It does
   however have one disadvantage: it's BIG. I found the document
   on the net (on the Western Digital homepage) in the form of an
   .exe file. When you run this file on a PC you are rewarded
   with a bigger-than-1 Mbytes .DOC file. That is a Word
   document. When you print (Wondows-95 has a Word viewer/printer
   application) it you get a nearly-200-pages paper. If you want
   to get into the IDE interface seriously I recommend you print
   this thing.
```

I hope the description I have given here will allow those that do not have a laser printer and a fast internet link available to get to grips with the IDE bus.

Intermezzo: Disk size limitations on the IDE bus and LBA modus

In the PC world there has been (and still is) a lot of discussion about 'limits' in the disk interface.

At first (MSDOS versions till 3.3) the disk interface was not able to access more than 32MB on one volume. That was a limitation of the MSDOS file system rather

than of the disk interface. The same DOS version that was unable to access bigger partitions than 32MB *WAS* able to access 650MB CDROMs. The limit came from the fact that each disk sector (512 bytes) was registered in the FAT in a 16-bits word. The total partition size was limited by the fact that only 65536 sectors could be addressed. The partition size was thus limited to 65536 x 512 bytes = 32 MBytes.

Later -as the disks became larger- the disk interface itself ran into its limits. The interface I describe here has room for 16 heads, 256 sectors per track and only 1024 cylinders. With the standard sector size of 512 bytes that leaves you a maximum disk size of 16 x 256 x 1024 x 512 = 2048 MBytes. That is a real limitation of the IDE interface as I describe it here. It can not access more than some 2 GBytes of disk space.

This was overcome by introducing the so-called LBA modus. In LBA modus the sectors are simply numbered from 0 to -big-. The lowest byte of the LBA sector number is written into the sector number register, the middle 16 bits of the LBA sector number are written in the cylinder number registers (low and high, all 16 bits are used). The highest 4 bits of the LBA sector number are written in the head and device register. That gives you 28 bits of LBA sector number. The sector size was again fixed at 512bytes, so in LBA modus you have access to: 2^28 x 512 = 1.37 E 11 (some 137.4 gigabytes) of disk space. This LBA modus has been made mandatory for all new disks (in the ATA-3 spec.) That should keep the disk makers busy for some while to come... If you want to connect a disk larger than 2 GBytes to this IDE interface you too will have to use the LBA modus. How to do that: the bit 6 of the head and device register is set to indicate that LBA modus is used (the fixed pattern of 101B in the bits 7..5 of the head and device register is to be changed into 111B). All other manipulation of the IDE interface is the same for Sector/Head/Cylinder modus and LBA modus.

All other limits in the MSDOS/Windows-whatever disk interface must be due to the BIOS implementation or the file system used. I can find no reason in the IDE definition for 512 MB limits or 8 GB limits at all.

End of intermezzo

IDE registers usage

Ok, now you know what registers the IDE system uses. Next question: How to use them? I do not pretend I have tried every possibility with these registers. As I stated before I have restricted myself to reading/writing data blocks to/from the disk. What to do for that is in fact fairly simple:

```
1) Before doing anything with a device you have to wait till it
   indicates that it is ready (RDY bit in the status register)

2) Next you load the parameters of a command into the appropriate
   registers. For read/write commands that comes down to writing
   the cylinder/head/sector numbers into the registers.

3) You issue a read or write command.

4) You wait till the device signals that it is ready for data
   transfer (DRQ in the status register).

5) Feed the device data (for write) or get the data from the
   device (for read). In case of a write you could wait for the
   operation to complete and read the status register to find out
   what has become of your data.
```

6) Finish!! That's all folks! The IDE interface is a surprisingly simple thing to get to work. If only I had an IDE disk and this kind of information when I was still programming my MSX-computer I'd have had a harddisk connected to it in no time.

IDE commands

What has been missing in this description till now is the command set. I do not think I can describe the complete command set here. The ATA-3 document is a better source for that than I can give here (All I would be doing is re-entering the ATA-3 document; I have neither the time nor any liking for that). The most useable commands I do intend to describe. Mind: When giving a command you first have to wait for device ready, next put the command parameters in the registers and only then can you give a command (by writing a command byte to the command register). The disk will start executing the command right after you've written the command into the command register.

IDE command:	Description:
1XH	recalibrate the disk. NB: 1XH means that the lower nibble of the command byte is a don't care. All commands 10H..1FH will result in a recalibrate disk command being executed. This command has no parameters. You simply write the command code to the command register and wait for ready status to become active again.
20H	Read sector with retry. NB: 21H = read sector without retry. For this command you have to load the complete circus of cylinder/head/sector first. When the command completes (DRQ goes active) you can read 256 words (16-bits) from the disk's data register.
30H	Write sector (with retry; 31H = without retry). Here too you have to load cylinder/head/sector. Then wait for DRQ to become active. Feed the disk 256 words of data in the data register. Next the disk starts writing. When BSY goes not active you can read the status from the status register.
7XH	Seek. This normally does nothing on modern IDE drives. Modern drives do not position the head if you do not command a read or write.
ECH	Identify drive. This command prepares a buffer (256 words) with information about the drive. If you want the details either look closely at the interface program I will add at the end of this description or get the ATA-3 document. To use it: simply give the command, wait for DRQ and read the 256 words from the drive. I have found that the modern drives I used give nice information about number of heads,sectors,cylinders etc... One of the disks I tried (a Miniscribe 8051A) gave wrong answers in this buffer. The disk is actually a 4 heads/28 sectors disk. It should be used in a translated modus with 5 heads/17 sectors. In the ident drive response it reported as 4 heads/28 sectors and it will NOT work in that modus. Two other disks (a Quantum 127 MB

	disk and a Western Digital 212 MB disk) report nicely. If not for the Miniscribe I would use the parameters reported to auto-config the interface to match the disk configuration.
E0H	Spins down the drive at once. No parameters. My Miniscribe 8051A does not respond to this command, the other disks do execute this command.
E1H	Spins up the drive again. Same remarks as E0H command.
E2H and E3H	Auto-power-down the disk. Write in the sector count register the time (5 seconds units) of non-activity after which the disk will spin-down. Write the command to the command register and the disk is set in an auto-power-save modus. The disk will automatically spin-up again when you issue read/write commands. E2H will spin-down, E3H will keep the disk spinning after the command has been given. Example: write 10H in the sector count register, give command E2H and the disk will spin-down after 80 seconds of non-activity. BTW: You can use this command easily on a PC disk too. The harddisk of the computer I am working on now gets this exact command at boot. That saves a lot of noise when I'm typing long stories like this one.
F2H and F3H	The same as E2H and E3H, only the unit in the sector count register is interpreted as 0.1 sec units. I have not tried this command. I think it will work (the E0H/E1H/E2H/E3H commands work, why should this one not work?)

Two-devices considerations

The IDE bus is intended for two devices. A master and a slave device. I have not tried anything myself, but the descriptions indicate that it is in fact very simple to connect two devices to the IDE bus. All you have to do is:

1) Configure the master/slave jumpers of the devices.
2) Select a device before you start giving commands to the devices.

The head and device register has the bit you need to switch from one device to another. You have to write the bit to either 0 for master or 1 for slave and start controlling the other device. Mind: BOTH devices will get their registers WRITTEN. Any data or register READ will come from the selected device. ONLY the selected device will execute commands.

Conclusions and ravings

ravings:

This description should be about what you need to connect an IDE disk to any controller. The only thing I have left out are my unsuccessful experiments with the interrupt. What happened there is that I enabled the interrupt, made an interrupt handler that simply read the status register (to get the interrupt to disappear) and re-scheduled the disk interface task. I was rewarded with occasional errors. Some of the read requests got an ABRT error. I do not yet know what I did wrong. I can not have been far off the mark, because most of the commands where executed

without comment from the disk. I do intend to try the interrupt modus again later. I theory the interrupt modus should give me a slightly bigger data rate. I have found data rates in the order of 32 KBytes per second when I was using interrupts.

About interrupts: When you want to use the interrupt mechanism all you have to do is enable the interrupt modus of the interface by clearing the interrupt disable bit in the reset and interrupt register. The disk will generate an interrupts as soon as it has completed a command. That means that it will generate an interrupt when it has read a sector from disk (as soon as DRQ gets active) or when it has finished writing a sector to disk. I am not sure about the other commands, but the description says that the disk will generate an interrupt upon the completion of each command.

About the data rate. This IDE interface will never win any award for its speed. It is in one word slow. I get data transfer rates in the order of 25 KBytes per second. I spend most of the time reading/writing data words from/to the disk on a word-by-word basis under software control. On the other hand, my controller has a total (RAM) memory of 24 KBytes. With the current interface I can dump the complete memory to disk in less than one second. That in itself makes me think that for this application the low speed is not so much of an issue.

About the future. I think I will give interrupt modus another try. Not because it gives me a fantastic data rate, I just can't stand it that it does not want to work good.

I am thinking about a proper file system to put on the disk. That will enable me to access the disk in a more normal way that the current 'sea of sectors' approach I use now. A FAT filesystem looks like an absolute horror when combined with a controller, so I'll have to either find something that looks good to me or strap something together myself.

I think I will give the ATAPI command set a try. I have just found and printed the 'SFF committee specification of ATA Packet Interface for CDROMs' (SFF-8020i) document. That document gives a description of how to control an ATAPI CDROM via the IDE bus. On the other hand: I would really not know what to begin with a CDROM connected to a microcontroller. But what the heck, I have no really good idea what to do with a microcontroller with a 40MB harddisk connected..... I have seen occasional questions in this mailing list about how to control a CDROM via its backside connector: I think this IDE interface with an ATAPI piggy-back software could just do the work.

Till now I've been concentrating on only one side of the interface. The IDE interface is in fact no more than a ready-decoded, buffered interface of a (mostly-PC) hardware system. I could in fact connect god-knows-what at the other end. I have an interrupt, DMA and I/O available, what else could you ask for? A disk or CDROM is only one of the less inspired devices you can connect to an IDE interface. Given the fact that modern PC-motherboards have two IDE interfaces aboard I can think of some nice things I'd like to connect to them. How about some 8 parallel input/output ports, how about a data acquisition system?

Lately I have been able to get a C-mos version of the 8255. My controller system is quite low-power (10 mA for the entire thing). I did not feel good about the IDE extension with a 8255 that used some 80 mA all by itself. The controller system with the D71055C (a C-mos NEC version of the 8255) has gone down to its normal - about 10 mA- power usage. the change from N-mos 8255 to C-mos D71055C has had no other noticeable effects than a lower power usage.

Even when I have no disk connected to this contraption it delivers me three 8-bits I/O ports. I am thinking about what other things I could connect to that.

conclusions:

As you can see it is far from difficult to connect an IDE disk to some computer. My implementation with a 8255 and an inverter chip may not be the fastest thing on earth, it works, it's simple and may be useable for many applications where a controller with a large backup store could save the day.

I'll wrap this up with a case-study: The software I use on my 63B03 controller to control the IDE bus. I hope it can clarify any points I have not covered in this description. I know this software to work, I hope it can give someone inspiration to do something similar on some controller system somewhere.

Appendix:

The software I use to control a single IDE harddisk using a 63B03 controller system and a 8255/74HC04 IDE interface.

Notes:

The interface program has been made on a 63B03 controller in assembly code. The 63B03 code is nearly 6800 code. If in doubt, consult some source of 6800 or 6803 or 63B03 instructions.

The source of the interface program is of course the most complete description of how to program this interface, but certainly not the most un-cryptic description. I try to program without any 'dirty tricks' but who knows....

This interface program makes sporadic use of the underlying 63B03 real-time schedular. If you see a 'jsr Sys' somewhere that is where I called the schedular to do something for me. As far as I can see I only use Sys to produce a 10-ms delay. I have left the code of the schedular itself out. Not because it is anything but public domain, but because it is another 150 KBytes of source file. This description is long enough as it stands here. The schedular is not really needed, so I left it out. If you are interested in the schedular as such, drop me a line (the address is a little up in the text).

The IDE interface software as such I hereby donate to the public domain. Use it as pleases you. Change it when you think it is wrong, suffer the consequences if you make mistakes. I do not take responsibility if you blow up harddisks, interface hardware or if anything goes wrong when you use it. I have found it to work on my system and, so sorry, that is all the guarantee I can give you about it.

63B03 assembly listing

Disk interface task for the 63B03 system
Update history:

02-02-1998 :	Started with the thing. The hardware has been finished today. All seems to work. Now I need software to get some life into it.
05-02-1998 :	Hit a bloody trick of the 8255: When you change the modus byte ALL outputs are set to 0. I now know why nobody wants to use this bugger. The solution for this shit is easy: I plan to put an inverter in all the control lines. The following lines will be inverted: /CS0, /CS1, /IORD, /IOW, /RES, IRQ (the latter for the

63B03, it has a /IRQ input).. This is the software change needed to make things work again...

06-02-1998 : Hardware change done, disk reset works Started testing the disk interface functions. The recalibrate command works too, the stop disk/start disk functions seem to be not present in this 40 MB disk. I also made a LBA routine, I can now access the disk as a set of sequential blocks, without counting heads etc.. It seems the PC hardware starts counting sectors starting at 1 (WHY???), all other numbers start at 0...

My disk (about 40 MB) tells me it has 980 cyls, 5 heads, 17 secs/track. In fact it does tell me different numbers, only the label indicates that it has been translated...

At this moment I only support disk block read and disk block write. I want to make some file- system too. Besides, this thing works code-bound. I want to start using interrupts too...

08-02-1998 : /IRQ hardware made ok. The disk can now generate interrupts. These will come into the system via the CPU's /IRQ signal. I even crashed the system by giving /IRQ's with no handler in place... Start making the disk software /IRQ-driven.

10-02-1998 : Changed the interrupt generation/detection software Found one nasty bug in the interrupt usage as I did it: I first gave a disk reset on the IDE bus, then started giving commends right away. On this ONE occasion I do NOT have the disk's interrupt facility available, the disk is still executing its internal reset. I have to wait for ready there, THEN start issuing commands... Also set the bus control signals to output BEFORE I start using the bus at all... Now the interrupt mechanism works like the beauty it is...

11-02-1998 : Cleaned up the data transfer code. The disk I/O was very slow due to very systematic code. ReadWord/WriteWord code substituted in the ReadBlock/WriteBlock code. Routines ReadWord/ WriteWord removed from the code. There was an error in the writeword routine, I've removed it.

15-02-1998 : I have given up about the interrupt-driven disk control. It KEEPS on giving unexplained errors. Does NOT want to work independently of the disk type and is a general pain in the ass. What exactly goes wrong I have no idea about. I now use the scheduler's delay routines to get ready status from the disk and that works fine. Both the 40 MB disk and the 127 MB disk run like the sunshine in this modus. The non-interrupt modus does in fact not make any significant difference for the transfer speed. I dropped from 32 KB/s to 25 KB/s. That is very acceptable for this microcontroller. The controller is in this modus a lot slower than the disk....

I'm now going to set the thing up for the 127 MB disk with proper track/head numbers. Works ok too.

Next test is with the 212 MB disk. I have to take care that I stay below track 600 for this disk, tracks 630 .. 660 are bad... Sunshine again..

16-02-1998 : Made Identify working. It now dumps a nice display of what the

disk can do. My antique 42MB disk gives only half an answer (and a wrong one as well, it does NOT tell about the 5 heads,17 sectors translation it does..) but the other ones like this ident command/display a lot. Also shifted the buffers in memory a bit. I now want to start writing to the disk (till now I have only been reading...). Ok, I really messed up the data on my 42 MB disk, but it DOES write as well as read. I get back what I have written, so far, so good. Now I have to make:

1. A proper disk I/O task. That means that I have to implement some way of communicating with the disk I/O task. signals? I REALLY would like to use some sort of mailbox mechanism. For that I will have to extend my scheduler.
2. Some sort of file-system. I am already contemplating a MFS (Microcontroller File System) for some time now. It's about time to start working on one..

18-02-1998 : Started making task # 0E into a disk monitor.

I have left out the remainder of the code as it is fairly specific to the processor it was written for.
